arXiv:2505.19395v1 [cs.CR] 26 May 2025

VADER: A Human-Evaluated Benchmark for Vulnerability Assessment, Detection, Explanation, and Remediation

Ethan TS. Liu
AfterQuery, UC Berkeley
ethan@afterquery.com

Austin Wang
AfterQuery, University of Pennsylvania
austin@afterquery.com

Spencer Mateega
AfterQuery
spencer@afterquery.com

Carlos Georgescu
AfterQuery
carlos@afterquery.com

Danny Tang
AfterQuery
danny@afterquery.com

Abstract

Ensuring that large language models (LLMs) can effectively assess, detect, explain, and remediate software vulnerabilities is critical for building robust and secure software systems. We introduce VADER, a human-evaluated benchmark designed explicitly to assess LLM performance across four key vulnerability-handling dimensions: assessment, detection, explanation, and remediation. VADER comprises 174 real-world software vulnerabilities, each carefully curated from GitHub repositories and annotated by security experts. For each vulnerability case, models are tasked with identifying the flaw, classifying it using Common Weakness Enumeration (CWE), explaining its underlying cause, proposing a patch, and formulating a test plan. Using a one-shot prompting strategy, we benchmark six state-of-the-art LLMs (Claude 3.7 Sonnet, Gemini 2.5 Pro, GPT-4.1, GPT-4.5, Grok 3 Beta, and o3) on VADER, and human security experts evaluated each response according to a rigorous scoring rubric emphasizing remediation (quality of the code fix, 50%), explanation (20%), and classification and test plan (30%) according to a standardized rubric. Our results show that current state-of-the-art LLMs achieve only moderate success on VADER—OpenAI's o3 attained 54.7% accuracy overall, with others in the 49-54% range, indicating ample room for improvement. Notably, remediation quality is strongly correlated (Pearson $r > 0.97$) with accurate classification and test plans, suggesting that models that effectively categorize vulnerabilities also tend to fix them well. VADER's comprehensive dataset, detailed evaluation rubrics, scoring tools, and visualized results with confidence intervals are publicly released, providing the community with an interpretable, reproducible benchmark to advance vulnerability-aware LLMs. All code and data are available at: https://github.com/AfterQuery/vader.

Preprint. Under review.

1 Introduction

Large language models are increasingly used to assist in software development, raising both hopes and concerns regarding code security. On one hand, LLMs offer the promise of automatically detecting and even fixing vulnerabilities in code. Major industry players have begun integrating LLMs into security tools. For example, GitHub's Advanced Security now leverages AI to suggest fixes for detected code vulnerabilities via Copilot autofix [5], and OpenAI's ChatGPT has introduced a GitHub "connector" that allows analyzing codebases for security issues [11]. These developments underscore a growing applied interest in using LLMs for coding tasks. However, vulnerability analysis and remediation, specifically in multi-language environments, remains an under-explored area. On the other hand, systematically evaluating an LLM's capability in this domain remains an open challenge. Prior datasets for vulnerability detection often focus on synthetic or single-language code and evaluate only detection accuracy [15, 4, 7].

```
def validd():
    code = input("Enter book
        ↪ code: ")
    if not is_valid(code):
        print("Invalid code. Try
            ↪ again.")
        validd()  # Dangerous
            ↪ recursion
```

(a) Before patch (recursive stack overflow)

```
def validd():
    retries = 0
    while retries < 3:
        code = input("Enter book
            ↪ code: ")
        if is_valid(code):
            break
        print("Invalid code. Try
            ↪ again.")
        retries += 1
```

(b) After patch (iterative fix)

Test	Expected Result
Input "ZZZZ" repeatedly	RecursionError (stack overflow)
Input special characters (e.g., `!@#$%`) repeatedly	Stack overflow
Press Enter (empty input) repeatedly	Stack overflow
Input very long string (e.g., 1000 characters) repeatedly	Stack overflow
Input only whitespace (spaces or tabs) repeatedly	Stack overflow
Alternate invalid inputs (e.g., `123`, `abc`, empty) repeatedly	Stack overflow
Enter valid borrower ID, then invalid codes repeatedly	Stack overflow
Enter one valid book code, then revert to invalids repeatedly	Stack overflow resumes

(c) Test cases used to confirm stack overflow in unpatched code

Figure 1: One-shot example passed to LLMs, omitting input files due to length. Patch illustration and test cases for CWE-674 (Uncontrolled Recursion). Full patch format is included in Appendix A.

Furthermore, typical automated metrics (e.g. pass@k for code generation) do not directly capture whether a model-produced patch truly fixes a security bug or if its explanation is correct. There is a clear need for a comprehensive benchmark that assesses all parts of security: Finding a vulnerability, explaining the issue, fixing the code, and generating a test plan that properly verifies it.

In this work, we introduce VADER: **V**ulnerability **A**ssessment, **D**etection, **E**xplanation, and **R**emediation—a human-annotated benchmark that evaluates large language models (LLMs) on end-to-end software-security assistance. We use a one-shot prompting method; the example is shown in Figure 1. Comprehensive statistics are summarized in Figures 2 and 3. Our benchmark is distinguished by the *following five characteristics*:

Expert-Curated Ground-Truth. All 174 *real-world* vulnerability cases are submitted by experienced cybersecurity experts and double-checked by an independent reviewer, each with over 6 years of cybersecurity experience, guaranteeing reliable labels and patches for evaluation.

