SecureMind: A Framework for Benchmarking Large Language Models in Memory Bug Detection and Repair

Huanting Wang
University of Leeds
Leeds, United Kingdom
schwa@leeds.ac.uk

Dejice Jacob
University of Glasgow
Glasgow, United Kingdom
Dejice.Jacob@glasgow.ac.uk

David Kelly
University of Glasgow
Glasgow, United Kingdom
2604833K@student.gla.ac.uk

Yehia Elkhatib
University of Glasgow
Glasgow, United Kingdom
Yehia.Elkhatib@glasgow.ac.uk

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom
Jeremy.Singer@glasgow.ac.uk

Zheng Wang
University of Leeds
Leeds, United Kingdom
Z.Wang5@leeds.ac.uk

Abstract

Large language models (LLMs) hold great promise for automating software vulnerability detection and repair, but ensuring their correctness remains a challenge. While recent work has developed benchmarks for evaluating LLMs in bug detection and repair, existing studies rely on hand-crafted datasets that quickly become outdated. Moreover, systematic evaluation of advanced reasoning-based LLMs using chain-of-thought prompting for software security is lacking. We introduce SecureMind, an open-source framework for evaluating LLMs in vulnerability detection and repair, focusing on memory-related vulnerabilities. SecureMind provides a user-friendly Python interface for defining test plans, which automates data retrieval, preparation, and benchmarking across a wide range of metrics. Using SecureMind, we assess 10 representative LLMs, including 7 state-of-the-art reasoning models, on 16K test samples spanning 8 Common Weakness Enumeration (CWE) types related to memory safety violations. Our findings highlight the strengths and limitations of current LLMs in handling memory-related vulnerabilities.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Artificial intelligence.*

Keywords: Software bug detection, Bug repair, Large language models

ACM Reference Format:
Huanting Wang, Dejice Jacob, David Kelly, Yehia Elkhatib, Jeremy Singer, and Zheng Wang. 2025. SecureMind: A Framework for Benchmarking Large Language Models in Memory Bug Detection and Repair. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3735950.3735954

ISMM '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735954

1 Introduction

Large language models (LLMs) such as ChatGPT [4], Gemini [8], Llama [45], and DeepSeek [34] are emerging as powerful tools for detecting and fixing software bugs and vulnerabilities. Although LLMs have demonstrated remarkable capabilities in programming tasks, they still face a significant challenge – *correctness.*

Errors in bug detection and repair include failing to identify or fix bugs or explain the reason, generating false-positive predictions, or introducing new bugs. Undetected vulnerabilities can lead to severe security risks, while excessive false positives overwhelm developers and hinder adoption. Ensuring correctness is crucial for automatic bug fixing and code generation, as LLM-generated code may inadvertently introduce new bugs or vulnerabilities [21, 35, 39].

Since formally verifying LLM-generated content is still impractical [29], empirical evaluation using benchmark datasets remains the primary method for assessing LLM performance. However, existing benchmark datasets for code analysis [15, 46] predominantly rely on manually constructed test cases. While these datasets provide valuable insights, their coverage is inherently limited due to the expensive effort required to create high-quality test scenarios. Other benchmarking datasets, such as those based on competitive programming [32] or classroom-style coding tasks [10], fail to represent real-world software engineering tasks sensitive to security vulnerabilities. Additionally, data leakage poses a challenge: since LLMs are trained on public data, many benchmark cases may already be in their training set [43], resulting in misleadingly high performance and an inflated sense of the models' capabilities [9].

An automatic benchmarking framework is important for systematically testing LLMs' ability to detect and fix software vulnerabilities. Such a framework should reduce the need for manual test dataset collection, ensuring alignment with real-world software development. The framework must adapt to emerging vulnerabilities and evolving code patterns whilst mitigating data leakage by ensuring test samples fall outside an LLM's training data cutoff. Additionally, it should support test sample augmentation to assess LLMs' handling of code complexity. By providing an automated and reproducible evaluation process, such a framework helps identify weaknesses and improve LLM reliability and effectiveness for vulnerability detection and repair.

We present SecureMind[1], an open-source framework and dataset for systematically evaluating LLMs in vulnerability detection and repair. SecureMind automates benchmarking with customizable test plans, enabling a reproducible evaluation pipeline with minimal human intervention.

SecureMind offers a Python interface that allows users to define test plans with minimal effort. This can typically be achieved using only a few dozen lines of Python code to define testing parameters such as the target LLM, API credentials, and model knowledge cutoff date. Upon execution, it automatically retrieves and caches test samples from six public repositories, including the Common Vulnerabilities and Exposures (CVE) database and GitHub. To prevent data leakage, only datasets published after the LLM's training cutoff are included. To introduce adversarial challenges, SecureMind applies source code obfuscation and evaluates LLMs' ability to analyze optimized assembly. Code repair quality is validated using developer-created test inputs retrieved automatically, with syntax and functional correctness checked via static analysis [5, 11].

SecureMind provides an automated evaluation pipeline to systematically assess multiple aspects of LLM performance, including prediction accuracy, reasoning capabilities, and code repair correctness. By default, it quantifies LLM performance across six key dimensions: (1) response consistency, (2) prompt effectiveness, (3) reasoning ability, (4) vulnerability detection and repair effectiveness, (5) sensitivity to code obfuscations, and (6) robustness on assembly code. These criteria can be easily customized and extended via SecureMind APIs.

SecureMind is part of the community's efforts in developing benchmarks to evaluate LLMs in code-related tasks [46]. It differs from prior work by offering a customizable toolkit for test planning, automated data collection, and evaluation rather than relying solely on static datasets. Our study targets memory-related vulnerabilities such as buffer overflows and use-after-free errors, which account for a significant portion of software security issues [12, 50]. For instance, 70% of the most serious security bugs in the Chromium project are memory-related vulnerabilities [16]. While tested on memory-related vulnerabilities, SecureMind can be adapted to cover other software vulnerabilities.

We demonstrate the benefit of SecureMind by applying it to evaluate 10 leading LLMs for detecting and repairing eight vulnerability types. Our test set includes seven state-of-the-art reasoning LLMs, such as ChatGPT-o1 and DeepSeek-R1, which leverage *chain-of-thought* prompting [49] at inference time to enhance coding and logical reasoning.

Using SecureMind, we construct a dataset of over 16K memory-related vulnerabilities, sourced and augmented from six data sources, including online repositories (e.g., CVEs [1, 2]), developer-curated datasets (e.g., SARD [37]), and bug reports from GitHub and Bugzilla. The dataset spans C, C++, Java, Python, and x86 assembly code, making this the most comprehensive evaluation of reasoning-based LLMs for vulnerability detection and repair to date. Our evaluation demonstrates SecureMind's effectiveness in benchmarking LLMs, providing empirical insights into their strengths and weaknesses. For example, we find that LLMs perform poorly in automatic patch generation for real-life programs, with success rates ranging from 3% to 37% across the tested LLMs. Additionally, minor source code changes can cause LLMs to miss vulnerabilities or generate false positives.

This paper makes the following contributions:

- An automated and customizable framework for evaluating LLMs on identifying and repairing vulnerabilities;
- A large-scale evaluation of 10 state-of-the-art LLMs on vulnerability detection and repair;
- Identifying limitations of state-of-the-art LLMs, providing a checklist for researchers working in this space.

2 Background

2.1 Large Language Models

LLMs generate content based on user prompts (or queries). Techniques like instruction tuning *teach* LLMs to follow instructions effectively [41], while RLHF *trains* them to engage in human-like reasoning and conversation. This has led to the development of chat-based LLMs such as CodeLlama and ChatGPT-4o, which can handle interactive discussions.

2.2 Chain-of-Thought

The latest development in LLMs, such as ChatGPT-o1 and DeepSeek R1, incorporate *chain-of-thought* (CoT) reasoning [49] to enhance multi-step problem-solving. CoT structures reasoning steps explicitly, improving LLMs' ability to handle complex tasks requiring logical inference and contextual understanding. By decomposing problems into sequential steps, CoT-equipped models enhance performance in code-related tasks.

2.3 Model Parameters

Two key parameters affect LLM output: *temperature* and *top-p*. *Temperature* controls randomness - higher values (≥ 1.0)

[1] Code, documentation, test data, and full evaluation results are available at: https://github.com/HuantWang/SecureMind.

