CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities

Peiran Wang[1], Xiaogeng Liu[2], Chaowei Xiao[2],
[1]Tsinghua University, [2]University of Wisconsin–Madison,

Abstract

Automated vulnerability repair is a crucial field within software engineering and security research. Large Language Models (LLMs) and LLM agents have demonstrated significant potential in this domain by understanding descriptions in natural language and generating corresponding formal code. Although the coding capabilities of LLMs have advanced rapidly, evaluation benchmarks for real-world programming setups are still lagging, preventing the development of LLM and LLM agents in real-world vulnerability repair. To this end, we introduce `CVE-Bench`, an evaluation framework consisting of 509 Common Vulnerabilities and Exposures (CVEs) from four programming languages and 120 popular open-source repositories. Unlike previous vulnerability repair benchmarks, which only involve the code input and output, we provide LLM agents with a test environment that simulates the real-world vulnerability repair process. This environment provides multiple levels of CVE information modeling, such as black-box testing and white-box testing. It enables the agents to use static analysis tools to assist their repair process. Our evaluation reveals that the SWE-agent can only repair 21% of vulnerabilities at its best. Furthermore, they lack expert knowledge about how to use the analysis tool to assist in vulnerability repair.

1 Introduction

Repairing vulnerability is an essential task due to the possible significant loss brought by the vulnerability. For instance, the outbreaks of WannaCry (CERT, 2017) and NotPetya (Aidan et al., 2017) caused a 180 billion loss in 2017, and the Equifax (Zou et al., 2018) data breach exposed the sensitive data of more than 143 million consumers. The computer security community follows a "report and repair" workflow to deal with the vulnerability (Mu et al., 2018). Experts can report the existence of vulnerabilities by black-box or white-box penetration testing (Verma et al., 2017). The black-box testers can only access the compiled or deployed program, so they can only report vague information on how to exploit the vulnerability. Meanwhile, the white-box testers can report rich information about where to fix the problem because they can access the source code during the mining. After that, the security expert or developers can manually repair the vulnerability based on the reported information. However, repairing software vulnerabilities based on the above information often requires significant input from human experts, who must navigate complex code bases and understand intricate inter-dependencies within the software. This process is inefficient and labor-intensive, especially when dealing with ambiguous or sparse information from sources like black-box penetration testing. As a result, the task consumes substantial human resources, highlighting the motivation to design an automated solution.

On the other hand, Large language models (LLMs) agents have been used for software development (Chen et al., 2021; Austin et al., 2021; Jain et al., 2022; Nijkamp et al., 2022; Li et al., 2022a) to reduce extensive the developer labor-intensive. To speed up the development of LLM agents for software development, diverse benchmarks have been proposed on benchmarking general code generation ability for software developments (Chen et al., 2021; Jimenez et al., 2023; Liu et al., 2024; Ji et al., 2023), leaving the benchmarking for vulnerability vastly less explored. Despite there are some benchmarks (Jimenez et al., 2023; Bhandari et al., 2021; Wang et al., 2024; Bui et al., 2022). for vulnerability fixing, these benchmarks mainly support the code blocks as the input (Bhandari et al., 2021) or provide a general bug-repairing evaluation (Jimenez et al., 2023), without insight into the inherent rationale of the LLM-based agent execution process (e.g., interactive features with code