Comprehensive Four-Stage Evaluation Protocol. We utilize a subdivision of each case into four tasks, which are sorted into three rubric buckets, as shown below in Table 1.

Table 1: Mapping between the tasks of a security engineer and the scoring buckets used in VADER.

Task	What the model must do	Rubric bucket
Classification/Assessment	Identify the correct CWE category and assign the appropriate severity level.	Other
Explanation	Pinpoint the root cause and describe its impact.	Explanation
Remediation	Produce a clear, compile-ready patch that eliminates the vulnerability.	Remediation
Test Plan	Outline concrete steps or inputs that confirm the fix.	Other

Breadth of Languages and Vulnerability Types. Cases span 15 programming languages—most frequently JavaScript and Python (45% each), but also TypeScript, PHP, Go, C/C++, HTML/CSS, Shell, Solidity, Java, Ruby, and more—capturing a wide spectrum of real security flaws.

Realistic Multi-File / Multi-Language Context. Over 75% of the benchmark is multi-language and 23% involve up to four source files, reflecting the cross-file logic and heterogeneous stacks often seen in production-level code.

Severity-Focused Sampling. Using the 5-level rubric in Table 2, High (Level 4) and Critical (Level 5) issues dominate (41% and 20%, respectively; see Figure 2), ensuring VADER stresses vulnerabilities of serious business impact while still containing lower-severity examples (18%) to test fine-grained discrimination.

Table 2: Concise five-level severity rubric used in VADER.

Level	Description	Criteria
1	Very Low	Latent weakness not currently exploitable.
2	Low	Hard-to-exploit or low-impact bug.
3	Medium	Exploitable issue with limited scope.
4	High	Easily exploited flaw with major impact.
5	Critical	Grants full compromise or breaks functionality.

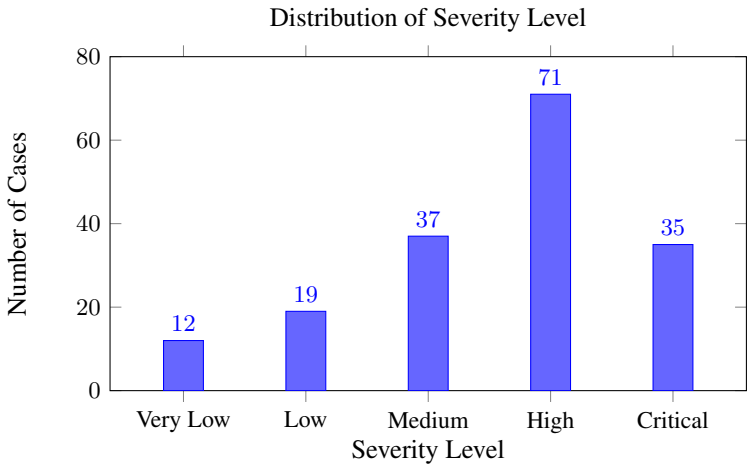


Figure 2: Distribution of severity levels across VADER cases.

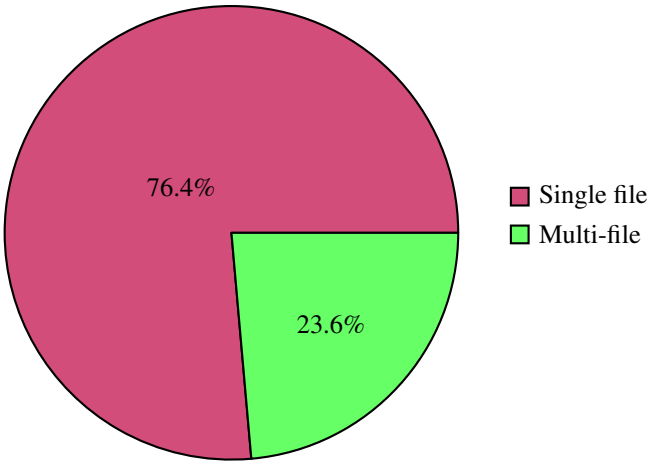


Figure 3: Distribution of Cases by Number of Files per Case

2 Related Work

2.1 Synthetic Vulnerability Dataset Benchmarks

Early research on machine learning for code security relied heavily on synthetic or static benchmarks. The NIST JULIET test suite is a prime example, providing thousands of small C/C++ and Java programs with known vulnerable and safe variants for dozens of CWE categories [3]. JULIET (part of the SARD repository) has been widely used to evaluate static analyzers and as training data for vulnerability detectors [2]. For instance, one of the first deep learning vulnerability detectors, VULDEEPECKER, introduced its own dataset but also leveraged such synthetic examples [8]. Moreover, DEVIGN [19, 7] and BIG-VUL [4] frame vulnerability analysis as a *detection* problem: given a single function, classify it as vulnerable or not. Although valuable for training classifiers, these corpora merely rely on synthetic or simplified examples and never require an explanation / patch.

These static benchmarks offer reliable ground truth but cover only simplified scenarios. They focus narrowly on vulnerability presence detection (e.g., buffer overflow or SQL injection in a given function) and often assume a single vulnerability per sample. Consequently, models tuned to these datasets can overestimate real-world performance [6].