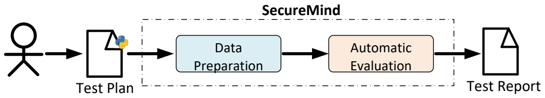


Figure 1. The overall workflow of SecureMind.

```
1   from SecureMind import DefinitionInterface
2   from openai import OpenAI
3
4   class MyInterface(DefinitionInterface):
5       def __init__(self, temp: list, top_p: list):
6           # Set the LLM parameters to be evaluated
7           super().__init__()
8           self.temp = temp
9           self.top_p = top_p
10
11      def set_model(self, model_name: str, api_key:
            str):
12          # Set the model to be tested
13          self.model = OpenAI(api_key=api_key)
14          self.model_name = model_name
15
16      def set_data(self, cutoff_date: str):
17          # Set the dataset cutoff date.
18          self.cutoff_date = cutoff_date
19
20  if __name__ == "__main__":
21      evaluator = MyInterface(temp=[...], top_p
            =[...])
22      llm = evaluator.set_model(model_name, api_key)
23      test_data = evaluator.set_data(cutoff_date="
            ...")
24      #Use the default evaluation pipeline
25      results = evaluator.evaluation(test_data, llm)
```

Figure 2. A simplified test plan using SecureMind APIs.

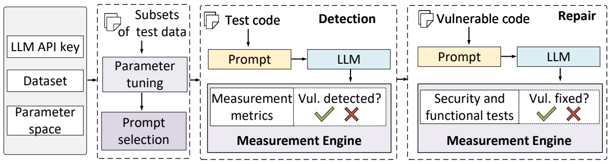


Figure 3. The SecureMind evaluation pipeline.

yield more diverse outputs, while lower values make responses more deterministic. *Top-p* selects from the smallest set of words whose cumulative probability exceeds a threshold *p*, adapting to the model's confidence. Higher *p* values (e.g., 0.9–1.0) increase variation but risk errors; lower values (e.g., 0.3–0.5) improve coherence. SecureMind allows users to specify temperature and top-p ranges during automated tuning (Sec. 5.1)

3 SecureMind Workflow

SecureMind is designed to be flexible and customizable, supporting the evaluation of any chat-based LLM compatible with the OpenAI API. Integrating a test LLM with SecureMind is straightforward, requiring minimal effort from the user. A basic setup involves writing a short Python script - usually just a dozen lines of code - to specify the API key, LLM knowledge cutoff date, and test configuration. SecureMind is highly customizable, allowing users to override relevant methods within its interface class to tailor and extend default test strategies to meet specific requirements.

Table 1. Vulnerability databases used by SecureMind.

Name	URL
Security advisories reported on GitHub	github.com/advisories
The National Vulnerability Database (NVD)	nvd.nist.gov
Common Vulnerabilities and Exposures (CVE)	cve.org
The Python Packaging Advisory Database	github.com/pypa/advisory-database
NIST Software Assurance Reference Dataset (SARD)	samate.nist.gov/SARD/
Bugzilla	bugzilla.org

3.1 Test Plan Program

Figure 1 depicts the workflow for using SecureMind, beginning with the definition of a test plan. A simplified Python implementation of this test plan is shown in Figure 2. The test plan defines methods from the SecureMind interface class, including specifying the model to be tested and the API key (lines 13, 14) required to query the model. Additionally, it sets the knowledge cutoff date of the test model, allowing SecureMind to prepare the appropriate test data. The example test plan follows the default evaluation method provided by the SecureMind interface, which returns a Python numpy data frame containing the evaluation results. However, users can override this method to customize the evaluation process and define their own metrics.

Figure 3 shows the automated evaluation pipeline of SecureMind. Upon executing the test plan, SecureMind automatically retrieves and synthesizes test data from public code repositories such as GitHub, NVD, SARD, and Bugzilla. It then automatically selects the appropriate model parameters (Sec.2.3) for detection and repair, before assessing the LLM's performance across a range of predefined metrics (Sec.3.4).

3.2 Automated Data Preparation

Given a test plan, SecureMind automatically retrieves and prepares test samples based on the knowledge cutoff date. It supports code samples written in C, C++, Java, and Python, and compiles C and C++ samples into assembly code.

By default, SecureMind downloads test samples for eight predefined CWE types, extensible via the test plan. It queries CVEs from six online sources (Table 1) using GraphQL [6], extracting the vulnerability description, affected GitHub repositories, CWE type, and commits. Samples missing these details are discarded. SecureMind then inspects each CVE's patch commits to locating the vulnerable code in earlier versions and retrieves the vulnerable function, its patched counterpart, and any relevant descriptions. Finally, vulnerable snippets are mixed with benign samples - sourced from other projects or generated by applying developer-written patches to CVE records - at a configurable ratio.

As this work targets memory-related bugs, we use Gemini-2.0-Flash to analyse vulnerability descriptions and download a sample only if the LLM confirms its relevance. Users can customise bug types and LLM choice via a SecureMind interface.

Table 2. Source code level code augmentation methods.

No.	Augmentation Method
A1	Replace variable names with vulnerability-related keywords
A2	Replace function names with vulnerability-related keywords
A3	Rename a vulnerable function's name to a non-vulnerable equivalent
A4	Introduce a potentially dangerous library function (e.g., `strcpy`, `strcat`) but use it safely
A5	Define safe function names using macros (e.g., `fgets`) while embedding vulnerable functions (e.g., `gets`) in their implementation
A6	Use sanitization functions (e.g., `realpath`) in vulnerable code without mitigating the vulnerability
A7	Use map-defined expressions for safe function names (e.g., `fgets`) while introducing vulnerable functions (e.g., `gets`) in their implementation
A8	Rename parameters
A9	Rename function names
A10	Insert dead code
A11	Add comments
A12	Modify whitespace
A13	Insert additional functions
A14	Insert new lines

3.2.1 Data augmentation.

SecureMind provides 14 source code augmentation methods, as detailed in Table 2, to obfuscate the collected source code and enhance dataset diversity. Following previous studies [46], our augmentation techniques include: 1) function and variable renaming, which evaluates the LLM's noise resistance by assessing whether it can correctly understand program functionality; 2) adding unreachable functions or code segments, which introduces structural noise to test the LLM's equivalence reasoning and determine whether it can recognize non-executable code; and 3) adding security-related segments, which obfuscate program logic to assess the LLM's security reasoning and deep code-understanding ability.

3.2.2 Assembly code.

Unlike prior work focused on high-level code (e.g., C, Python, Java), SecureMind also evaluates LLMs on optimized binaries. This is especially useful for evaluating LLMs when third-party source code is unavailable, and only assembly can be recovered from compiled binaries (e.g., 3rd party libraries) using tools like `objdump`. To support this, SecureMind compiles C and C++ code into assembly using standard compiler optimization levels (`O0` - `O3`).

3.3 Prompt Templates

Tables 3 and 4 list the built-in SecureMind prompt templates for vulnerability detection and repair. These can be customized or extended via SecureMind APIs. By default, SecureMind supports both zero-shot and few-shot prompts. Zero-shot prompts rely solely on the LLM's pre-trained knowledge, while few-shot prompts include examples to guide the model more effectively. Following [46], SecureMind provides prompts with step-by-step instructions to simulate CoT reasoning (e.g., C3 in Table 3, R8 in Table 4), mirroring how human experts approach vulnerability detection [47]. These CoT-based prompts also support comparison between models with and without explicit CoT generation. To reduce evaluation cost, SecureMind selects the most suitable prompt at the start using a small random sample. Users may override this by supplying a custom prompt set.

3.4 Built-in Evaluation Metrics

SecureMind provides a range of built-in evaluation metrics for vulnerability detection and repair, described as follows.

3.4.1 Deterministic score.

The deterministic score (ranging from 0 to 1) quantifies the consistency of model responses across multiple runs, irrespective of correctness. By default, SecureMind measures the deterministic score by running the test LLM on the same prompt 10 times per test input.

3.4.2 Metrics for detection.

For each test sample collected and synthesized by SecureMind, a ground truth label indicates whether it contains a vulnerability. SecureMind automatically evaluates LLM performance in vulnerability detection using four standard metrics: accuracy, precision, recall, and F1-score. Users can also define custom metrics by implementing their own evaluation methods.

