SecVulEval: Context-Aware Benchmarking of LLMs for Vulnerability Detection

Anonymous Author(s)

Abstract

Large Language Models (LLMs) show promise for vulnerability detection, but their evaluation is limited by the lack of high-quality benchmarks. Most existing datasets rely on coarse function-level labels, overlook fine-grained vulnerability patterns, and lack critical program context such as data/control dependencies. They also suffer from data quality issues, including mislabeling and duplication, leading to unreliable evaluation and limited real-world relevance.

To address these limitations, this paper introduces SecVulEval, a context-aware benchmark designed to evaluate LLMs on vulnerability detection with rich contextual information. SecVulEval focuses on real-world C/C++ vulnerabilities at the statement level. This granularity enables more precise evaluation of a model's ability to localize and understand vulnerabilities, beyond simple binary classification at the function level. By incorporating rich contextual information, SecVulEval sets a new standard for benchmarking vulnerability detection in realistic software development scenarios. This benchmark includes 25,440 function samples covering 5,867 unique CVEs in C/C++ projects from 1999 to 2024. We evaluated state-of-the-art LLMs in both standalone and multi-agent settings. Results on our dataset indicate that current models remain far from accurately identifying vulnerable statements within a given function, although agent-based approaches provide modest but promising improvements. The best-performing Claude-3.7-Sonnet-driven agent achieves an F1-score of 23.83% for vulnerable statement detection. We believe this benchmark can serve as a foundation for advancing context-aware vulnerability detection with LLMs.

ACM Reference Format:

Anonymous Author(s). 2026. SecVulEval: Context-Aware Benchmarking of LLMs for Vulnerability Detection. In . ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Security vulnerabilities pose a major threat to the safety and robustness of software systems. Much of today's technological infrastructure relies on C/C++ projects, making vulnerabilities in these systems particularly impactful, with consequences that can propagate to many downstream applications [8, 19]. Despite their importance, existing C/C++ vulnerability detection tools often struggle with real-world complexity, diverse codebases, and evolving security threats [23]. The rapid adoption of large language models (LLMs) in software engineering has created new opportunities for automating critical tasks [10, 27, 28]. However, while LLMs have shown strong capabilities in code-related tasks, their effectiveness in detecting real-world C/C++ security vulnerabilities at scale remains largely underexplored.

As more and more LLMs emerge, a reliable benchmark is crucial to evaluating LLMs' capability to detect security vulnerabilities in C/C++ projects. Recently, many benchmarks have been proposed for C/C++, such as BigVul [6], CVEFixes [1], DiverseVul [3], MegaVul [20], PrimeVul [5], etc. Although promising, the existing benchmarks suffer from a few major limitations. First, they lack essential features such as statement-level vulnerability localization, which poses a significant challenge for tasks that require fine-grained analysis, training, or evaluation. Second, some datasets omit crucial details, like bug-fix code pairs, vulnerability types (CWE), and precise CVE metadata. The absence of this information limits researchers' and developers' ability to conduct in-depth investigations or build effective repair tools, ultimately hindering advancements in the field. Third, existing datasets frequently include only the vulnerable functions, omitting the broader program context that is essential for accurately identifying and understanding security flaws. This missing context encompasses critical aspects such as data and control dependencies, interprocedural interactions, and environment constraints, all of which play a key role in determining whether a piece of code is truly vulnerable and how the vulnerability manifests. Finally, although some datasets offer line-level labels (e.g., BigVul [6]), simply having added-deleted lines from a commit may not be very useful. Specifically, for C/C++, some statements can be very long, and it is common practice to break them down into several lines[1]. Therefore, it is hard to make a meaningful understanding of only the line fragments without seeing the entire statements.

Most of the current vulnerability detection techniques conduct vulnerability detection at a local scope, often focusing on a given function in isolation [7, 13, 22]. These approaches frequently overlook critical contextual information from related codebases, such as variable state returned from an external function, function arguments, execution environment, etc. A recent study [21] has shown through empirical evaluation that most vulnerabilities in C/C++ require some level of external context to be correctly identified, such as variables, functions, type definitions, and environmental constraints that affect the function. As a result, neglecting the contextual information of a code snippet hinders these techniques from accurately assessing the presence of vulnerabilities within the code. Their study further reveals that many of the machine learning techniques that report high scores in vulnerability detection may be learning spurious features instead of the true vulnerability. This

Conference'17, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://github.com/bminor/glibc/commit/2864e76#diff-70c1d4052e9de7ad76e1437a17f00b70a9ea6732a3667041ccfb8dbb0caccae4

underscores the need for a more granular identification of vulnerabilities, along with correct reasoning to determine whether the models can truly spot vulnerabilities.

In this work, we address these limitations by introducing a context-aware C/C++ vulnerability dataset that provides granular information up to the statement level. We focus on vulnerabilities that have been patched, ensuring that the dataset reflects real-world fixes. For each vulnerability, we gather detailed metadata, including CWE types, corresponding CVE (Common Vulnerabilities and Exposures) IDs and descriptions, commit IDs, commit descriptions, changed files, changed functions, and the modified statements that were deleted or added before and after the patch. We also extracted the contexts related to the vulnerable functions using GPT-4.1 and added them to the dataset.