2.2 Real-World Code Benchmarks

To move beyond synthetic examples, more recent real-world datasets expand both language coverage and task scope. SECURITYEVAL [15] curates 130 vulnerabilities across 75 CWE types and measures code-generation models on detection and patch synthesis. REPOSVUL, CROSSVUL, VULEVAL [16, 10, 18] add multi-file contexts drawn from production repositories. Researchers have also curated larger benchmarks from real code. BIG-VUL and MEGA-VUL are collections of thousands of real vulnerable functions and their fixes mined from open-source C/C++ projects (with links to CVE records) [14, 9]. Such datasets enable vulnerability classification (e.g., predicting if a given function is vulnerable and sometimes identifying the vulnerability type) on more realistic codebases.

On the remediation side, several benchmarks aggregate code patches that fix security bugs. CVEFIXES and PATCHDB automatically gathered thousands of vulnerability-fixing commits from open-source software, pairing vulnerable code with the corrected code [1, 17]. These resources support evaluation of automated patching: given a vulnerable snippet, generate or identify the correct fix. Additionally, Ponta et al. manually curated a dataset of real fixes to known vulnerabilities to facilitate studies on vulnerability mitigation [12].

Despite this progress, existing static benchmarks still omit two key dimensions: (i) structured root-cause *explanations* and (ii) *test plans* that verify a fix. VADER is broader in scope, combining classification of vulnerability type, explanation of the issue, proposal of a code fix, and even test-case generation to validate the fix.

2.3 Interactive / Agent-Based Evaluations

A parallel line of work probes whether LLM *agents* can exploit and repair live systems. CVE-BENCH [20] supplies Docker targets with real common vulnerabilities and exposures (CVEs) and lets an autonomous tool chain iterate until the vulnerability is fixed. Similarly, the NYU CTF benchmark [13] collects real-world CTF challenges (spanning web, binary exploitation, cryptography, etc.) to test an AI agent's prowess in finding flags (exploiting vulnerabilities). These interactive evaluations push beyond static analysis by requiring a sequence of actions (reconnaissance, exploit, and sometimes patch).

However, they primarily assess attack performance or autonomous patch deployment, without examining an AI's ability to *explain* vulnerabilities or generate *human-readable* remediation plans. This leaves a gap in evaluating how well systems can reason about vulnerabilities in depth: describing *why* code is insecure, *fixing* it, and *validating* the fix. VADER addresses this gap by providing a comprehensive, human-evaluated benchmark that spans from detection and explanation to repair and test planning, covering the full vulnerability handling lifecycle in code.

3 Benchmark Construction and Analysis

3.1 Construction

VADER's dataset was constructed through a rigorous double-annotator process mirroring real-world secure code review. In total, 174 real vulnerabilities were curated from open-source software, emphasizing projects with real-world functionality (e.g., web servers, CLI tools, databases) in popular languages (Python, Java, JavaScript, C/C++, Go, etc.). Each case underwent two stages: (1) an initial submission by a vulnerability author and (2) a subsequent independent review to ensure accuracy and consistency.

Case Submission. In the first stage, security expert annotators identified a genuine, non-trivial vulnerability in an open-source codebase and prepared a full report for that case. Annotators were instructed to target security-relevant flaws (e.g., issues in input handling, access control, memory management) that were real and demonstrable (not hypothetical or toy examples), often involving complex or multi-file code logic. For each vulnerability, the submitter extracted the relevant code snippet(s) exhibiting the flaw and provided all required context and documentation. This included classifying the vulnerability with the appropriate CWE identifier (and, optionally, an OWASP Top-10 category if applicable) and writing a concise natural-language explanation of the issue. The explanation (typically 2–5 sentences) was expected to clearly describe the root cause of the bug, how an attacker could exploit it, and the potential impact or damage. The submitter also proposed a golden patch—a minimal, clean code fix addressing the root cause without unnecessary changes—and supplied a test case or test plan to validate the fix. Additionally, each case was assigned a severity level (1-5) based on a standard rubric (see Table 2) considering the vulnerability's exploitability, scope, and potential damage.

Review and Validation. In the second stage, every submission was independently reviewed by one of five hand-selected security engineers (min. 6 years of experience) to ensure it met all quality criteria before inclusion in the benchmark. The reviewer verified that the reported flaw was indeed a real, impactful vulnerability and not a trivial bug or false issue. They checked that the correct CWE category was assigned and that the written explanation covered all key aspects (the underlying cause of the vulnerability, why it occurred, and an exploit scenario demonstrating how an attacker could leverage it). The proposed patch was scrutinized to confirm it truly fixed the vulnerability at its source while adhering to best practices (e.g., input validation or proper error handling) and minimal change principles (avoiding large refactors or unrelated modifications). The reviewer also ensured the severity rating was appropriate (following the defined severity rubric) and that the provided test case(s) effectively demonstrated the vulnerability's presence before the patch and its resolution after the patch. Submissions that did not satisfy any of these requirements were revised or rejected. Only after the second annotator's approval was a vulnerability instance accepted into the VADER dataset. This two-tier annotation procedure yielded a high-quality benchmark of thoroughly documented vulnerabilities, each with a verified flaw, explanation, fix, and validation test.