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 4207–4224

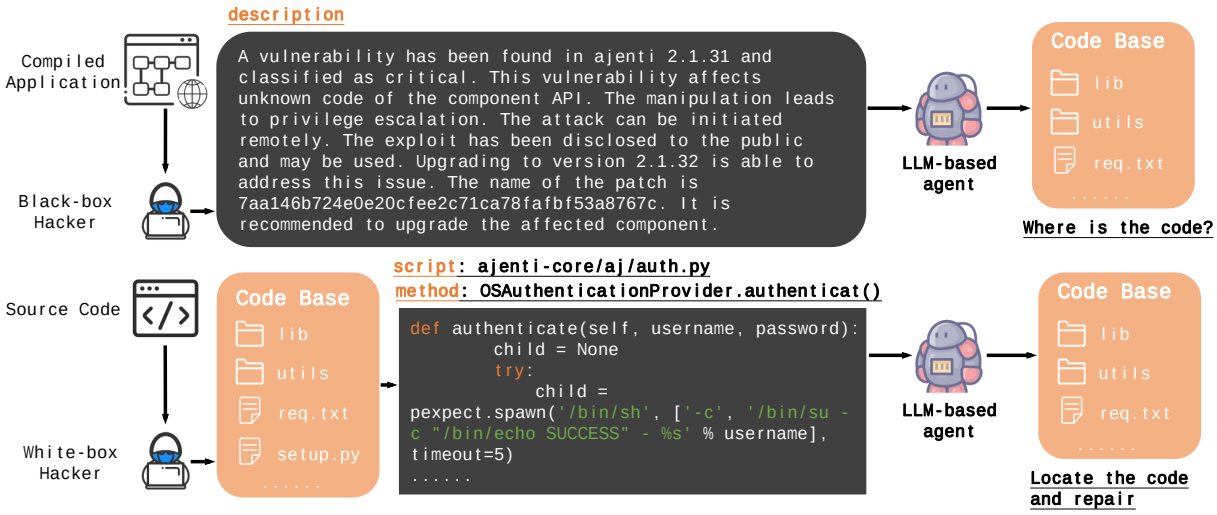


Figure 1: An example of black-box repairing and white-box repairing for LLM-based agents. We use the agent's CVE-2019-25066 as an example: (1) Black-box hackers can only access the compiled application and realize the exploitation chain. His reported information about the repository's vulnerability issues may only include how to exploit it but not where to repair it. (2) White-box hackers can access the source code. Thus, their reported information will include the direct script path and the corresponding code method. Thus, the LLM agents may face multiple levels of information when required to repair the vulnerability issues.

interpreter, tool-using features).

In this paper, we introduce CVE-Bench (§2), a benchmark that evaluates LLM-based agents in a realistic vulnerability-repairing setting. CVE-Bench contains three unique characteristics:

(1) Instead of input-output evaluation, CVE-Bench supports agent-based evaluation by offering real-world interactive execution-guided programming environments. Such an environment can provide feedback to the agent using the interpreter to fix vulnerabilities.

(2) CVE-Bench supports tool-using. In real-world vulnerability repair settings, developers can use diverse static or dynamic analysis tools to assist the vulnerability repair process. For instance, given vague information about a vulnerability's exploit process, developers can use the analysis tool to assist the repair process (e.g., scan and locate the vulnerability code block position and debug the repaired code).

(3) CVE-Bench simulates the vulnerability-repairing process in the real world by providing multiple-level information consideration that models white-box and black-box CVE reports. For real-world vulnerability fixing, different levels of information may provided by either the white-box or black-box analyzers (as shown in Figure 1). White-box analyzers can access the source code, locate the problematic code blocks, and report them to the developers. Thus, the developers can directly locate the vulnerability code and repair it. However, black-box analyzers can only access the compiled or applied programs and exploit the vulnerability but do not know the actual location of the code blocks. Thus, developers can only fix the vulnerability with just simple instructions.

Specifically, CVE-Bench provides 509 CVE samples across four programming languages (Python, Java, JavaScript, and PHP). The selected CVEs come from 120 open-source GitHub repositories, with at least 5000 stars for each repository. Over 16 types of CWEs are included in the collected CVEs. You can see more details about the selected CVEs in §A. Furthermore, we provide four static analysis tools (Prospector, Bandit, Pylint, and Mypy) in our interactive environment to assist the repair process.