3.4.3 Metric for bug repair.

For bug repair, we report the ratio of buggy test samples successfully fixed. SecureMind validates LLM-generated code through a two-step process. First, it uses CodeQL [5] to verify syntax correctness and identify any unfixed or newly introduced bugs. If the code is both compilable and executable, SecureMind then evaluates the generated binary using test cases from the corresponding project repository. These test cases, originally created by developers after applying human-written patches, are retrieved by SecureMind by tracing the vulnerable code samples back to their repository and associated patch commit. Users can also customize the validation process via the SecureMind API. To generate repair instances, we use the repair prompts listed in Table 4, instructing the LLM to produce 10 repair attempts per test sample per prompt. A repair is considered successful if at least one attempt passes both the syntax check and the test case execution.

3.4.4 Metrics for reasoning.

Besides assessing the test LLM's performance in classifying and repairing vulnerabilities, SecureMind provides an interface to evaluate the LLM's explanations for its decisions. Our objective is to assess the reasoning capabilities of LLMs and determine whether they can justify their decisions effectively. To achieve this, SecureMind prepares the ground truth explanation for a given vulnerability collected from a vulnerability database like CWE, and automatically evaluates the LLM's explanation using three quantified metrics proposed by Ullah et al [46]: (1) *Rouge* [33], (2) *Cosine similarity* [42], and (3) an *LLM score*.

Rouge. This metric evaluates LLM-generated reasoning text by measuring its n-gram overlap with the ground truth.

Cosine similarity. This measures the similarity between two vectors in an embedding space, where a higher value indicates greater similarity. In this study, we compute this score by converting the LLM-generated text and ground truth description into fixed-length vectors using Gemini's embedding model 'text-embedding-004', but other embedding models can be used when defining the test plan.

Table 3. SecureMind built-in prompt templates of vulnerability detection

ID	Few-shot	Description
D1	No	Does this code contain instances of the memory-related security vulnerability?
D2	No	Does this code contain instances of the memory-related security vulnerability known as {*cwe_type*}?
D3	No	Similar to D2, but with the LLM role set as a "helpful assistant."
D4	No	Similar to D2, but with the LLM role set as a "code security expert."
D5	No	You are a code security expert who analyzes the given code for the memory-related security vulnerability known as {*cwe_type*}.
D6	Yes	Similar to D2, but includes an example of a vulnerability and its corresponding patch, along with reasoning texts.
D7	Yes	Similar to D4, but includes the few-shot information as D5.
C1	No	Similar to D1 but use CoT prompt: "Let's think step by step."
C2	No	Similar to D2 but use CoT prompt: "Let's think step by step."
C3	No	Similar to D3 but use a multi-step prompt: 1. First, you describe the overview of the code. 2. Then, based on the overview, you identify the sub-components in the code that could lead to {*cwe_type*}. 3. After that, you conduct a detailed analysis of the identified sub-components for the existence of the {*cwe_type*} vulnerability. 4. Based on the detailed analysis, you determine and answer whether the {*cwe_type*} vulnerability is present in the given code.
C4	No	Similar to D3 but use a multi-round conversation: Provide a brief overview of the code. Based on the overview, identify the sub-components in the code that could lead to a memory-related security vulnerability known as {*cwe_type*}.
C5	Yes	Similar to C3 but with the role "helpful assistant" and add few-shot information as in D5.
C6	Yes	Similar to C2 but with few-shot information as in D5.
C7	Yes	Similar to C5, but without the role assigned.

Table 4. SecureMind built-in prompt templates of vulnerability repair

ID	Few-shot	Description
R1	No	Remove the vulnerable memory-related code/function body and replace it with a secure version.
R2	No	Remove the vulnerable memory-related code/function body known as {*cwe_type*} and replace it with a secure version.
R3	No	Similar to R2, but with the LLM role set as a "code security repair expert".
R4	No	Similar to R1, but with the LLM role set as a "code security repair expert".
R5	Yes	Similar to R2, but includes the few-shot information as in D5.
R6	Yes	Similar to R5, but with the role set as in R2.
R7	No	Similar to R2 but using the CoT prompt: "Let's think step by step".
R8	No	Similar to R2 but using a multi-step prompt: analyze the given code for {*cwe_type*} security vulnerabilities and systematically fix them by following these steps: 1. Remove insecure memory-related functions and replace them with safe alternatives. 2. Initialize all allocated memory before use to avoid uninitialized memory vulnerabilities. 3. Implement buffer overflow protection by ensuring all writes stay within buffer limits. Avoid unsafe pointer arithmetic and always validate pointer dereferences. 4. Enable stack canaries to detect and prevent stack-based buffer overflows. 5. Verify the fixed code: Ensure that the {*cwe_type*} vulnerability is fully mitigated.
R9	Yes	Similar to R7 but with few-shot information as in D5.
R10	Yes	Similar to R8 but with few-shot information as in D5 and with the role set as "code security repair expert".

LLM score. To compute the LLM-evaluated similarity, SecureMind instructs an LLM (Gemini-2.0-Flash in this work) to determine whether the response generated by the test LLM and the ground-truth reason are similar.

Compute reasoning metrics. Following the methodology in [46], we compute reasoning metrics by comparing an LLM-generated explanation (L_r) to the ground truth (G_r). Specifically, L_r is considered similar to G_r if its Rouge score and Cosine similarity exceed 0.34 and 0.84, respectively. Once three similarity scores are computed, SecureMind determines reasoning correctness through a majority vote: if at least two of the scores indicate similarity, the LLM's reasoning is classified as aligned with the ground truth.

4 Evaluation Setup

This section describes the parameters and experimental setup used by SecureMind to evaluate some of the state-of-the-art LLMs for detecting and repairing memory-related bugs. In this work, we focus on memory-related bugs because they are common, critical, and challenging for LLMs, which struggle with the complex reasoning required for pointers, heap management, and execution paths. However, SecureMind is applicable to other bug types, which we consider a strength.

4.1 Language Models

Table 5 lists the LLMs used in this study, including several state-of-the-art CoT-enhanced reasoning models that have not been previously evaluated for bug detection and repair in prior peer-reviewed publications.

Because each LLM may have different preferred prompting methods, SecureMind allows users to tailor prompt

Table 5. LLMs evaluated in this work

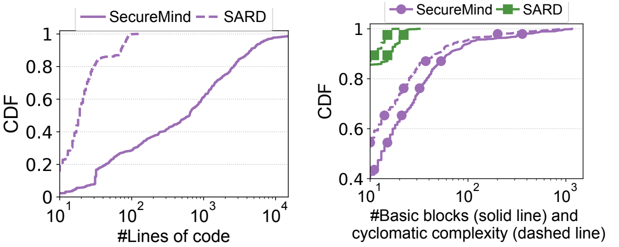
CoT	LLM	#Params.	Context len.	Gen. len.	Know. Cutoff
No	Llama-3.3-70B-Inst.	70B	128K	128K	12/2023
	Llama-3.1-405B-Inst.	405B	128K	128K	12/2023
	Qwen2.5-7B-Inst.	7B	128K	8K	04/2023
	Qwen2.5-Coder-32B-Inst.	32B	128k	8K	04/2023
at training	ChatGPT 4o	~ 200B	128K	16.4K	04/2023
	DeepSeek V3	671B	128K	8K	07/2024
	Gemini 1.5 PRO	~ 200B	128K	8K	09/2024
at infer.	ChatGPT o1	~ 200B	128K	128K	10/2023
	DeepSeek R1	671B	128K	8K	07/2024
	Gemini 2.0 Flash	40B	1M	8K	06/2024

Table 6. Vulnerable test samples used to evaluate detection; all were reported after Sept. 2024 - the latest knowledge cutoff date of the evaluated LLMs.

CWE	Description	#Raw samples	#Aug. samples
CWE-119	Improper restriction of operations within the bounds of a memory buffer (a.k.a buffer overflow)	120	1,800
CWE-125	Out-of-bounds read	180	2,700
CWE-190	Integer overflow or wraparound (focusing on memory leaks under this category)	24	360
CWE-415	Double free	198	2,970
CWE-416	Use after free	270	4,050
CWE-476	NULL pointer dereference	144	2,160
CWE-787	Out-of-bounds write	78	1,170
CWE-824	Access of uninitialized pointer	64	960

Table 7. Raw test samples per programming language.