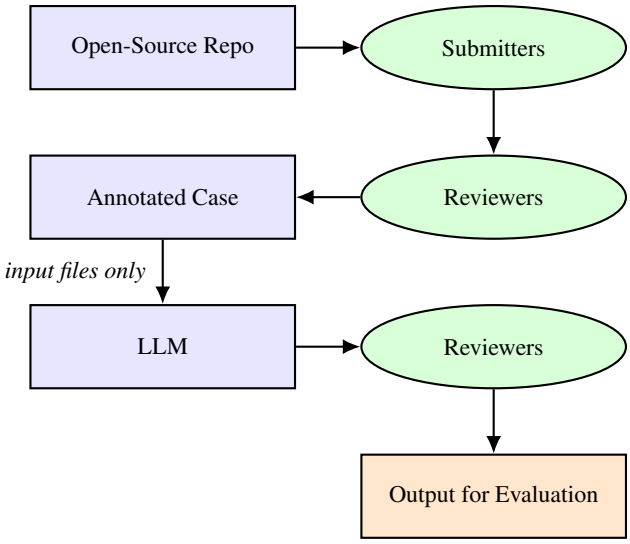


Figure 4: Curation and evaluation pipeline for VADER.

The detailed scoring sheet used during annotation can be found in Appendix C. Annotated cases that do not meet submission criteria are rejected or revised.

3.2 Analysis

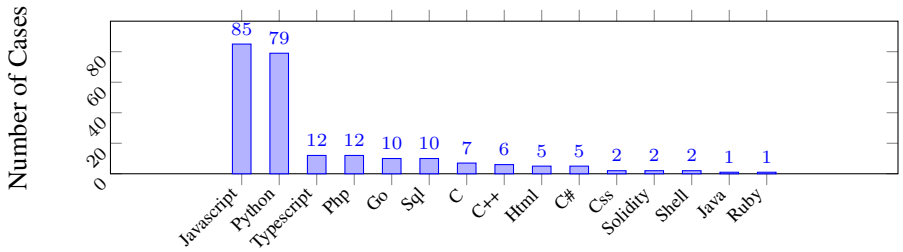


Figure 5: Distribution of programming languages across VADER cases.

VADER encompasses 15 programming languages. In Figure 5, JavaScript and Python are the most frequent, trailed by web scripting languages like TypeScript and PHP. Go and SQL reflect moderate representation of cloud backend and database workloads. To further explore the relationships between languages, we examine their co-occurrences in vulnerability cases. Understanding these co-occurrence patterns can also reveal common tech stacks that lead to vulnerabilities. In Figure 6, Javascript and Python exhibit the highest co-occurrence, followed by Python and SQL. Additionally, HTML and Javascript, as well as Javascript and PHP, co-occur often. These patterns reflect frontend web development and backend database workflows. There are distinct file distribution patterns. As shown in Figure 7 in appendix, Go, SQL, C, and C++ all have 2-3 of their cases involving 2–5 files, indicating more complex code bases. Javascript, has many cases with more than 6 files, reflecting the extensive code changes often required in web applications for vulnerability remediation, a pattern consistent with its prevalence in CWE-79 (Cross-Site Scripting) cases observed in Figure 8b in the appendix. Python demonstrates sizable cases involving multiple files, due to its association with vulnerabilities like CWE-89 (SQL Injection) as seen in Figure 8b in the appendix.

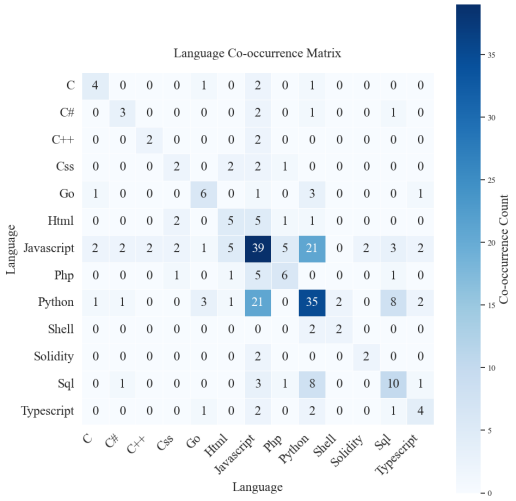


Figure 6: Language co-occurrence matrix showing the frequency of language pairs across cases.

4 Evaluation

Model Selection. We selected six state-of-the-art LLMs based on their popularity, architectural diversity, extended context support, and prominence in academia/industry. The models are OpenAI's o3, Google's Gemini 2.5 Pro, Anthropic's Claude 3.7 Sonnet, xAI's Grok 3 Beta, and two versions of OpenAI's GPT-4 (denoted GPT-4.1 and GPT-4.5). These represent a broad cross-section of leading large models: for example, Claude 3.7 is known for its 64k-token extended context (which we leveraged to handle our longest multi-file inputs), while Gemini and Grok introduce architectural variety from Google and xAI, respectively. All chosen models support large context windows (at least 128k tokens or more), a necessary feature for VADER's multi-file vulnerability cases. By including multiple vendors and model families (OpenAI's GPT-series and new o-series, Anthropic's Claude, Google's PaLM/Gemini, and xAI's Grok), we aimed to make the evaluation comprehensive and representative of the best available LLMs in 2025.