We evaluate our benchmark with SWE-agent (Yang et al., 2024). We find that the SWE agent can achieve the repair with a success rate of about 21% at its best. Furthermore, we observe that the tested agent can only solve 14% of tasks under the black-box-level information setting. Further observation also uncovers that the agents lack the knowledge to use the analysis tool. This underscores the need to build more interactive LLM-based agents and enhance the agents' ability to use tools.

2 CVE-Bench

In this section, we discuss the overall design of the CVE-Bench. We first discussed constructing the

whole benchmark in §2.1. Then, we formulate the task for the CVE-Bench in §2.2. At last, we list the features of CVE-Bench that make it different from previous works and contribute to future research in §2.3.

2.1 Benchmark Construction

The construction process of CVE-Bench follows the three steps (see Figure 2):

Step I: Environment construction. CVE-Bench starts by selecting target CVEs for the benchmark. We explore the CVEFixes database (Bhandari et al., 2021) and select 509 CVEs from 4 programming languages, with the corresponding repositories' star counts over 5,000 (see details in §A). Then, we query three types of information, including the CVE description, the CVE script, and the CVE method/function, and use the three types of information to construct three levels of information to model the black-box (description), white-box (description + script + method) reports, and an intermediate level between the two (description + script). After the information is constructed, CVE-Bench builds the code base for the evaluation. CVE-Bench first clones the CVEs' corresponding repositories, then queries the database for the CVEs' repair belonging commits and checkouts the repository to the parent commits.

Step II: Repair patch generation. Next, CVE-Bench sends the generated information to the agent. The agent builds a docker environment for the constructed code base when receiving the input information. Then, the agent performs an iterative repair process: (1) The LLM generates the instruction to be executed; (2) The agent inputs the instruction on the terminal using the docker; (3) The terminal execute and output feedback; (4) The feedback is sent back to the LLM to repeat (1)-(4). During the patch generation process, CVE-Bench enhances the executable environment by providing four static analysis tools (Prospector, Pylint, Bandit, and Mypy) for the agents to call to simulate the real-world vulnerability repair environment (see detail in §C).

Step III: Repair validation. At last, CVE-Bench performs execution-based repair patch validation using the unit tests. We crafted unit tests (including the Dockerfile to construct the environment for the unit test and the unit test scripts) for each selected CVE. The unit test is written following the exploit chain of each CVE. When the repair patch is generated, CVE-Bench will apply the patch to the code base and execute the unit test to judge whether the CVE is repaired.

2.2 Task Formulation

Model input. First of all, CVE-Bench extracts three types of information from the CVEFixes database (Bhandari et al., 2021): (1) Description: This information describes the rationale and the exploit logic of the CVE, described by natural language; (2) Script: This describes which script contains vulnerability issues, along with the script's path from the code base; (3) Method: This information detailedly describes which method triggers the vulnerability issues.

Based on these three types of information, CVE-Bench further constructs three levels of information to model real-world vulnerability report information: (1) Black box: This level only contains the description information, modeling black-box vulnerability report; (2) White box: This level contains all the description, script, and method information to model the white-box vulnerability report. See a detailed example in Figure 2. Additionally, we also provide the Intermediate category: This level contains description and script information, modeling an intermediate state between the black box and white box;

Evaluation metrics. We apply the generated patch to the code base using GitHub's patch program to validate the generation and execute the unit tests associated with the target vulnerabilities. If the patch applies successfully and the unit test passes, we consider the generated patch to have successfully repaired the vulnerability. The metric for our benchmark is the percentage of resolved vulnerabilities (repair rate).

2.3 Features of CVE-Bench

Traditional benchmarks in NLP typically involve only short input and output sequences and consider somewhat "contrived" problems explicitly created for the benchmark. In contrast, CVE-Bench 's realistic construction setting imbues the dataset with unique properties, which we discuss below.