We adopt the five levels of context defined by Risse et al. [21] to represent the essential context for a vulnerability: **Function Arguments**, **External Functions** (functions called from within the target function), **Type Execution Declarations** (e.g., struct, enum, and other type definitions), **Globals** (such as global variables and macros), and **Execution Environments** (e.g., the presence of a specific file in a given path). Our manual analysis of a subset of 1k (around 10% of the whole dataset) samples shows that GPT-4.1 can identify the contexts necessary for a given vulnerability in a function with 83.16% accuracy. The statement-level granularity and contextual information, along with other metadata, enable deeper analysis and a more accurate evaluation of vulnerability detection.

We further evaluate five LLMs, including both open-source models such as *Qwen2.5-Coder-32B*, *Deepseek-Coder-33B*, *Codestral-22B* and proprietary models like *GPT-4.1* and *Claude-3.7-Sonnet* on our dataset to show their ability to detect C/C++ vulnerabilities at statement level. Note that, in our experiments, we assess LLMs both in their standalone form and as components of a multi-agent pipeline, in which each agent, i.e., powered by an LLM, collaborates to detect vulnerable statements. This design is motivated by prior work showing that decomposing complex tasks into smaller, actionable components can enhance LLM performance [18, 24, 25]. Our initial experiments show that state-of-the-art LLMs are still far from ready for practical use as vulnerability-detection tools for C/C++. Standalone LLMs achieve detection rates below 4%, while the agent-based setting improves performance to approximately 23.83%. Our results show the promise of an agentic approach to vulnerability detection, while urgently underscoring the need to develop stronger semantic models and deeper domain understanding to improve performance. The primary contribution of this work is as follows.

- **A new statement-level C/C++ vulnerability benchmark**: We introduce a large-scale C/C++ vulnerability dataset with statement-level vulnerability annotations, paired vulnerable/fixed code, and rich CVE–CWE metadata, addressing key limitations of existing benchmarks.
- **Context-aware vulnerability representation**: We augment each vulnerable function with multi-level contextual information (e.g., external functions, globals, types, execution environment), enabling more realistic and semantically grounded vulnerability analysis.
- **Comprehensive evaluation of LLMs and agentic pipelines**: We evaluate state-of-the-art LLMs in both standalone and multi-agent settings, showing that current models perform poorly at statement-level vulnerability detection, while agent-based approaches offer promising but still limited improvements.

Replication Package and the Dataset: https://anonymous.4open.science/r/SecVulEval-C02C

Table 1: Comparison of SecVulEval to widely used C/C++ vulnerability datasets from key aspects, i.e., the number of vulnerable functions, the availability of metadata, the duplication rate, the availability of context, and the detection level. Duplicate rates marked with * are reported from [5].

	#Vul Funcs	Metadata	Duplicate (%)	Context Info	Statement-Level
DiverseVul [3]	18,945	✗	3.3*	✗	✗
ReVeal [2]	1,658	✗	1.54	✗	✗
Devign [29]	12,457	✗	0.26	✗	✗
CVEFixes [1]	5,365	✓	18.9*	✗	✗
BigVul [6]	3,754	✓	12.7*	✗	✗
SVEN [11]	417	✓	0.44	✗	✗
PrimeVul [5]	6,968	✓	0.0	✗	✗
MegaVul [20]	8,254	✓	0.0	✗	✗
SecVulEval	10,998	✓	0.0	✓	✓

2 Related Work on Vulnerability Datasets for C/C++

Zhou et al. [29] provide the *Devign* dataset, collected to evaluate their Devign detection model. This dataset includes over 12,457 C/C++ vulnerabilities. However, it does not include other metadata, such as CWE and CVE. Additionally, they collect vulnerable functions using a simple commit search with string matching, which results in the inclusion of many inaccurate functions, i.e., up to 20% according to a manual analysis conducted by [21] on a random subset. The *ReVeal* dataset proposed by [2] includes 1,658 C/C++ vulnerable functions, but only from the Chromium and Debian projects. Chen et al. [3] proposed *DiverseVul*, a C/C++ vulnerability detection benchmark with 18,945 vulnerabilities (from 797 projects and covering 150 CWEs). They showed that code-specialized models, e.g., *CodeT5* and *NatGen*, surpass graph-based methods but face persistent issues such as high false positives, poor generalization, and limited data scalability, highlighting the need for improved deep learning approaches. Ding et al. [5] introduced *PrimeVul*, another C/C++ vulnerability benchmark with rigorous de-duplication, chronological splits, and VD-S metrics, exposing code models' near-random failure despite prior overestimation and underscoring the urgency for innovative detection paradigms. However, all these datasets only include vulnerability annotations at the function level, i.e., whether the function is vulnerable or not. They lack vulnerability information at a more granular level, like the statement level. Statement-level labels are necessary to understand how the vulnerability is caused, which can then be utilized for better training and evaluation of vulnerability detection models.