Languages	#Samples
C and C++	934
Java	120
Python	24



(a)

(b)

Figure 4. The CDF of the number of lines (a), basic blocks, and cyclomatic complexity (which measures the complexity of a program's control flow [23]) (b) for our test samples and the SARD benchmark at log scale.

formatting to optimize interactions with the test model. In this study, we follow the guidance of LLM vendors to apply the recommended prompting techniques. This customization is achieved by overriding the default prompt method in the SecureMind API. For example, OpenAI's GPT documentation suggests enclosing the input content within triple quotes (‘"""‘) to clearly separate it from instructions [4].

4.2 Datasets

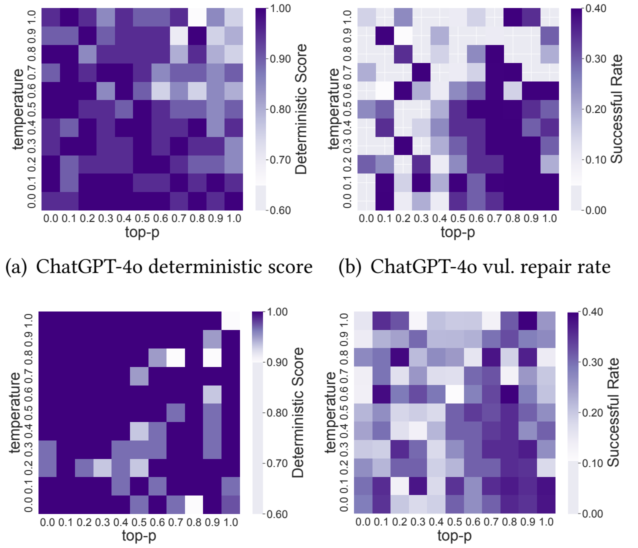
Raw test samples. We use SecureMind to automatically collect data from six databases (Table 1), yielding 1,078 test samples for vulnerability detection - split evenly between vulnerable and benign (patched) versions. Table 7 shows the language distribution of these samples. The cumulative distribution functions (CDF) in Figure 4 highlight some key characteristics of the SecureMind dataset (Table 6) compared to SARD (Table 1). Over 80% of our samples have at least 100 lines of code and 20 basic blocks, whereas SARD samples are simpler. This suggests that our dataset better reflects real-world program complexity. For vulnerability repair, we collect a smaller dataset of 23 samples, as SecureMind automatically retrieves developer-written patches and test cases, limiting sample availability.

Code augmentations. We use the code augmentation methods described in Sec. 3.2.1 to obfuscate both vulnerable and benign samples at the source code level, resulting in a total of 16K test samples for vulnerability detection.

4.3 Test Plan and Evaluation Platform

For this study, we define a test plan using the SecureMind API in less than 50 lines of Python. We then execute the plan on six Google Cloud instances, switching test LLMs via cloud-based LLM APIs. Each instance runs Ubuntu 20.04 with 16× Intel(R) Xeon(R) CPUs (2.20GHz) and 64GB RAM.

In total, our evaluation used over 5,000 CPU hours, generating more than 10 billion tokens from the tested LLMs. Most of this time was spent querying LLMs for evaluation, while data download and preparation took under 5 hours. To prevent overloading data source websites, SecureMind limits concurrent download requests by default.



(a) ChatGPT-4o deterministic score

(b) ChatGPT-4o vul. repair rate

(c) DeepSeek R1 deterministic score

(d) DeepSeek R1 vul. repair rate

Figure 5. Temperature and top-p settings for deterministic scores and vulnerability repair rates.

5 Evaluation Results

We use SecureMind to evaluate all LLMs listed in Table 5, reporting the built-in metrics from Sec. 3.4 along with illustrative examples. Following the default evaluation pipeline, we first tune the temperature and top-p parameters (Sec.2.3). SecureMind then identifies the optimal prompt template (Tables 3 and 4) for each LLM and assesses its reasoning ability in explaining decisions. Using the selected parameters, we further analyze LLM performance across varying code complexities and vulnerability types. All the quantified metrics are automatically generated by SecureMind as Python data frames and CSV files.

5.1 Parameter Tuning and Determinism

To robustly evaluate LLMs, ensuring response consistency is crucial - repeated tests for identical inputs should yield stable results. SecureMind begins with parameter tuning (Sec. 2.3), using 10% of randomly selected test samples to find the highest *temperature* and *top-p* values that still yield a deterministic score (Sec. 3.4.1) of at least 0.9.

Each parameter pair is tested across 10 trials, varying values from 0 to 1 in 0.1 increments. For tuning, SecureMind uses prompt D4 (Table 3) for detection and R4 (Table 4) for repair. These zero-shot prompts avoid prior context, making them well-suited for evaluating response stability.

Observations. Optimal parameter settings vary across LLMs. As shown in Figure 5, setting the temperature and top-p to

```
int open(wtap *wth, int *err, gchar **err_info)
{   ...
  wth->p = g_malloc(...);
  while (1) {
     if (file_seek(...)) == -1){
      g_free(wth->p); wth->p = NULL;
      return -1;
     }
     ...
    if (bytes_read != 4){
      *err = ...;
      if (*err != 0){
        g_free(wth->p); wth->p = NULL;
        return -1;
      }
      g_free(wth->p); wth->p = NULL;
      return 0;
  }}}

static void close(wtap *wth) {
    if (wth->p != NULL){
      g_free(wth->p);
      wth->p = NULL;}
```

(a) A simplified patch for a CWE-415 double-free vulnerability. The patch ensures that pointer p is set to NULL immediately after calling g_free().

ChatGPT-4o's incorrect response to the code in (a)

The code has multiple paths where memory (wth->p) is freed. If an error occurs and the caller subsequently calls close, a double free can occur. Multiple return paths also increase the risk, as the caller might unknowingly free the memory again. ✗

(b) A false-positive provided by ChatGPT-4o

ChatGPT-4o's correct response to the code in (a)

The code frees memory at each error exit point and sets the pointer to NULL to prevent accidental reuse. [...] The memory is managed correctly. ✓

(c) Correct responses given by ChatGPT-4o.

Figure 6. A benign code sample and inconsistent responses given by ChatGPT-4o for the same prompt.

0.2 and 0.1, respectively—values suggested by OpenAI for code-related tasks [3]—yields a deterministic score below 0.9 for ChatGPT-4o but exceeds 0.98 for DeepSeek R1 for vulnerability detection. Notably, even with a temperature of 0, full consistency is not guaranteed, with deterministic scores averaging only 0.96. For example, ChatGPT-4o (temperature 0.0, top-p 0.1) misclassifies a patched code snippet in Figure 6 as vulnerable in 1 out of 10 runs. As seen in Figure 5, increasing temperature and top-p values (e.g., ≥ 0.9) enhances response diversity and "creativity," improving the likelihood of generating a correct patch. Since optimal parameter settings depend on both the LLM and the task, choosing appropriate values is non-trivial. SecureMind addresses this challenge by automatically adjusting parameters based on user-defined criteria, such as ensuring a deterministic score above 0.9 while maintaining accuracy above 0.6.

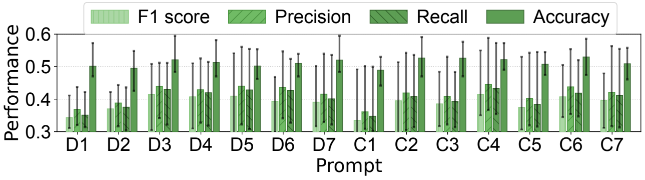


Figure 7. Vulnerability detection performance using different prompts. Min-max bars show the variances across tested LLMs and samples.

Response to prompt D1

Upon reviewing the provided code, I have identified a few potential issues related to memory management... ✗

Response to prompt D3

Yes, the code contains instances of double free. ... In the close function, wth->p is freed again without checking if it is NULL. ✗

Response to prompt D6

... The ‘close‘ function checks if ‘wth->p‘ is not ‘NULL‘ before freeing it, and then sets it to ‘NULL‘ to prevent any potential double free. Overall, the code appears to handle memory management correctly and does not contain any instances of double free. ✓

(a) Llama-405B's responses to prompts D1, D3, and D6.