Evaluation Framework. We perform a one-shot evaluation with the prompt, shown in Table 7. This prompt was used uniformly across all models and for every test case, without any model-specific tuning. Each LLM was given the exact same one-shot demonstration and task description, then asked to produce a solution for the target vulnerable code.

Table 3: Model scoring rubric used by reviewers to evaluate outputs.

Category (Pts)	Give full points	Give partial points	Give 0 points
Explanation (0–2)	Crystal-clear and technically accurate; explains root cause and impact.	Understandable but missing impact or has minor technical flaws.	Missing, incorrect, or irrelevant explanation.
Remediation (0–5)	Fix is minimal, compiles, style-compliant, and eliminates the vulnerability.	Fix resolves issue but is messy, has style issues, or introduces minor risk.	Does not fix the issue, breaks code, or introduces new vulnerability.
Other (0–3)	CWE classification is exact; test plan runnable and covers both success and failure.	Only CWE or test plan is partially correct or missing.	Both CWE and test plan are incorrect or missing.

Thus, each model's output on a single case received a total score from 0 to 10 by summing these components. All models' outputs for a given case were scored by at least two independent expert evaluators (security researchers) following this rubric. The evaluators were blind to which model produced which output, to reduce bias.

5 Results

Primary Results. We find that all six LLMs achieve only moderate success on the VADER benchmark – even the top model solves just over half of the issues according to our strict rubric. OpenAI's o3 model ranks first, with an average Final score of about 5.47 out of 10 (54.7%). Google's Gemini 2.5 Pro is the runner-up at roughly 5.2/10, and OpenAI's GPT-4.5 comes in third (5.0/10). At the lower end, xAI's Grok 3 Beta has the lowest performance with an average around 4.4/10 (44%), while Anthropic's Claude 3.7 Sonnet (4.9) and the older GPT-4.1 (4.8) fall in the middle of the pack. In absolute terms, the gap between the best and worst model is only about 1 point (out of 10), highlighting that no current model excels at this task; none approaches a near-perfect score. This underscores the difficulty of the VADER benchmark, as even highly advanced LLMs can at best remediate a little over half of the vulnerabilities correctly on average.

In summary, our results show that OpenAI's o3 is the top-performing model on VADER, but the margin over other leading LLMs is not huge. Performance on the benchmark is capped at a relatively low absolute level for all models – even o3 on average missed nearly half the points. The high correlation between remediation and classification/test-plan ability suggests that improving a model's deep understanding of code vulnerabilities will yield gains across multiple evaluation aspects simultaneously. Future models that can more reliably identify and fix complex vulnerabilities (perhaps via better reasoning or domain knowledge) should also naturally provide better explanations and test plans. The VADER benchmark thus provides a rigorous measure of these intertwined capabilities, and our evaluation highlights that significant improvements are needed before automated code assistants can consistently handle real-world security flaws.

(a) Overall performance across models

Statistic	**Claude-3.7**	**Gemini-2.5-Pro**	**GPT-4.1**	**GPT-4.5**	**o3**	**Grok 3 Beta**
Mean	52.31%	53.58%	50.00%	49.19%	54.62%	52.02%

(b) <Remediation>

Statistic	Claude	Gemini	GPT-4.1	GPT-4.5	o3	Grok
Mean	52.30%	52.76%	49.08%	49.20%	54.60%	51.38%
Std	47.34%	47.22%	46.65%	47.92%	48.67%	47.37%
25%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
50%	80.00%	80.00%	60.00%	60.00%	90.00%	60.00%
75%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

(c) <Explanation>

Statistic	Claude	Gemini	GPT-4.1	GPT-4.5	o3	Grok
Mean	53.74%	56.03%	53.45%	50.29%	55.46%	53.74%
Std	48.39%	48.75%	49.15%	48.83%	49.41%	48.98%
25%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
50%	100.00%	100.00%	100.00%	50.00%	100.00%	100.00%
75%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

(d) <Other> (CWE + severity)

Statistic	Claude	Gemini	GPT-4.1	GPT-4.5	o3	Grok
Mean	51.72%	53.83%	49.81%	48.66%	54.21%	52.11%
Std	47.21%	47.76%	47.24%	47.76%	48.40%	48.00%
25%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
50%	66.67%	66.67%	66.67%	66.67%	83.33%	66.67%
75%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Table 4 Performance comparison across four evaluation dimensions.

6 Limitations

Selection of Models. Our study evaluated only a narrow selection of models—specifically six proprietary state-of-the-art LLMs, due to compute and access constraints. We did not include open-source or specialized code-focused models (e.g., LLaMA 2, Mistral, Code LLaMA, WizardCoder, etc.), which limits the breadth of our comparisons. This constrained model pool may reduce the generalizability of our findings: the results primarily characterize the chosen systems and might not hold for other LLMs. For example, recent instruction-tuned code models like WizardCoder (an open-source 15B parameter model) have demonstrated performance on coding benchmarks that rivals or even surpasses some closed-source models like Anthropic Claude and Google Bard (SOURCE). Because such models were excluded, our benchmark cannot confirm whether similar success (or failure) patterns would occur with them.