Fan et al. [6] proposed *BigVul*, a C/C++ vulnerability dataset derived from open-source GitHub projects containing 3,754 vulnerabilities from 348 projects to support vulnerability detection. This dataset is closer to our work, as it includes line-level labels for vulnerable functions. The dataset, spanning 348 projects and 91 vulnerability types, is publicly available and uniquely integrates code-centric fixes with CVE metadata for enhanced accuracy and

usability. Bhandari et al. [1] collected a large collection of 5,365 C/C++ vulnerabilities from NVD. It includes vulnerabilities at five levels of abstraction, including line-level vulnerability labels and other metadata. However, both datasets heavily suffer from high duplication rates, which risks data leakage in the testing or evaluation of detection models. The *SVEN* dataset proposed by He et al. [11] also includes line-level vulnerability information and has accurate labeling as the entire dataset is manually annotated. But, due to this manual process, the dataset only includes 417 vulnerable C/C++ functions, limiting its use cases. Moreover, these three datasets include line-level labels, which may not be useful in many cases. C/C++ is a verbose language, and it is common for a statement to span multiple lines. Therefore, a single line might only be a fragment of a statement, and therefore, by itself, does not carry meaningful information. Wu et al. [26] proposed a Java vulnerability benchmark, *VJBench*, and its transformed version, *VJBench-trans*, to evaluate and compare the effectiveness of LLMs and deep learning-based automated program repair (APR) techniques in fixing security vulnerabilities. It supports innovations such as expanding vulnerability-specific training data, fine-tuning LLMs with APR data, and leveraging code transformations to enhance APR capabilities, revealing significant limitations in current models while highlighting Codex's relative robustness. Henriques et al. [12] proposed an automated pipeline to dynamically mine and update software vulnerability data from diverse open-source projects, expanding an existing dataset's domain while introducing a dashboard for real-time analysis. Lee et al. [17] employed a multi-agent scaffold that automatically constructs code repositories with harnesses, reproduces vulnerabilities in isolated environments, and generates gold patches for reliable evaluation. While existing benchmarks have advanced vulnerability research, critical gaps remain. *DiverseVul* and *PrimeVul* lack statement-level vulnerability localization and precise CVE metadata, limiting their utility for fine-grained analysis or repair tasks. *DiverseVul*'s lack of code pairs (bug-fix) further constrains its usage. *Big_Vul* includes line-level labels but suffers from high duplicate rates (12.7%), low labeling accuracy, and an outdated period (up to 2019). *PrimeVul*'s outdated CWE labels (e.g., CVE-2011-2707 misclassified) further reduce reliability.

Our benchmark addresses these limitations by providing statement-level granularity through paired vulnerable and fixed code versions, rigorous de-duplication and filtering, and up-to-date CVE–CWE metadata. Specifically, we include both vulnerable and non-vulnerable functions annotated with statement-level vulnerability labels, enriched with relevant contextual information to support precise analysis. Additional metadata is provided to enable diverse evaluation settings, while strict quality control procedures ensure high data fidelity. Table 1 presents a detailed comparison between our benchmark and prior work.

3 Benchmark Construction

In this section, we provide a detailed overview of the different steps and stages in building our benchmark data, i.e., vulnerability collection, commit data collection, noisy data filtering, and contextual information collection. The workflow is illustrated in Figure 1.

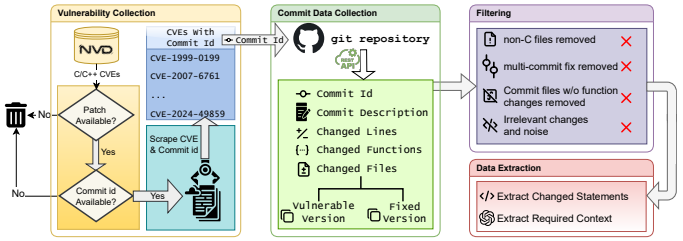

Figure 1: Data collection pipeline of SecVulEval

3.1 Vulnerability Collection

We start by collecting CVEs recorded in the National Vulnerability Database (NVD)[2]. NVD has a rich collection of vulnerabilities and is regularly updated, making it a standard vulnerability repository. NVD provides detailed metadata for each CVE entry, including descriptions, severity scores (e.g., CVSS), affected products, and references to patches or advisories. However, one limitation of the NVD is that it does not explicitly categorize vulnerabilities by programming language. To focus specifically on C/C++ vulnerabilities, we leverage the project names identified as C/C++ projects in prior vulnerability datasets, such as BigVul [6], CVEFixes [1], and PrimeVul [5], as listed in Table 1. By reusing these curated project names, we ensure that the collected CVEs are associated with C/C++ codebases. We further retrieve CVE records for these projects (called 'product' in NVD) where the CVE status is not *REJECTED*. We also utilize the keyword search feature of the NVD API to search with related keywords ('C++', 'C language', '.cpp', etc.). These results are then filtered by the file types, and only C/C++ vulnerabilities are kept. To enable the study of actual vulnerabilities, only CVEs with patch-related information are retained. Using the "Patch" tag from the NVD Developers API, CVEs without patch references are discarded. Additionally, to avoid duplication, we only kept the CVEs that had at least one link to a patch commit in their references. However, some of the commit links point to many forked repositories. We discarded such forked commits and only kept commits to the original repo. We ended up with a collection of CVEs, each with a description, associated CWE, fixing commit ID, and other metadata.

3.2 Commit Data Collection

Given the commit IDs collected for each CVE, the next step is to collect the commit-related information. We utilize the GitHub REST API to fetch commit details. For each commit, we collect its commit ID, commit message, and the files changed in the commit. We also sanitize the commit descriptions by removing accreditation lines (such as reporter emails, cc emails, etc.) since they do not contain any information related to vulnerable code changes. For each changed file, we extract the changed lines (i.e., added lines and deleted lines) and changed functions, i.e., functions containing the changed lines. For all changed files, lines, and functions, we include two copies: one *before* the fixing commit (i.e., vulnerable version) and one *after* the fixing commit (i.e., fixed version).