Real-world CVE vulnerability repair tasks. Since each task instance in CVE-Bench consists of a large and comply CVE-Bench requires demonstrating sophisticated skills and knowledge possessed by experienced software engineers but are

Figure 2: There are four steps in CVE-Bench: (1) Environment construction: CVE-Bench uses the three-level information to generate a vulnerability issue as the input to the agent. CVE-Bench clones the CVEs' corresponding repositories and checks out the repository to the parent commits. (2) Repair patch generation: Next, CVE-Bench sends the queried information divided into multiple levels to the agents for patch generation. (3) Repair validation: At last, CVE-Bench performs execution-based repair patch validation using the unit tests. CVE-Bench also considers comparing the generated patches with the ground truth repair code crafted by the man.

not commonly evaluated in traditional code generation benchmarks.

Real-world CVE vulnerability repair process modeling. Unlike previous benchmarks, which only provide one level of information to evaluate LLMs, CVE-Bench provides multiple levels of information as input. This approach analyzes what the LLM-based agent needs to repair a vulnerability issue. By doing so, we can dive deep into an observation about enhancing the agents' ability. This approach also aligns with the real-world vulnerability repair settings (low information input-black box, rich information input-white box).

Interactive repair environment simulation. Another feature of the LLM-based agent is its ability to use tools. Previous research only evaluates an agent's ability to move from input to output, treating agents as black boxes without concern and observing their inside functionality. Unlike them, CVE-Bench encompasses the analysis tool using the vulnerability issues repair approach. We evaluate whether the agent chooses and uses the tool effectively.

Execution-based validation. Unlike previous vulnerability repairing benchmarks, which only use the sequence accuracy as the repair metrics, CVE-Bench uses the unit test pass portion as the repair portion of the CVEs. This is a more reliable and robust evaluation metric since the LLMs may generate diverse but equally effective repair patches.

3 Evaluation

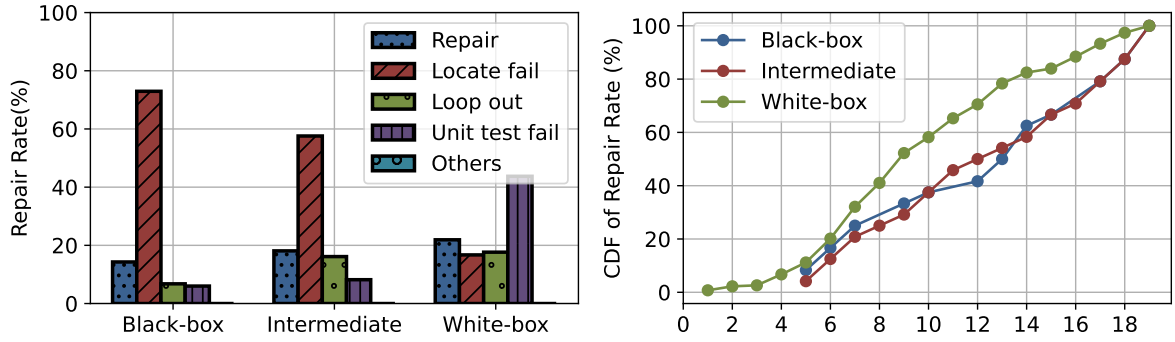
We calculated the unit test pass rate as the patch repair rate for the evaluation metrics. Furthermore, we also categorized the failed reasons as follows:

- Locate fail: LLM can not locate the vulnerability codes.
- Loop out: The repairing process reaches the maximum step limit.
- Unit test fail: The generated repair patch does not pass the unit test.
- Others: Other reasons for the agents' generation failure (e.g., failure to generate formal command).

3.1 Different Level Information

Method. We compare the SWE-agent's ability to repair the vulnerability across three information levels (Black-box, intermediate, and white-box). We evaluate the agents on the selected Python CVEs, with the GPT-4 as the foundation model backend. We summarized the patch repair rate using the validation and counted the reasons for failure. We also counted the fixed rate's CDF as the interaction step increased.