Resource and Cost Constraints. These limited the scale and scope of our inference runs, especially for cases requiring long context or multi-file analysis. Some real-world vulnerabilities span multiple files or large codebases, pushing beyond the context window and budget of our evaluation setup. We had to uniformly apply one-file-at-a-time, one-shot prompting for all models, which meant certain complex cases could not be fully tested in their entirety. This restriction forced us to drop particularly large cases), thereby narrowing the benchmark's coverage of very complex vulnerabilities. Consequently, one should be careful in extrapolating our results to large-scale industrial codebases or vulnerabilities that require cross-file reasoning.

7 Conclusion

This paper introduces a novel benchmark, VADER, a human-evaluated benchmark for assessing large language models on end-to-end software vulnerability handling. VADER contains 174 real-world cases annotated by security experts, covering detection, CWE classification, explanation, patching, and test plan generation. Models are evaluated using a rigorous rubric weighted toward remediation (50%) and demonstrate only moderate performance (top score: 54.7%). VADER spans 15 languages, multi-file scenarios, and a 5-level severity scale to stress real-world complexity. All benchmark data, evaluation tools, and results are publicly released to support reproducible progress in vulnerability-aware LLMs. All code and data is available at

Acknowledgments and Disclosure of Funding

We thank the AfterQuery operations and engineering teams for their support throughout the development of the VADER benchmark. In particular, we are grateful to Spencer Mateega, Carlos Georgescu, and Danny Tang for their contributions to research discussions and supervision. Moreover, we acknowledge the security engineers and annotators at AfterQuery who contributed to the expert-curated vulnerability cases and performed a detailed evaluation of model responses. We compensated annotators at a rate of $30/hour for their expertise and time.

This work was supported by AfterQuery Inc., which provided funding, infrastructure, and personnel for dataset construction, annotation tooling, and evaluation infrastructure. The authors declare no competing financial interests outside the submitted work.

References

[1] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, August 2021. doi: 10.1145/3475960.3475985.

[2] Yingzhou Bi, Jiangtao Huang, Penghui Liu, and Lianmei Wang. Benchmarking Software Vulnerability Detection Techniques: A Survey, March 2023.

[3] Paul E Black. Juliet 1.3 test suite: Changes from 1.2. Technical Report NIST TN 1995, National Institute of Standards and Technology, Gaithersburg, MD, June 2018.

[4] Jiahao Fan, Yi Li, Shaohua Wang, and Nguyen Nho Tien. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. *IEEE Access*, 8:197158–197172, 2020. doi: 10.1109/ACCESS.2021.3117124.

[5] Tiferet Gazit. Fixing Security Vulnerabilities with AI. `https://github.blog/engineering/platform-security/fixing-security-vulnerabilities-with-ai`, 2024. GitHub Engineering Blog, 14 Feb 2024.

[6] Kao Ge and Qing-Bang Han. Hidden code vulnerability detection: A study of the Graph-BiLSTM algorithm. *Information and Software Technology*, 175:107544, November 2024. ISSN 0950-5849. doi: 10.1016/j.infsof.2024.107544.

[7] Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. Interpreters for GNN-based vulnerability detection: Are we there yet? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, pages 1407–1419, 2023. doi: 10.1145/3597926.3598145.

[8] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. doi: 10.14722/ndss.2018.23158.

[9] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 738–742, Lisbon Portugal, April 2024. ACM. ISBN 979-8-4007-0587-8. doi: 10.1145/3643991.3644886.

[10] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. CrossVul: A cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1565–1569, New York, NY, USA, August 2021. Association for Computing Machinery. ISBN 978-1-4503-8562-6. doi: 10.1145/3468264.3473122.

[11] OpenAI. Connecting GitHub to ChatGPT deep research. `https://help.openai.com/en/articles/11145903-connecting-github-to-chatgpt-deep-research`, 2025. OpenAI Help Center, accessed 12 May 2025.

[12] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software, March 2019.

[13] Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. NYU CTF Bench: A Scalable Open-Source Benchmark Dataset for Evaluating LLMs in Offensive Security, February 2025.

[14] Ridwan Shariffdeen. Rshariffdeen/Big-Vul, October 2023.

[15] Mohammed Latif Siddiq and Joanna C. S. Santos. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22)*, pages 29–33, 2022. doi: 10.1145/3549035.3561184.

[16] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 472–483, New York, NY, USA, May 2024. Association for Computing Machinery. ISBN 979-8-4007-0502-1. doi: 10.1145/3639478.3647634.

[17] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A Large-Scale Security Patch Dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160, Taipei, Taiwan, June 2021. IEEE. ISBN 978-1-6654-3572-7. doi: 10.1109/DSN48987.2021.00030.

[18] Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection, April 2024.

[19] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, September 2019.

[20] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities, April 2025.

A Submitter Instructions

Table 5: Submitter Guidelines for VADER Benchmark Annotation and Evaluation

Category	Guidelines
Detection	• Is there a real and demonstrable vulnerability? • Can it be classified under CWE? • Avoid trivial or overly broad examples (e.g., `this whole file is vulnerable`).
Explanation	• Includes: root cause, how it can be exploited, and why it occurred. • Optional: link to real-world exploit or blog post.
Remediation	• Write a minimal, clean, and correct `golden_patch`. • Apply defensive programming and sanitize inputs. • Avoid large rewrites—modify as little as needed.
Checklist Before Submission	• Vulnerability is real, detectable, and high-impact. • `golden_patch` is minimal, clean, and secure. • Correct CWE/(OWASP optional) label used. • Test cases validate the fix.
Explanation Template (Min. 50 characters per field)	**Vulnerability Type**: CWE ID and short title. **Severity**: Use chart from Section 2. **Root Cause**: What specific flaw in the code leads to the issue? **Exploit Scenario**: How could an attacker exploit the flaw? **Why It Happens**: What in the system causes the issue? **Security Implications**: What could an attacker achieve? **Suggested Fix**: Summary of the fix or design change.