[2]https://nvd.nist.gov

Table 2: Top 10 projects in SecVulEval.

Projects	CWEs	CVEs	Vul. Funcs	Non-vul. Funcs
Linux	69	1,920	2,793	3,288
Chrome	35	526	1,748	1,940
TensorFlow	34	355	547	959
Android	23	209	551	749
ImageMagick	27	172	208	354
vim	21	162	182	290
gpac	22	138	151	215
tcpdump	8	109	145	206
radare2	16	103	168	214
FFmpeg	20	99	110	152
Total (707 Projects)	145	5,867	10,998	14,442

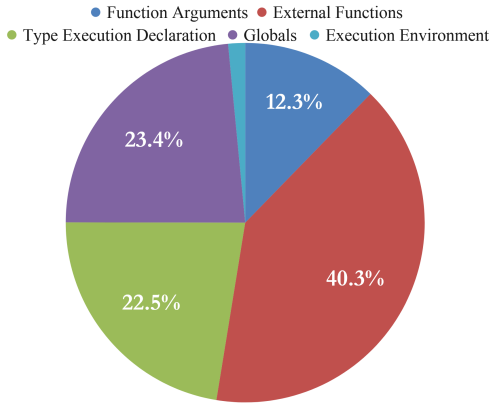

Figure 2: Distribution of five context categories in SecVulEval.

3.3 Filtering & De-noising

After collecting the commit artifacts, we apply a series of filtering and denoising steps to enhance the quality and reliability of our dataset. The filtering criteria are ① removing non-C files, ② removing multi-commit fixes, ③ removing commits with no functions changed, and ④ using heuristics to remove refactoring, reformatting, etc. Below, we describe them in detail.

① We exclude non-C/C++ files (e.g., *.S*, *.rst*, config files) as their changes are often in documentation, macros, or signatures, and are side effects unrelated to the vulnerability. ② We retain only CVEs with single-commit fixes for simplicity, as multi-commit data can be challenging to present effectively to the model. Our manual investigation revealed that, in many cases, multi-commit fixes primarily involve similar changes across multiple files or refactoring of related functions in other files. To maintain clarity and consistency, we discard only 43 CVEs with multiple commits. ③ We exclude commit files that do not involve any changes to functions. Some commits may only modify function prototypes, add comments, or update enum values or struct fields[3]. While these changes may be related to the codebase, they provide minimal insight into the actual vulnerability and instead introduce unnecessary noise. ④ Finally, we use heuristics to filter out tangled files in changes and improve the labeling accuracy. Specifically, when a commit changes several functions in a file, all the changes are not necessarily related to the vulnerable code. In fact, in many cases, variable/function renaming or function signature updating is carried out across the whole file, resulting in multiple functions being updated. We filter out tangled changes by retaining only functions (i) solely modified in the file, or (ii) explicitly referenced in the CVE or commit message, avoiding unrelated edits like renaming or formatting.

[3]https://github.com/torvalds/linux/commit/4e8771a3, https://github.com/torvalds/linux/commit/c06cfb0

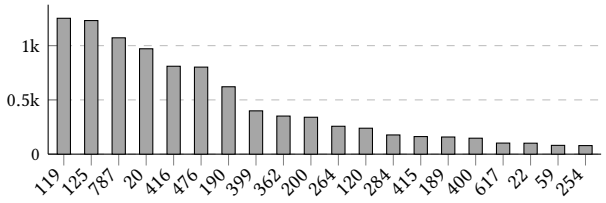

Figure 3: Number of vulnerable functions in Top-20 CWE types in SecVulEval. The x-axis represents the list of CWE IDs, while the y-axis indicates the number of corresponding samples.

The final step in the filtering process eliminates any duplicate functions. Duplicates can arise for several reasons in the initial collection process, such as multiple copies of the same function in different files, or a CVE being assigned to multiple CWE types, etc. Duplicate entries are a big problem in vulnerability benchmarks as they can leak data to training and also make the benchmark biased towards excessive duplicates. Previous benchmarks such as DiverseVul [3], BigVul [6], and CVEFixes [1] suffer from 3.3%-18.9% function duplication problem as shown in [5]. Our filtering process eliminates duplicate functions by mapping each function to an *md5* hash as done by [5]. We normalize each function by stripping away all leading/trailing whitespaces, '\n', and '\t'. Then we convert the function string to an *md5* hash and keep only one copy of a function in the case of a collision. In this way, we ensure that all functions in our dataset are unique, eliminating the data leakage problem.

After filtering, we end up with a collection of 5,867 unique C/C++ vulnerabilities (CVEs) from 707 different projects, distributed over 145 CWE types. Figure 3 shows the number of vulnerable functions from the top-20 CWE types. The benchmark consists of 25,440 functions, including 10,998 vulnerable and 14,442 non-vulnerable functions (i.e., patched versions of the prior vulnerable functions). The functions range in various sizes from 4 lines to 541 lines (from 2.5 to 97.5 percentiles), with a median size of 44 lines per function. The average number of statements changed in each function is around 4 (deleted) and 6 (added). Table 2 shows the overall statistics of the dataset along with the top 10 most vulnerable projects.