Response to CoT prompt C1

... **let's analyze it step by step**... **1. Buffer Overflows... 2. Use-After-Free... ... 4. Other Issues**... In conclusion, based on the provided code snippet, there do not appear to be any obvious instances of memoryrelated security vulnerabilities such as buffer overflows, useafterfree, or memory leaks. ✓

Response to CoT prompt C2

... **1. Memory Allocation... 2. Freeing Memory...** 3. ... In conclusion, the provided code does not appear to contain instances of the double free vulnerability. ✗

Response to CoT prompt C3

Overview: ... The following **sub-components** ...1. Memory Allocation and Freeing in ‘open‘; ... 2. Potential Double Free... **Detailed Analysis** ... **Conclusion** The potential double-free vulnerability arises because if the function encounters an error while reading the next record type and length, it frees the allocated memory and returns -1. If the function is called again, it will attempt to free the same memory again, leading to a double free. ✗

Response to CoT prompt C7

Overview: ... **sub-components**: S1: Memory allocation for ‘wth->p‘ in ‘open‘. S2: ... S3: ... **Analysis** A-S1: The memory for wth->p is allocated using g_malloc ... Decision: It seems that the authors have taken necessary precautions to avoid double-free issues ✓

(b) Llama-405B's responses to CoT prompts C1-C3 and C7.

Figure 8. Llama-405B's responses to standard and CoT prompts to detect vulnerability for the code in Figure 6(a).

5.2 Prompt Template Selection

SecureMind automates the evaluation of prompt effectiveness on LLMs for vulnerability detection and repair using the templates in Sec. 3.3. Such an evaluation provides insights into prompt effectiveness, supporting the development of more effective prompt strategies. In this experiment, we use the parameters during the parameter tuning stage for individual LLMs (Sec. 5.1). This evaluation is conducted on 16K test samples for vulnerability detection and 23 samples for vulnerability repair.

5.2.1 Vulnerability detection.

Figure 7 presents the performance of different prompts for vulnerability detection, with min-max bars indicating variation across LLMs and

Table 8. Repair success rate of individual prompts of Table 4.

Prompts	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Rep. rate	27%	25%	22%	29%	29%	33%	25%	33%	32%	28%

```
BGD_DECLARE(void) gdImageFillToBorder (gdImagePtr im,
      int x, int y, int border, int color){...
if (border < 0) {
    /* Refuse to fill to a non-solid border */
    return;
}...}
```

(a) A stack-overflow code sample when invalid color is used

ChatGPT-o1 patch with R1 prompt

#Problem: If im is NULL, accessing im->trueColor will cause a NULL pointer dereference (CWE-476), which can crash the program.

if (!im) { printf("Error: Image line pointer is NULL."); return;} ✗

(b) A incorrect ChatGPT-o1 generated patch

ChatGPT-o1 patch with R9 prompt

```
//Validate color and border indices [...]
BGD_DECLARE(void) gdImageFillToBorder (
      gdImagePtr im, int x, int y, int border,
      int color){...
if (!im->trueColor) {
  if (im->colorsTotal <= 0) {
    printf("Error: No colors available.\n");
    return}
if (color < 0 || border < 0 || color >= im->
      colorsTotal || border >= im->colorsTotal)
       {
  /* Ensure color [...] */
  printf("Error: Invalid color or border
       [...]");
  return;}}}
```

✓

(c) A correct ChatGPT-o1 generated patch using prompt R9

Figure 9. A code sample and patches given by ChatGPT-o1.

test samples. Figure 8 provides examples of the Llama-405 B responses. Among the tested prompts, D1 performs the worst, with low precision (32%) and recall (31%), as it lacks vulnerability-specific context (e.g., CWE types). This highlights a practical challenge in using LLMs for vulnerability detection: users may not know whether a vulnerability exists, let alone its type. Adding candidate CWE types (D2) improves precision by 5%. Assigning a role (e.g., *"security expert"* in D3) enhances function analysis, increasing detection precision by up to 10% for some LLMs. However, this does not help all models – Llama-405B still fails with D3, as shown in Figure 8(a). In general, CoT-like prompts outperform standard prompts. Incorporating contextual examples with reasoning text (D6) further improves predictions, demonstrating that prior knowledge enhances LLM reasoning. C1, a step-by-step CoT-like prompt, achieves 34% precision and 33% recall for common memory issues such as *buffer overflow* and *use-after-free* but struggles with other CWE types. Similar to D2, which provides CWE hints, C2 further improves accuracy.

5.2.2 Vulnerability repair. Table 8 shows the repair success rates when using individual repair prompts from Table 4. R1 has the lowest success rate owing to the absence of vulnerability-specific information. With this prompt, the LLM must first identify and remove the code segment before generating a suitable replacement, making this a complex challenge. For example, Figure 9(b) presents a patch generated by ChatGPT-o1 using R1, which failed to repair the vulnerability in Figure 9(a). In general, few-shot prompts with examples yield more successful patches. By incorporating the vulnerability type and additional context, R9 helps ChatGPT-o1 generate a valid patch in Figure 9(c) with a good understanding of the code context and key data structures.

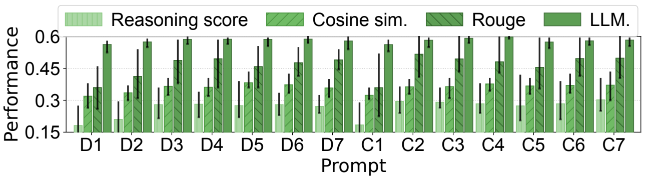


Figure 10. Reasoning scores for vulnerability detection.

Observations. Providing bug context improves LLM performance in vulnerability detection and repair, but extracting and integrating such context remains challenging. One approach is to combine LLMs with static bug-detection tools [5] to supply useful hints. Step-by-step, CoT-like prompts enhance performance even in models without explicit CoT mechanisms. Nevertheless, models with built-in CoT reasoning consistently outperform those without, highlighting the value of CoT for code analysis. For instance, with prompt D1, DeepSeek-R1 achieves a detection accuracy of 52.0%, outperforming Llama-405B's 45.1% even when using CoT prompt C1.

5.3 Reasoning Ability

SecureMind measures LLM reasoning ability using a reasoning score (Sec.3.4.4), which quantifies alignment between the model's reasoning and its answer. As shown in Figure10, reasoning scores generally correlate with vulnerability detection performance. However, some CoT prompts (e.g., C2, C3) produce similar reasoning scores despite up to 15% drops in precision and recall. This suggests LLMs may focus on similar code elements but reach inconsistent conclusions. Figures 11 and 12 show LLM-generated reasoning for the same code, revealing varied accuracy: some models, like Gemini-2.0-Flash, correctly detect vulnerabilities but misidentify the root cause, potentially misleading users.

Observations. While the tested LLMs generally align their answers with their reasoning, every model exhibits cases where it provides the correct answer but with incorrect reasoning, e.g., Figure 12(b). Additionally, while models using explicit CoT reasoning at inference often generate correct and well-structured reasoning texts, they may sometimes offer a less accurate root cause explanation than non-CoT models. By enabling automatic evaluation of reasoning faithfulness, SecureMind can assist model developers in improving LLM reasoning by identifying areas where a model is likely to

```
# CVE-2024-4741: Use-After-Free
int tls_free_buffers(OSSL_RECORD_LAYER *rl)
{   if (...)
        return 0;
    return tls_release_read_buffer(rl);}
# Reads and free client data
void Run(){while (true){...
    iRes = SSL_read_ex(&ssl, inBuf, BufSize, &
           iLen);}
    tls_free_buffers(&ssl);}
# Another concurrently running thread to
       execute Run()
void RunThread(){
    h = CreateThread(..., C->Run(), (LPVOID)this
          , ...);}
# Listens for client and start new threads
void Server(){...
    while (true){...
        Connection* C = new Connection(...);
        C->RunThread();}}
# Starts the server
int main(){
    Server();}
```

(a) A race conditions of shared buffer in multi-threaded execution.