Results. As shown in Figure 3 (a), we can observe that the repair rate under the white-box setting is better than the black-box setting. When under the black-box setting, the repair rate is only



(a) Repair rate & Failure reason

(b) CDF of repair rate with the interaction step increase

Figure 3: We evaluated the tested agents using three different levels of information: CVE description, CVE script, and CVE method to model the real-world vulnerability fixing (black box and white box). (a) We summarized the repair rate and the three reasons for failure at each level. (b) We counted the CDF of the fixed rate among the three levels of information with the interaction step increase.

about 14.1%. While under the intermediate setting, the repair rate reaches about 17.8%. The white-box setting gives the agent the best ability to repair the vulnerability issues, with a repair rate of over 21%.

Then, we studied the failure reasons across the three levels of information in Figure 3 (a). Under the black-box setting, the most common reason for failure commands is that the agent can not locate the vulnerability. This aligns with our intuition: the agent cannot independently locate the detailed location (script and method). The same observation also occurs in the intermediate setting, indicating that the agent can not locate where the vulnerability is even when the script path is given. We checked the data for the white-box setting and found that the main reason for failure was failing to pass the unit test. This indicates that the white box can always craft the repair patch but not a stable one.

For the success repair, we can observe more details in Figure 3 (b). For the white-box setting, the agent can quickly fix the vulnerability within low steps because it knows the actual position of the code blocks. The other two levels (black box and intermediate) require more steps to accomplish the tasks because the agent needs to find the location of the vulnerability code blocks in the first few steps.

3.2 Programming Languages

Method. We compare the performance of the SWE-agent when solving vulnerability repair tasks for different programming languages (Python, Java, JavaScript, and PHP). For this test, we only consider the GPT-4 foundation model backend and the information at the white-box level as input.

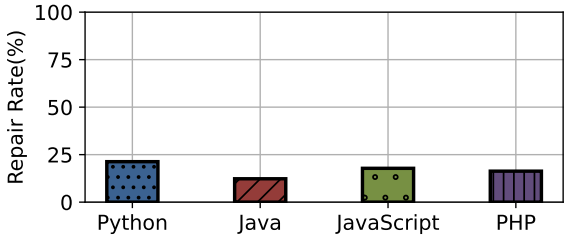
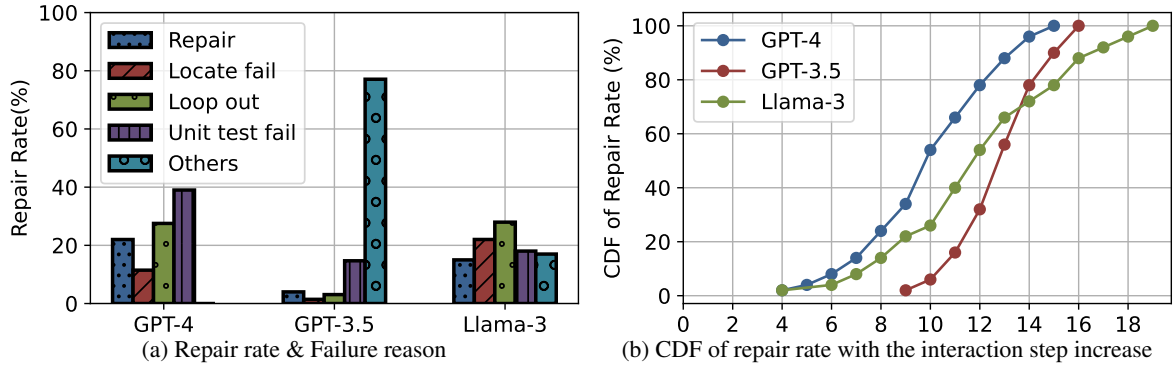


Figure 4: We evaluate the tested agents across four programming languages: 1)Python, 2)Java, 3)JavaScript, and 4)PHP.