3.4 Contextual Information Collection

Real-world vulnerabilities are intricate and often result from the interaction of multiple entities. However, previous works do not incorporate this important feature into their datasets. Indeed, it is extremely arduous to manually check each function in the dataset and identify the required contexts. To this end, we harness the code-analyzing ability of LLMs to automatically extract required contexts for vulnerable functions.

We prompt GPT-4.1 with all the available information to identify the context required to understand the vulnerability in this function, and categorize them according to the five definitions. We provide the following information to the model: *the vulnerability type* (CWE-ID and description), *the full function body*, *the patch* (deleted-added lines), *the commit message*, and *the CVE description*. Using this detailed information, the model decides which symbols (variables, functions, etc.) are required or help to identify the vulnerability in the given function.

To measure how accurately GPT-4.1 can identify related contexts in a given function, the authors worked together and manually validated 1k randomly selected samples (around 10% of the whole dataset). Ground truth contexts were determined by tracing variables used in vulnerable statements back to their external symbols or verifying their origin within the function, with external symbols being defined as the relevant contextual elements. Heuristics were also used to capture additional potentially influential symbols (e.g., those that affect branching within `if` or `switch-case` statements). A prediction was deemed correct if GPT-4.1 identified the ground truth contexts, permitting the inclusion of up to one superfluous symbol within any single category. Samples where no relevant context could be identified from the function (e.g., only hard-coded strings changed in the function) were marked as 'N/A' and excluded from the evaluation, resulting in ten such cases out of the initial 1k samples. Among the remaining samples with identifiable context, GPT-4.1 achieved an accuracy of $83.16 \pm 6.93\%$ (with 99% Confidence Interval). Incorrect predictions resulted primarily from missing one or two required symbols or incorrectly including unrelated ones, particularly when analyzing larger code modifications. Figure 2 shows the distribution of the five context categories in the dataset.

4 Experiment Setup

4.1 Research Questions

RQ1: How effectively can LLMs detect vulnerable statements?

In this research question, our objective is to determine how effective LLMs are when directly applied to the task of identifying vulnerable statements within a function.

RQ2: Can an LLM-based agent, equipped with additional planning and contextual analysis capabilities, improve vulnerability-detection accuracy?

In this research question, we build upon the findings of RQ1 by exploring whether the introduction of an agent framework enables the model to reason more thoroughly about control flow, data dependencies, and code semantics, ultimately resulting in more accurate detection of vulnerable statements.

4.2 Evaluation Metrics

In this work, we use *Precision*, *Recall*, and *F1-Score* to measure the vulnerability detection performance of the models. Specifically, vulnerable instances are regarded as positive, and non-vulnerable instances as negative. For statement-level vulnerability detection, we measure a prediction as a True Positive if it correctly predicts vulnerable statements along with accurate reasoning. If the model correctly identifies a function as vulnerable, but the reasoning is incorrect, we consider this a True Positive for function-level detection but a False Negative for statement-level detection since it misses the correct vulnerability. Precision measures the likelihood that a prediction is correct when it identifies an instance as positive. Recall, on the other hand, measures the ability to correctly identify all the positive instances, even if the model makes some False Positives. Finally, F1-Score is the harmonic mean of both Precision and Recall.

Table 3: Statement-Level vulnerable detection performance of experimented LLMs

Model Name	Precision(%)	Recall(%)	F1-Score(%)
Deepseek-Coder-33B	0.28	0.24	0.26
Codestral-22B	1.12	0.67	0.84
Qwen2.5-Coder-32B	2.24	**22.33**	**4.07**
GPT-4.1	1.98	5.23	2.87
Claude-3.7-Sonnet	**3.75**	2.95	3.30

4.3 Model Setup

We selected five state-of-the-art LLMs for our evaluation tasks, including open-source and proprietary models. Specifically, we select *Deepseek-Coder-33B-Instruct* [9], *Codestral-22B-v0.1* [16], *Qwen2.5-Coder-32B-Instruct* [14], *GPT-4.1*, and *Claude-3.7-Sonnet* because these models are widely used in the community and have demonstrated high performance in various software engineering tasks. For the open-source models, we have used the weights from HuggingFace.

We report all LLM scores with pass@1 and use $temperature = 0.1$ for stable outputs as commonly used in the literature [4, 15]. We use pass@1 as it is more representative of the real scenario where a developer does not have the output reference to validate the attempts. All our experiments were carried out in a system with Intel(R) Xeon(R) Gold 6442Y CPU and a GPU cluster of 4 NVIDIA L40S.

5 RQ1: LLM Effectiveness in Vulnerability Detection

In this RQ, our objective is to assess how effectively LLMs can identify vulnerable statements when applied directly to the task at the statement level. To conduct this evaluation, we randomly pick 1k from our benchmark. Due to computational cost and inference constraints, we are unable to experiment with the full benchmark in this setting.