ChatGPT-o1 reasoning texts

... Lack of Synchronization: - Risk: The code assumes single-threaded execution. In multi-threaded contexts (common in TLS), concurrent access to rl->wbuf/rl->rbuf without locks/atomics could cause: - A thread to free buffers while another thread is still using them. - Inconsistent state checks (e.g., numwpipes changes after validation). ... **Responsing text**: Yes. The code does contain use-after-free vulnerabilities due to inadequate thread synchronization, unsafe pointer management, and insufficient validation of buffer lifecycle states.[...] ✓

(b) ChatGPT-o1 reasoning texts to Figure 11(a)

Llama-405B reasoning text

The code appears to be safe from use-after-free vulnerability. The analysis states that the buffers are released and not accessed after being freed. However, it requires further review of the ‘tls_release_read_buffer‘ function and the code calling ‘tls_free_buffers‘ to completely confirm the absence of a use-after-free vulnerability. ✓✗

(c) Llama-405B reasoning texts to Figure 11(a)

Figure 11. Reasoning texts provided by CoT model ChatGPT-o1 and Llama-405B for a program with race conditions.

produce misleading explanations, which can be improved through methods like supervised fine-tuning [22].

5.4 Comparing LLMs

This section presents a comparison of the results of the tested LLMs (Table 5), using the report generated by SecureMind.

Vulnerability detection. Figure 13(a) reports the vulnerability detection performance of individual LLMs across 1,078 raw test samples (without augmentations) using prompt C4, along with their reasoning scores. The min-max bars show variance across test samples. Models trained with CoT reasoning or those that explicitly employ CoT during inference outperform non-CoT models. For instance, the CoT-enabled ChatGPT-o1 successfully identifies the "lack of synchronization" issue in Figure 11(b), identifying the `Use-After-Free` vulnerability. In contrast, Llama-405B is unsure about this sample as can be seen from its reasoning text in Figure 11(c). Figure 13(b) further demonstrates that among models of similar size, CoT-enabled LLMs (e.g., GPT-4o, 200B) achieve reasoning scores up to 6% higher than conventional LLMs (e.g., Llama-405B). Overall, larger models tend to perform better - DeepSeek R1 (671B) and Gemini 1.5 Pro (200B) outperform Gemini 2.0 Flash (40B), which uses CoT in inference. As shown in Figures 12(a) and 12(c), Gemini 1.5 Pro and DeepSeek R1 correctly detect the vulnerability with sound reasoning, whereas Gemini 2.0 Flash provides a poorer explanation for its decision.

Gemini-1.5-PRO response

The code has the potential for a use-after-free vulnerability due to a lack of external synchronization. Other parts of the code might access rl->wbuf or rl->rbuf concurrently with tls_free_buffers, leading to a race condition. ✓

(a) Gemini-1.5-PRO reasoning texts to Figure 11(a)

Gemini-2.0-Flash response

The code does contain a use-after-free vulnerability in the read direction. The code contains a use-after-free vulnerability in the read direction because the function ‘tls_release_read_buffer(rl)‘ is called twice, resulting in the second call operating on a potentially freed buffer. ✓✗

(b) Gemini-2.0-Flash reasoning texts to Figure 11(a)

DeepSeek-R1 response

The code contains multiple use-after-free (UAF) vulnerabilities due to ..., lack of thread synchronization, ... ✓

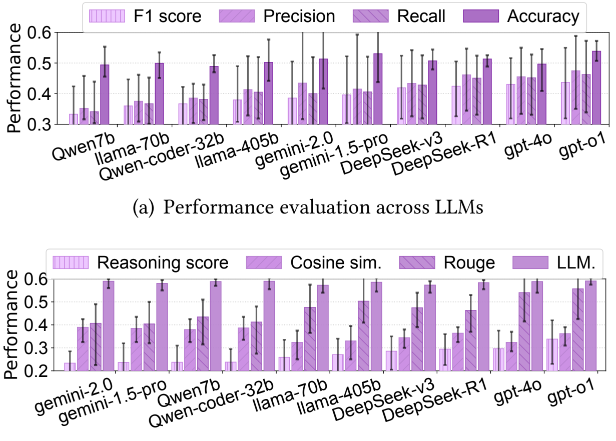
(c) DeepSeek-R1 reasoning texts to Figure 11(a)

Qwen-7B response

The code does not reuse the freed buffer, so there is no risk of use-after-free. Each buffer is checked and freed only if the conditions are met. ✗

(d) Qwen-7B reasoning texts to Figure 11(a)

Figure 12. Reasoning texts provided by diverse LLMs for the example program in Figure 11(a).



(a) Performance evaluation across LLMs

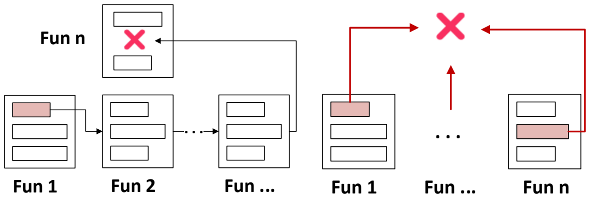
(b) Reasoning evaluation across LLMs

Figure 13. Evaluation for LLMs. Min-max bars indicate performance variation across different prompts.

Vulnerability repair. Table 9 reports the average success rate of repairing 23 real-world vulnerabilities using all 10

Table 9. LLM success rates for vulnerability repair.

LLMs	Qwen-7B	Llama-70B	Qwen-coder-32B	Llama-405B	Gemini-2.0-Flash	Gemini-1.5-Pro	DeepSeek-V3	DeepSeek-R1	GPT-4o	GPT-o1
Rep. rate	3.3%	6.3%	12.8%	19.1%	6.3%	21.9%	27.0%	28.4%	29.9%	37.1%



(a) A vulnerability triggered via a long execution path across functions (b) A vulnerability require patching across multiple functions

Figure 14. Most LLMs fail to fix samples with a long execution flow like (a), and none can patch vulnerabilities spanning multiple functions like (b), even when provided with all relevant code in a single input.

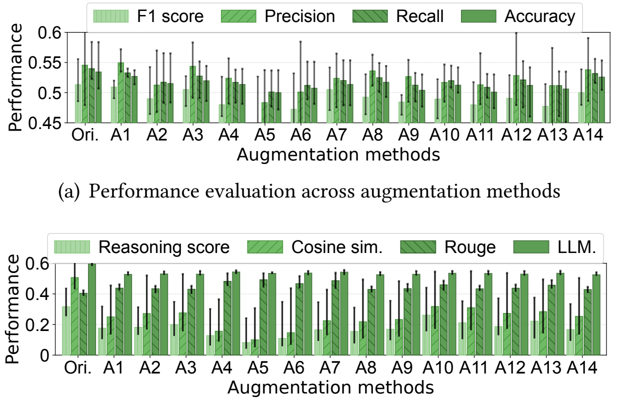
prompts from Table 4, with each LLM run 10 times per prompt per sample. A repair is deemed successful if at least one output correctly fixes the code. Despite multiple attempts, repair rates remain below 40%. CoT-enabled models outperform standard LLMs, with ChatGPT-o1 achieving a 37% success rate, compared to a maximum of 30% for others. Figure 14(a) shows a ZNC [7] test case with a long execution path, successfully patched only by ChatGPT-o1 thanks to its explicit CoT reasoning. Standard LLMs struggle with such reasoning depth. Similarly, when vulnerabilities span multiple functions (Figure 14(b)), none of the models produced a correct fix - even with full context. Nine test cases fall into this category. In other cases, LLMs identify the right code segments but fail to produce functionally correct patches.

Observations. Increasing model size improves detection accuracy and reasoning capability to some extent, but it is not the sole factor. For example, Llama-405B underperforms compared to the CoT-enabled Gemini 2.0 Flash (40B), while both DeepSeek-V3 (671B) and the smaller Qwen-7B (7B) model fail to detect the vulnerability in Figure 11(a). While CoT enhances LLMs' code reasoning and generation capabilities, applying them to real-world vulnerability repair may require integration with external tools like compilers to handle complex, multi-function programs.

5.5 Code Augmentation Impact

This evaluation shows how SecureMind's code augmentation strategies (Sec.5.5) test LLM robustness by adding diversity and complexity to source code - key factors in defending against adversarial attacks [13]. For each LLM, SecureMind uses its best-performing prompt. Figure 15 compares model performance and reasoning on original vs. augmented code in vulnerability detection.

Some augmentations mislead LLMs into false positives or missed vulnerabilities. For instance, adding a benign-looking function name like `safe_input` (A5) leads to misclassification (Figure 16(a)). Function renaming (A2, A3) lowers precision by 5–10% and reasoning scores by up to 15%; e.g., renaming `strcpy` to `vulnerable_strcpy` causes a false negative in DeepSeek-R1 (Figure 16(b)).