B Explanation Template Provided to Annotators

To ensure high-quality and consistent explanations across vulnerability cases, we provided annotators with a structured template. The template encouraged clear, technical descriptions of the issue, its cause, and mitigation strategies. An example is shown below.

Explanation Template
Vulnerability Type: CWE-89: SQL Injection
Severity: 4 (High)

Root Cause:
The code constructs a SQL query by directly concatenating user input without any sanitization or parameterization.

Exploit Scenario:
An attacker could supply input like `' OR 1=1-` to the `user_id` field, which would allow unauthorized access to all records in the database.

Why It Happens:
The application uses string formatting to dynamically build SQL queries, which makes it vulnerable to injection attacks.

Security Implications:
Exploitation could lead to full database compromise, including reading, modifying, or deleting sensitive data.

Suggested Fix:
Use parameterized queries (prepared statements) to separate query logic from user-provided values, eliminating injection risk.

The patch file is provided below.

C Comprehensive Reviewer Checklist

Table 6: Reviewer Rubric Guidelines for Annotated Case Validation

Review Area	Validation Criteria
Detection	• The vulnerability must be real and demonstrable. • It must be classifiable under a valid CWE category. • Avoid trivial or overly broad examples (e.g., `this whole file is vulnerable`).
Explanation	• Clearly describes the root cause of the vulnerability. • Explains how an attacker could exploit it. • Includes rationale for why the vulnerability occurs in this specific context. • (Optional) May include a link to a real-world exploit or blog post.
Remediation	• The patch (`golden_patch`) is minimal, clean, and technically correct. • Updates to function signatures or input sanitization are included if needed. • Defensive techniques (e.g., input bounds checks, least privilege) are preferred. • Avoids unnecessarily long rewrites—fixes should be scoped and concise.

(continued on next page)

(continued from previous page)

Review Area	Validation Criteria
Bonus Considerations (Optional)	• Case involves at least one common programming language (e.g., Python, JavaScript, C++). • Cross-language vulnerabilities (e.g., Python calling into unsafe C++) are especially valuable.
Final Submission Quality	• Vulnerability is high-impact, well-scoped, and grounded in real code. • `golden_patch` is clean, secure, and resolves the root cause. • Correct CWE label is assigned. • Test case or plan must validate the fix effectively. • Explanation includes all required fields and is technically sound.
Required Explanation Fields (Minimum 50 characters each)	• **Vulnerability Type:** CWE ID and short title. • **Severity:** Based on the standardized rubric. • **Root Cause:** Specific flaw and how it arises. • **Exploit Scenario:** Realistic attacker action and outcome. • **Why It Happens:** Deeper reasoning or design flaw. • **Security Implications:** Consequences or potential damage. • **Suggested Fix:** Short summary of how to resolve the issue.

D Data Analysis on Vulnerability Patterns

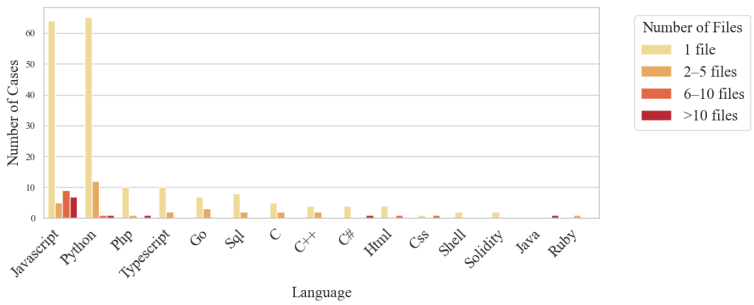
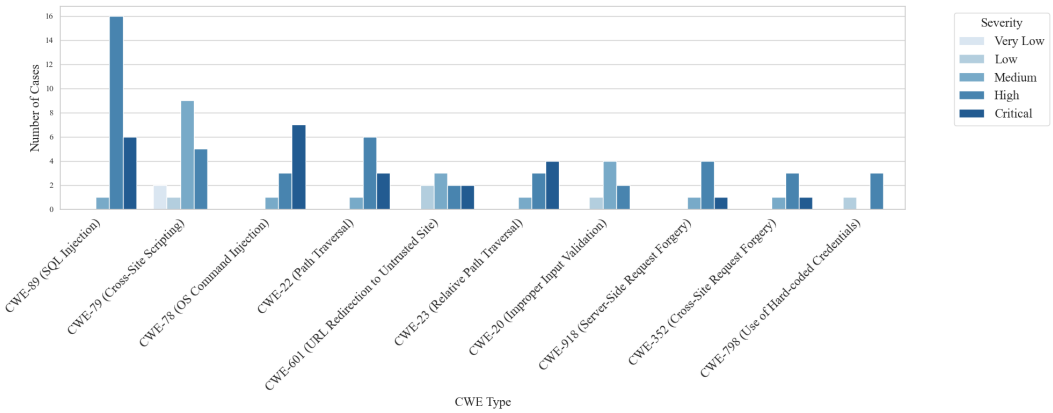