Approach: To find the effectiveness of LLMs in statement-level vulnerability detection, we asked the LLMs to analyze a function and identify if the function has vulnerable statements. We adopted a four-shot prompting technique with two examples from vulnerable functions and two from their corresponding non-vulnerable functions (patched vulnerable functions). We selected the few-shot examples from the same CWE type as the input function. If the model finds any vulnerability, then it shall output the vulnerable statements as a JSON list with a brief explanation. Otherwise, the model should output an empty list. We tweaked some instructions in the prompt based on each model's output behavior. After generating the outputs, we matched the output statements against the actual vulnerable lines in our benchmark. We identify an output to

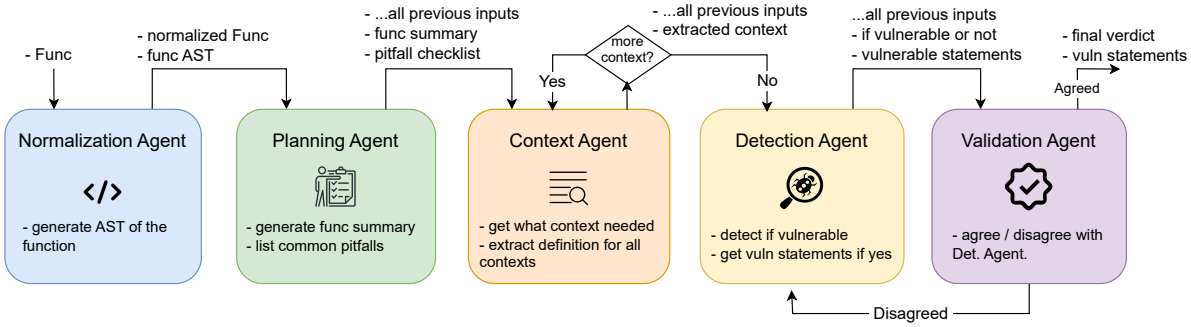

Figure 4: The multi-agent vulnerability detection pipeline.

be True Positive for statement-level detection if: ① the model outputs the exact vulnerable statements and the correct reasoning, ② if same vulnerability is fixed at multiple places in the function, the model at least returns one such statement with correct reasoning, ③ the model returns the vulnerable statements with correct reasoning and at most two unrelated statements, ④ the model outputs only the sink statements or both vulnerable and sink statements with correct reasoning.

Results: The results are presented in Table 3. As shown, model performance falls far short of practical usability. The best-performing model in terms of F1-score, *Qwen2.5-Coder-32B*, achieved only 4.07%, with a precision of 2.24%. *Claude-3.7-Sonnet* followed with an F1-score of 3.30% and also had the highest precision with 3.75%, while *GPT-4.1* showed opposite performance with higher recall but lower precision. Other open-source models, including *DeepSeek-Coder* and *Codestral*, all recorded F1-scores below 1%. The relatively high recall of *Qwen2.5-Coder-32B* suggests that the model tends to over-predict vulnerabilities, often labeling benign lines as vulnerable. This results in a lot of false positives, which limits its practical applicability and further motivates our agent-based solution for vulnerability detection.

Answer to RQ1: The models are not effective at detecting vulnerable statements, as the best performing Qwen2.5-Coder-32B model achieves only 4.07% F1-score. SecVulEval's diverse set of statement-level vulnerabilities uncovers LLMs' severe lack of ability to find vulnerable statements in real-world C/C++ code.

6 RQ2: Performance of Agentic LLMs in Vulnerability Detection

6.1 Multi-agent Vulnerability Detection Pipeline

Results in RQ1 show that single LLMs perform very poorly on the C/C++ vulnerability detection task, which aligns with prior studies that explore LLMs' performance on function-level binary vulnerability detection [5]. Therefore, we adopt a multi-agent pipeline for vulnerability detection as illustrated in Figure 4. These LLM-based agents have separate responsibilities and complete the entire task through collaboration. This type of approach is more effective than a single LLM by multiple studies [18, 24, 25].

The pipeline has five agents, four of which are LLM-based. The first agent, called the *Normalization Agent*, receives a function body. It generates the AST of the given function using *tree-sitter*. Then, the normalized function body and the AST are passed to the *Planning Agent*. Using this information, the *Planning Agent* leverages an LLM to generate a summary of the function and create a checklist of some common pitfalls that can make this function vulnerable. The inputs to this agent, the generated summary, and the checklist are sent to the *Context Agent*. The *Context Agent* queries an LLM to identify external symbols required to detect vulnerability in the given function; the LLM returns a list of necessary symbols. This interaction constitutes an iterative process where the agent repeatedly prompts the LLM to determine if more contextual information is necessary, continuing until the LLM indicates sufficient context or a predefined number of attempts are exhausted. After the list of contexts is finalized, the definitions for these external symbols are extracted using *tree-sitter*. Next, the pipeline moves to the *Detection Agent*, which uses all the inputs and outputs from the previous agents, and decides if the function is vulnerable or not. If a vulnerability is detected, the LLM outputs the statements that are vulnerable and a short reasoning about the vulnerability. Finally, all the outputs, except the reasoning from the *Detection Agent*, are given to the *Validation Agent*. The LLM is asked whether it agrees with the decision of the *Detection Agent*. If the *Validation Agent* does not agree, then the *Detection Agent* is executed again, and this loop continues until both agents agree or a predefined number of attempts are exhausted. The maximum retries for *Context Agent* and *Validation Agent* are three each in our experiments.