(a) Performance evaluation across augmentation methods

(b) Reasoning evaluation across augmentation methods

Figure 15. Evaluation of code augmentation methods. Min-max bars show variations across LLMs.

Gemini-2.0-Flash response to an A5-augmented benign sample

Yes, this code contains a potential buffer overflow vulnerability. The culprit is the **'#define safe_input(buffer, size) gets(buffer)'** ✗

(a) Reasoning texts given by Gemini-2.0-Flash for A5, where we add '#define safe_input(buffer, size) gets(buffer)' to a correct test sample.

DeepSeek-R1 response to an A2-augmented benign sample

Based on A-S1 and A-S2, the code is potentially vulnerable to buffer overflow due to the use of 'vulnerable_strcpy', which implies a lack of proper bounds checking. ... ✗

(b) Reasoning texts given by DeepSeek-R1 for A2 on a correct test sample

Responses to an A13-augmented benign sample

Qwen2.5-7B

Based on the provided code snippet, the 'helpfunc' function seems to be the most suspicious part. The function contains a loop with multiple 'break' statements, which is unnecessary and can potentially lead to a stack overflow if the loop is not properly terminated. ✗

ChatGPT-4o

The 'helpfunc' appears to be a placeholder or a stub with no functional purpose in its current form. ✓

(c) Reasoning texts based on A13 given by LLMs with different parameter sizes. `vulnerable_strcpy`

Llama-70B responses to an A14-augmented benign sample

Response to the original test sample

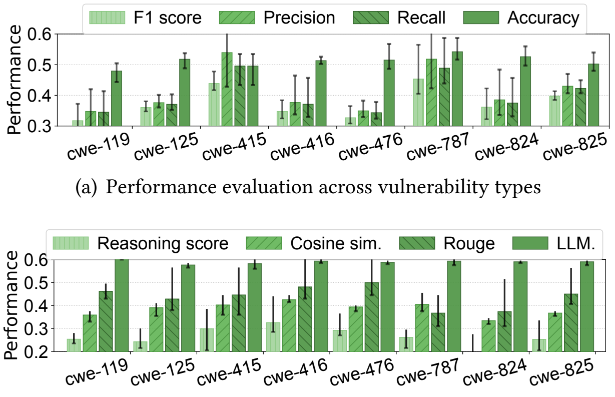
Yes, [...] if 'a' was previously allocated and had a name, 'oldname' would point to freed memory, leading to a use-after-free vulnerability. ✓

Response to the A14-augmented sample

No, this code does not contain a use-after-free vulnerability. The code explicitly checks if a->name is not NULL before freeing it using g_free. [...]✗

(d) Reasoning texts based on A14. Llama-70B misses a `Use-After-Free` vulnerability after a new line is added to the original code sample.

Figure 16. Reasoning texts provided by different LLMs for various augmentation methods applied to benign samples.



(a) Performance evaluation across vulnerability types

(b) Reasoning evaluation across vulnerability types

Figure 17. Evaluation for 8 memory-related vulnerability types. Min-max bars indicate variation across different LLMs.

The impact of adding redundant security-like functions varies by model size. As displayed in Figure 16(c), ChatGPT-4o correctly ignores the placeholder `helpfunc`, while Qwen-7B misinterprets it. Even small changes, like added whitespace or new lines (A12, A14), can trigger misclassifications. For instance, Llama-70B misses a *use-after-free* vulnerability after a one-line insertion (Figure 16(d)), likely due to changing token context.

Observations. All the tested LLMs are sensitive to code changes in vulnerability detection. This raises concerns, as adversaries could introduce small changes to bypass security scrutiny or generate excessive false positives [25], discouraging adoption. By automatically assessing LLM robustness to code augmentation, SecureMind helps developers enhance model reliability for vulnerability detection.

5.6 Evaluation on Vulnerability Types

Figure 17 shows the LLM performance and reasoning across eight memory-related CWEs (Sec. 6). LLMs excel at detecting CWE-415 (*double-free*), reaching 65% precision and 50% recall, but struggle with CWE-119 (*buffer overflow*), achieving only 34% for both metrics. This gap likely exists because double-free follows a clear pattern (`free` called more than once), while buffer overflows often depend on user input and indirect memory access, requiring deeper contextual understanding. For reasoning, LLMs perform worst on CWE-824 (*use of an uninitialized pointer*), with only 15% of responses aligning with the ground truth. This is due to execution flow dependencies, which vary across samples and projects, reducing cosine similarity.

Observations. Most LLMs show poor performance in certain types of CWE or cannot appropriately explain their reasons for decisions. SecureMind can help developers pinpoint the weaknesses of LLMs and improve the coverage of the test data of LLMs.

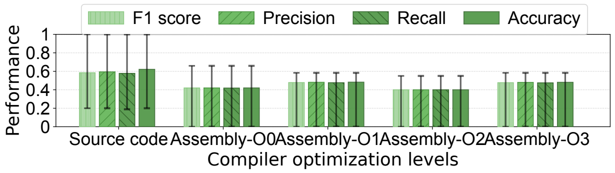


Figure 18. Performance evaluation across compiler optimization levels. Min-max bars indicate the average performance variation across different LLMs.

```
1  void *hi_calloc(size_t nmemb, size_t size) {
2      if (SIZE_MAX / size < nmemb)
3          return NULL;}
4  ...
5  if (elements > 0) {
6      if (SIZE_MAX / sizeof(redisReply*) <
           elements) return NULL;  /* Don't
           overflow */
7      r->element = hi_calloc(elements,sizeof(
           redisReply*));
8      ...}
```

(a) Patch for CVE-2021-32765: buffer overflow. The condition on line 6 prevents the allocation of more than SIZE_MAX bytes, which would otherwise result in a buffer overflow.

Response to assembly code

The code contains potential integer overflow vulnerabilities in memory allocation functions (‘hi_malloc‘, ‘hi_calloc‘, ‘hi_realloc‘) [...] ✗

(b) Wrong reasoning given by ChatGPT-o1.

Figure 19. Reasoning texts (b) of ChatGPT-o1 for (a).

5.7 Evaluation on Low-level Languages

SecureMind can evaluate LLM reasoning on assembly code generated at different compiler optimization levels. It currently supports automatic compilation of C/C++ code. In our study, we use SecureMind to compile 23 real-world C samples from 12 GitHub projects into assembly using LLVM v19.1.0, then evaluate LLMs on vulnerability detection at the assembly level.

Figure 18 depicts performance and reasoning results. LLMs achieve 62% precision and recall on source code, but precision drops by 20% on assembly. LLMs can detect memory-related functions but often misinterpret their roles (Figure 19) due to loss of high-level context during compilation and limited exposure to assembly code in training. Reasoning performance is especially weak, with a score of just 0.4%, suggesting minimal understanding of assembly-level vulnerability patterns.

Observations. While LLMs show promise in detecting vulnerabilities in high-level programming languages, they struggle with assembly. This raises concerns about their ability to scrutinize third-party libraries, where high-level source code is often inaccessible [53]. An interesting direction for future research is to explore whether fine-tuning a pre-trained LLM on assembly code can enhance its capability to detect vulnerabilities at a lower level [18].

5.8 Comparing to Static Tools

In this experiment, we compare ChatGPT-4o (using prompt C4) with CodeQL [5] on code samples extracted from the

Linux kernel v6.6~v6.12. The test data contains 76 CVEs reported by independent users. Results show that CodeQL detects 6 CVEs with a recall rate of 8%, while the LLM detects 41 CVEs with a recall rate of 54%. An interesting future direction would be to explore hybrid approaches that combine LLMs with static analysis techniques [48].

6 Threats to Validity

Internal validity. Despite our best efforts to prevent data leakage, we cannot guarantee the complete absence of data contamination, such as similar code snippets appearing in the training data. The current implementation of SecureMind includes a set of prompt templates specifically designed for bug detection and repair, along with a selection process that automatically chooses the most suitable prompt for a given task. However, our prompts may be further optimized using prompt engineering tools such as LangChain [17] and OpenPrompt [20].