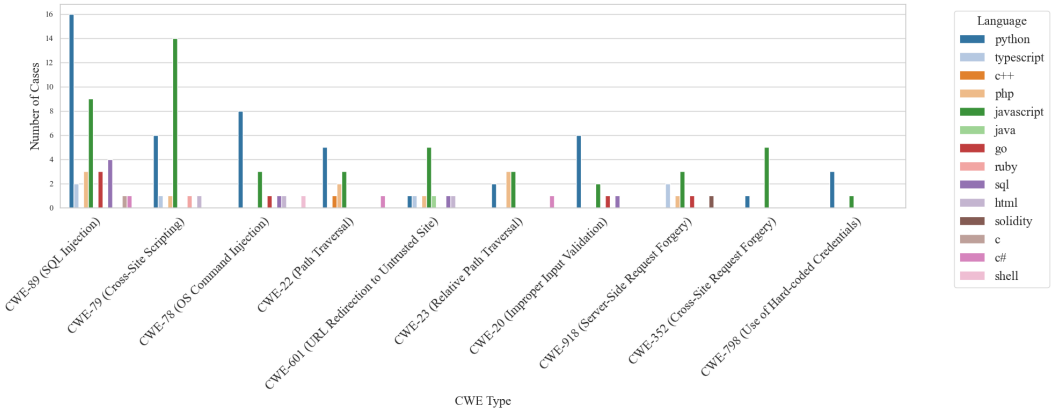
Figure 7: Distribution of languages, segmented by number of files.

Different languages are susceptible to different vulnerabilities and severities. Figure 8a presents a count plot of the top 10 Common Weakness Enumerations (CWEs), segmented by severity levels (Very Low to Critical), while Figure 8b displays the same CWEs, but segmented by programming languages. The most prevalent CWE-89 (SQL Injection) exhibits High to Critical severity as seen in Figure 8a, predominantly associated with Python (16 cases) and SQL (4 cases) as observed in Figure 8b, due to its capacity to manipulate database queries, allowing data theft in Python-based web frameworks like Flask or Django. The second most frequent, CWE-79 (Cross-Site Scripting), mostly Medium in severity in Figure 8a, is closely associated with Javascript (13 cases) and Python (6 cases) in Figure 8b, where injected scripts facilitate session hijacking or user redirection. CWE-78 (OS Command Injection), the third most frequent, is notable for its high proportion of Critical cases in Figure 8a, as it allows attackers to execute arbitrary OS commands with elevated privileges, resulting in severe impacts such as system compromise, data deletion (e.g., rm -rf /), or malware installation, with high occurencces in Python programs (8 cases) as seen in Figure 8b. CWE-22 (Path Traversal)

and CWE-20 (Improper Input Validation) are prevalent in Python applications, driven by Python's flexible file I/O and input handling, which, if not properly validated, can lead to unauthorized file access or broader exploits. CWE-601 (URL Redirection to Untrusted Site) and CWE-352 (Cross-Site Request Forgery) are predominantly associated with Javascript, where client-side scripting heightens susceptibility to unvalidated redirects and CSRF attacks.



(a) Distribution of Top 10 CWE Types by Severity



(b) Distribution of Top 10 CWE Types by Language

Figure 8: Combined analysis of CWE distribution, segmented by severity and language in VADER: (a) Distribution of CWEs, segmented by severity; (b) Distribution of CWEs, segmented by language.

E One-shot Prompt Template

Table 7: Exact prompt templates used for every evaluation run. All text inside is literal and case–sensitive.

Template	Literal string (line–breaks preserved)

System	`You are a cybersecurity engineer.`
	
	`For the very first case, you will receive:`
	`- A vulnerability description`
	`- The code files for inspection`
	`- The patch`
	`- Test case descriptions`
	
	`For all following cases, you will receive only the code files.`
	
	`For each case, produce exactly one JSON object (and nothing else) with these four keys:`
	`{`
	`  "cwe_id": "CWE-XXX",`
	`  "explanation": "A clear, concise technical explanation of the issue.",`
	`  "patch": "A Git-style patch that exactly matches the format of the patch example.",`
	`  "test_plan": ["step 1 description", "step 2 description", ...]`
	`}`
	
	`Requirements:`
	`1. Do NOT wrap the JSON in markdown or code fences.`
	`2. Do NOT include any extra keys or any commentary.`
	`3. Preserve all formatting inside the "explanation" and "patch" fields exactly as shown in the examples.`

Case prompt	`Case {case_id}:`
	`Files:`
	`{file_blocks}`
	
	`Please identify the vulnerability, explain it, propose a patch, and outline test steps to validate your fix.`
	`Your patch must be in the form of a GitHub-generated patch, as shown in the example patch.`
	`Respond in JSON with keys:  "cwe_id", "explanation", "patch", "test_plan".`

Example	`First case ({case_id}):`
	`Description:`
	`{description}`
	
	`Patch:`
	`{patch}`
	
	`Test Cases:`
	`{tests}`