6.2 Experiments

To evaluate the performance of the multi-agent-based approach as described in Section 6.1, we run our evaluation on the output of the Validation Agent. If the model finds any vulnerability, then it shall output each vulnerable statement and its reason as a pair. Otherwise, the model should output an empty list, and 'is_vulnerable' field in the output will be 'false'. Since the outputs include explanations for the vulnerability, it is not possible to automatically evaluate the results. In addition, the statements returned by the model may not always be the exact same statements as the changed statements. For example, sometimes the models output the 'sink' statement as vulnerable (where the vulnerability causes a crash or other symptoms) instead of the 'source' (where the vulnerability is

Table 4: Vulnerability detection performance of LLM-driven agents on Func-level (whether the function is vulnerable or not) and Stat-level (if the model identified vulnerable statements in the function with correct reasoning)

Models used in the agents	Precision (%)		Recall (%)		F1-Score (%)	
	Func-Level	Stat-Level	Func-Level	Stat-Level	Func-Level	Stat-Level
Qwen2.5-Coder-32B	35.48	12.10	34.92	15.46	35.20	13.57
Deepseek-Coder-33B	45.16	5.65	46.28	9.72	45.71	7.14
Codestral-22B	**47.41**	12.93	45.83	18.75	46.61	15.31
GPT-4.1	40.65	14.49	73.11	49.21	52.25	22.38
Claude-3.7-Sonnet	41.86	**15.35**	**75.63**	**53.23**	**53.89**	**23.83**

Table 5: Precision (P), Recall (R), and F1 scores of the models for statement-level vulnerability detection in each CWE type. It shows the top-10 most frequently detected CWE types. The CWE types where the models could not correctly detect any statement are marked with '-'.

Models		CWE-119	CWE-125	CWE-787	CWE-189	CWE-190	CWE-476	CWE-362	CWE-369	CWE-401	CWE-416
Qwen2.5-Coder-32B	P	**20.0**	13.6	10.0	-	14.3	**15.4**	-	-	50.0	20.0
	R	30.0	30.0	12.5	-	8.3	28.6	-	-	50.0	12.5
	F1	24.0	18.8	11.1	-	10.5	20.0	-	-	50.0	15.4
Deepseek-Coder-33B	P	-	5.9	10.0	33.3	14.3	8.7	-	-	-	-
	R	-	9.1	16.7	50.0	50.0	22.2	-	-	-	-
	F1	-	7.1	12.5	40.0	22.2	12.5	-	-	-	-
Codestral-22B	P	-	7.1	**20.0**	**66.7**	28.6	12.5	-	-	**66.7**	5.9
	R	-	10.0	28.6	66.7	66.7	33.3	-	-	100.0	9.1
	F1	-	8.3	**23.5**	**66.7**	40.0	18.2	-	-	**80.0**	7.1
GPT-4.1	P	17.4	8.8	13.6	40.0	27.3	13.6	-	33.3	50.0	23.1
	R	**57.1**	**75.0**	**75.0**	**100.0**	75.0	**100.0**	-	**100.0**	**100.0**	17.6
	F1	**26.7**	15.8	23.1	57.1	40.0	**24.0**	-	50.0	66.7	20.0
Claude-3.7-Sonnet	P	13.0	**13.8**	14.3	40.0	**33.3**	9.8	**20.0**	**66.7**	**66.7**	**31.3**
	R	42.9	50.0	60.0	**100.0**	**100.0**	**100.0**	**66.7**	**100.0**	**100.0**	**33.3**
	F1	20.0	**21.6**	23.1	57.1	**50.0**	17.8	**30.8**	**80.0**	**80.0**	**32.3**

introduced), or sometimes return both statements. For statement-level detection, we utilize the same approach as described for RQ1. For function-level detection, we automatically match the output of the 'is_vulnerable' field with the ground truth from the dataset. We use the same dataset as RQ1 for our experiments in this RQ.

6.3 Results

Table 4 shows that statement-level detection performance remains low. The best model, Claude-3.7-Sonnet, achieves only a 23.83% F1-score and 15.35% precision, with GPT-4.1 close behind. Open-source models perform worse overall. Closed-source models like Claude and GPT-4.1 adopt a more aggressive detection strategy, reflected in their higher recall (e.g., Claude's 53.23% vs. Codestral's 18.75%) but lower precision due to more false positives. This trend is even more pronounced at the function level, where Claude and GPT-4.1 show very high recall (75.63%, 73.11%) but still lower precision than open-source models, suggesting a tendency to over-flag functions as vulnerable.

Note that when comparing the performance of these models on function-level and statement-level vulnerability detection in Table 4, we can observe that identifying whether a function is vulnerable and pinpointing the exact vulnerable statements are distinct tasks, the latter being significantly more challenging with lower precision and recall values. Both open-source and proprietary LLMs show significant drops in precision and recall when required to locate vulnerable statements with correct root cause explanations. Unlike function-level detection, statement-level analysis demands fine-grained reasoning about issues like pointer arithmetic and memory bounds, requiring deeper contextual and interprocedural understanding. This raises concerns about the reliability of function-level predictions. *Risse et al.* [21] have shown that models may rely on spurious patterns rather than true vulnerabilities. Our results echo this, as models often fail to find the real vulnerability even when correctly flagging a function. Thus, advancing statement-level detection is essential, as without it, developers risk being misled by false positives or missing subtle but critical flaws.