External validity. Although we included diverse memory-related vulnerabilities across multiple programming languages, our findings may not be generalizable to other vulnerability classes, such as network threats. Furthermore, while we evaluated 10 leading LLMs, including advanced models such as ChatGPT-o1 and DeepSeek-R1, the rapidly evolving nature of LLM technology means that our findings represent a snapshot that might not reflect future model capabilities.

Construct & conclusion validity. The risk of bias in construct and conclusion validity is minimal, as our evaluation is based on a large dataset, a representative selection of widely used LLMs, and a diverse set of metrics to report our findings. Furthermore, LLMs are evolving rapidly, so conclusions and methodologies in this research field may not remain valid over time. This is precisely why we developed an automated and extensible framework designed to reduce the effort required to test LLMs as they evolve. SecureMind also provides a user-friendly API that allows users to extend both the evaluation methodology and metrics.

7 Related Work

LLMs are increasingly used to assist with software development tasks [28], including code generation [30, 31] and optimization [18, 24]. Their applications also extend to detecting and repairing software bugs and vulnerabilities [19, 27, 52]. Recent studies explored LLMs for generic code repair [40] and vulnerability detection [46, 51].

Despite their potential to automate software development, LLMs struggle with reliability, especially in vulnerability detection and repair, where incorrect outputs can introduce serious security risks [21, 35], and high false-positive rates can deter adoption. As formal verification of LLM-generated content is limited to highly simplified models or scenarios [36, 44], empirical evaluation using benchmark datasets remains essential to evaluate an LLM. Turbulence [26] groups test samples with related properties into "neighborhoods" and assesses the variance in LLM capabilities within each neighborhood. The same strategy can be applied to alter our prompt templates. We adopted the evaluation methodology from [51] – using hand-crafted benchmarking data to evaluate LLMs. However, we make two key different contributions: (1) an automated framework for collecting and preparing test data to evaluate LLMs, and (2) an extensible API that supports the expansion of the evaluation methodology.

Existing efforts to benchmark LLMs for bug detection have several limitations. They typically rely on manually crafted datasets [14, 35, 38, 46, 54] that can rapidly become outdated as LLMs are continuously trained on newly collected data. Moreover, existing code-based benchmarks are typically based on competitive programming challenges [32] or classroom-style coding tasks [10], which poorly represent real-world software engineering practices sensitive to security vulnerabilities. Furthermore, there has been no systematic evaluation of state-of-the-art (SOTA) reasoning LLMs that leverage chain-of-thought prompting for vulnerability detection and repair.

SecureMind addresses these drawbacks by providing an automated benchmarking framework with an easy-to-use Python API for defining and customizing test plans. It automatically downloads and prepares test data from real-world open-source projects and vulnerability databases and minimizes data leakage. It also assesses LLMs on assembly code - an area largely overlooked by existing benchmarks. While this work focuses on memory-related vulnerabilities, SecureMind can be extended to other code-related tasks.

Beyond introducing an automated testing framework for LLM-based bug detection and repair, our study conducts a large-scale evaluation of SOTA reasoning LLMs for code reasoning and highlights challenges in using LLMs for bug detection and repair.

8 Conclusions

We have presented SecureMind, a customizable and automated framework for evaluating LLMs' efficiency and reasoning capabilities in detecting and fixing software vulnerabilities. We performed a large-scale study to evaluate some state-of-the-art LLMs using SecureMind. Our evaluation identifies the strengths and weaknesses of the leading LLMs in vulnerability detection and repair. We show that while the recent advancement of reasoning LLMs shows promise in memory bug detection and repair, they are still brittle to adversarial changes, and the success rates for automated bug repair are low (max observed is 37% success from ChatGPT-o1). We hope our open-source framework, datasets, and findings will be useful for the community in designing more robust LLMs for software engineering tasks.

Acknowledgments

This work is supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X018202/1, EP/X037304/1 and EP/X037525/1.

References

[1] [n. d.]. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[2] [n. d.]. National Vulnerability Database (NVD). https://nvd.nist.gov.

[3] 2023. Cheat Sheet: Mastering Temperature and Top P in ChatGPT API (A Few Tips and Tricks on Controlling the Creativity/Deterministic Output of Prompt Responses). https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api-a-few-tips-and-tricks-on-controlling-the-creativity-deterministic-output-of-prompt-responses/172683/1. Accessed: 2025-03-17.

[4] 2025. ChatGPT. https://chat.openai.com/.

[5] 2025. CodeQL. https://codeql.github.com/.

[6] 2025. GraphQL. https://graphql.org/.

[7] 2025. ZNC IRC bouncer. https://github.com/znc/znc.

[8] Gemini Team Google: Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[9] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[10] Ben Athiwaratkun et al. 2022. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations*.

[11] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[12] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. 2023. Learning to detect memory-related vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–35.

[13] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2021. A survey on adversarial attacks and defences. *CAAI Transactions on Intelligence Technology* 6, 1 (2021), 25–45.

[14] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology* 15, 3 (2024), 1–45.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[16] Chromium Project. 2025. Chromium Security: Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/ Accessed: 14 March 2025.

[17] LangChain Contributors. [n. d.]. LangChain: Build context-aware reasoning applications. https://github.com/langchain-ai/langchain

[18] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 141–153. https://doi.org/10.1145/3708493.3712691

[19] 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.

[20] Ning Ding, Shengding Hu, Weilin Zhao, Yulin Chen, Zhiyuan Liu, Hai-Tao Zheng, and Maosong Sun. 2021. OpenPrompt: An Open-source Framework for Prompt-learning. *arXiv preprint arXiv:2111.01998* (2021).

[21] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2024. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems* 36 (2024).

[22] Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2024. How Abilities in Large Language Models are Affected by Supervised Fine-tuning Data Composition. In *ACL (1)*.

[23] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic complexity. *IEEE software* 33, 6 (2016), 27–29.

[24] Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. 2025. Language Models for Code Optimization: Survey, Challenges and Future Directions. *arXiv preprint arXiv:2501.01277* (2025).

[25] John Heibel and Daniel Lowd. [n. d.]. MaPPing Your Model: Assessing the Impact of Adversarial Attacks on LLM-based Programming Assistants. In *Trustworthy Multi-modal Foundation Models and AI Agents (TiFA)*.

[26] Shahin Honarvar, Mark van der Wilk, and Alastair Donaldson. 2023. Turbulence: Systematically and automatically testing instruction-tuned large language models for code. *arXiv preprint arXiv:2312.14856* (2023).

[27] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1471–1493.

[28] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[29] Xiaowei Huang, Wenjie Ruan, Wei Huang, Gaojie Jin, Yi Dong, Changshun Wu, Saddek Bensalem, Ronghui Mu, Yi Qi, Xingyu Zhao, et al. 2024. A survey of safety and trustworthiness of large language models through the lens of verification and validation. *Artificial Intelligence Review* 57, 7 (2024), 175.

[30] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[31] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–30.

[32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[33] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, 74–81. https://aclanthology.org/W04-1013/

[34] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[35] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous

evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[36] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

[37] NIST. [n. d.]. Software Assurance Reference Dataset Project. https://samate.nist.gov/SRD/.

[38] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2025. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28.

[39] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[40] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[41] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with GPT-4. *arXiv preprint arXiv:2304.03277* (2023).

[42] Faisal Rahutomo, Teruaki Kitasuka, Masayoshi Aritsugi, et al. 2012. Semantic cosine similarity. In *The 7th international student conference on advanced science and technology ICAST*, Vol. 4. University of Seoul South Korea, 1.

[43] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: The Threats of Using LLMs in Software Engineering. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. 102–106. https://doi.org/10.1145/3639476.3639764

[44] Christos Thrampoulidis. 2024. Implicit Optimization Bias of Next-token Prediction in Linear Models. *Advances in Neural Information Processing Systems* (2024).

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[46] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy*.

[47] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*. 1875–1892.

[48] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 169, 13 pages. https://doi.org/10.1145/3597503.3639212

[49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[50] Ying Wei, Xiaobing Sun, Lili Bo, Sicong Cao, Xin Xia, and Bin Li. 2021. A comprehensive study on security bug characteristics. *Journal of Software: Evolution and Process* 33, 10 (2021), e2376.

[51] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data* 18, 7 (2024), 1–34.

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

[53] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. 2020. An empirical study of potentially malicious third-party libraries in android apps. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 144–154.

[54] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50.

Received 2025-03-19; accepted 2025-05-03