Table 5 reports model performance per CWE type. We present the ten most frequently occurring CWEs, as the remaining categories appear too infrequently to support statistically meaningful comparisons. As shown, several CWEs (CWE-125, CWE-787, CWE-189, CWE-190, CWE-476, CWE-401, and CWE-416) are detected by most models, whereas CWE-362 and CWE-369 are identified

only by Claude-3.7-Sonnet and GPT-4.1. Although CWE-401 occurs relatively rarely in the dataset, it is detected reliably by most models. In contrast, CWE-20 is the fourth most common vulnerability (see Figure 3) yet is detected only by GPT-4.1. Likewise, CWE-120 and CWE-415 appear with moderate frequency but are not detected by any model, suggesting that these vulnerabilities are more challenging for current systems to recognize.

Answer to RQ2: Agentic approaches in a multi-agent orchestration can further improve the effectiveness of detecting vulnerable statements in C/C++ functions, with the best-performing model, *Claude-3.7-Sonnet*, achieving an F1-score of 23.83%.

7 Threats to Validate

The contextual information integrated into our dataset was automatically extracted by an LLM. While a manual validation performed on 1k random samples indicated an accuracy of 83.16 ± 6.93% for correctly identified contexts, this assessment method has inherent limitations. The very definition of "relevant context" is not an objective quantity, leading to potential inter-annotator variability and dependence on subjective human interpretation. Additionally, the dataset does not extend to repository-wide contexts, such as transitive effects of non-local interactions beyond immediate function calls and domain-specific knowledge of the projects. Despite leveraging available information, e.g., patch, commit description, CVE description, and issue tracking details (if available), manual validation poses several methodological limitations. The inherent subjectivity in defining precise vulnerability characteristics can lead to variability in judgments. To mitigate this, we define specific criteria (as detailed in our approach). In addition, manual analysis is a labor-intensive process that imposes significant scalability constraints. We evaluated the largest size of each model that we could run with our resources. We could not investigate intra-model size variability since running different sizes of the same model significantly increases the experiment scale and is not feasible with our resources and time constraints. Therefore, the same conclusion may not hold for the same models of smaller sizes and needs further exploration.

8 Conclusion

This paper presents SecVulEval, a novel dataset for C/C++ vulnerability detection that emphasizes statement-level granularity and rich contextual information. WLeveraging SecVulEval, we evaluated five state-of-the-art LLMs on vulnerability detection in both standalone and multi-agent settings. The results show that current LLMs perform poorly in detecting vulnerabilities, with Claude-3.7-Sonnet achieving the highest F1-score of 23.83%. Claude-3.7-Sonnet also outperformed others in identifying the contextual information necessary for detection. With its fine-grained labels and contextual annotations, SecVulEval provides a valuable benchmark for training and evaluating models on real-world vulnerability detection at the statement level.

References

[1] G. Bhandari, A. Naseer, and L. Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *pROMISE*, pages 30–39, 2021.

[2] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *TSE*, 48(9):3280–3296, 2021.

[3] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 654–668, 2023.

[4] M. DeLorenzo, V. Gohil, and J. Rajendran. Creativeval: Evaluating creativity of llm-based hardware code generation. *arXiv preprint arXiv:2404.08806*, 2024.

[5] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[6] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[7] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[8] Y. Gu, H. Shu, and F. Kang. Binaiv: Semantic-enhanced vulnerability detection for linux x86 binaries. *C&S*, 135:103508, 2023.

[9] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[10] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino. Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In *European symposium on research in computer security*, pages 271–289. Springer, 2024.

[11] J. He and M. Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879, 2023.

[12] J. R. Henriques, J. D'Abruzzo Pereira, and M. Vieira. Mining vulnerability and code repositories to study software security. In *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing*, pages 11–16, 2024.

[13] D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

[14] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[15] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904*, 2023.

[16] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[17] H. Lee, Z. Zhang, H. Lu, and L. Zhang. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. *arXiv preprint arXiv:2506.11791*, 2025.

[18] J. Li, Q. Zhang, Y. Yu, Q. Fu, and D. Ye. More agents is all you need. *arXiv preprint arXiv:2402.05120*, 2024.

[19] J. Lin, H. Zhang, B. Adams, and A. E. Hassan. Vulnerability management in linux distributions: An empirical study on debian and fedora. *Empirical Software Engineering*, 28(2):47, 2023.

[20] C. Ni, L. Shen, X. Yang, Y. Zhu, and S. Wang. Megavul: Ac/c++ vulnerability dataset with comprehensive code representations. In *MSR*, 2024.

[21] N. Risse and M. Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *arXiv preprint arXiv:2408.12986*, 2024.

[22] Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang. Lprotector: An llm-driven vulnerability detection system. *arXiv preprint arXiv:2411.06493*, 2024.

[23] N. Shiri Harzevili, A. Boaye Belle, J. Wang, S. Wang, Z. M. Jiang, and N. Nagappan. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys*, 57(3):1–36, 2024.

[24] K. Sreedhar and L. Chilton. Simulating strategic reasoning: Comparing the ability of single llms and multi-agent systems to replicate human behavior. 2025.

[25] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

[26] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah. How effective are neural networks for fixing security vulnerabilities. In *ISSTA*, pages 1282–1294, 2023.

[27] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang. Knighter: Transforming static analysis with llm-synthesized checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 655–669, 2025.

[28] X. Zhou, T. Zhang, and D. Lo. Large language model for vulnerability detection: Emerging results and future directions. In *ICSE*, pages 47–51, 2024.

[29] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

Received 28 September 2023; revised 5 March 2024; accepted 16 April 2024