Info. Level	Python	Java	JavaScript	PHP
Repair #	15	53	127	38
Repair %	21.17	12.33	17.81	16.33
Edit line	74.00	34.90	43.23	34.34
Edit word	3206.00	1735.98	1606.73	2010.42

Table 1: We evaluate the tested agents across four programming languages: 1)Python, 2)Java, 3)JavaScript, and 4)PHP.

Results. The results are shown in Figure 4. The SWE-agent performs best when repairing the Python CVEs with the highest repair rate of 21.17%. The repair rates on JavaScript (17.81%) and PHP (16.33%) are nearly the same and fall about 5% behind the repair rate on Python. However, the SWE-agent performs worst when solving Java CVEs with a repair rate of around 12%. This may be due to the GPT-4's training dataset, which may contain more knowledge of Python than Java.



(a) Repair rate & Failure reason

(b) CDF of repair rate with the interaction step increase

Figure 5: We evaluate the tested agents across three different foundation models: 1)GPT-4, 2)GPT-3.5 and 3)Llama-3.

Furthermore, the syntax complexity of Java is also higher than the complexity of the other three languages.

3.3 Different Foundation Models: GPT-4 vs. GPT-3.5 vs. Llama-3

Method. We compare the agent's ability to repair the vulnerability under the GPT-4, GPT-3.5, and the Llama models. Due to the high cost, we set the information level to the white-box level, with 50 Python CVE samples as the test set.

Results. Figure 5(a) lists the patch repair portion. We can observe that the repair portion of GPT-4 is the highest. While using GPT-3.5 and Llama, the repaired portion drops quickly.

Then, we analyzed the failure reasons for the three different foundation models. As shown in Figure 5(a), for GPT-4, the most common failure reason for failed commands is unit test failure. The most common reason for failure in GPT-3.5 is the failure to generate formal commands (including in others). This is because GPT-3.5 can not generate formal commands to execute on the terminal. It always falls short of generating unuseful or syntax error commands.

3.4 Tool Using

Method. We compare the performance of the SWE-agent when using different static analysis tools and when not using tools. We only considered Python CVEs and used 50 testing samples to simplify the experiment. The information level is set to the black-box setting. We consider four distinct tools: Pylint, Bandit, Mypy, and Prospector (see details in §C). We compare the repair rate and the repair CDF changes across the four tools used.

Results. As shown in Figure 6 (a), the tool used does not enormously increase the repair portion; some tools even make the repair rate drop compared with non-tool use (Default). This may be attributed to the fact that the LLM does not have enough knowledge to use these tools. Also, using the tool requires more loops of interactions, which may cause the agents to reach the maximum limit of the loops.

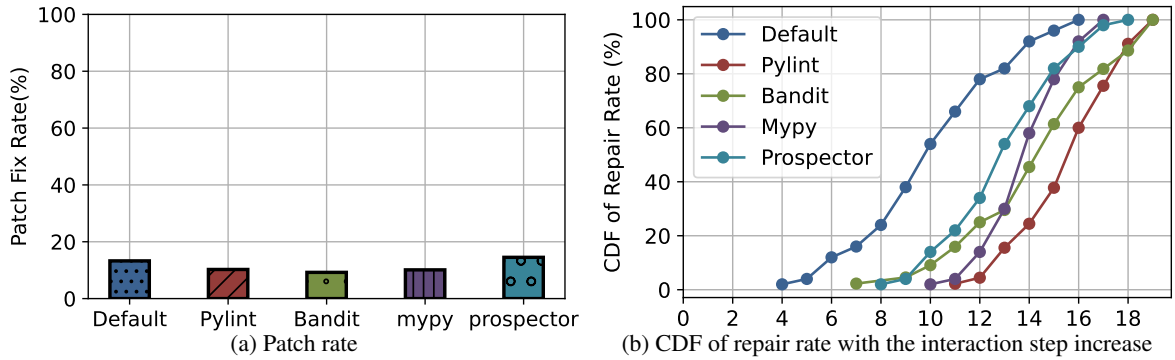
Furthermore, we observe in Figure 6(b) the tools require the agents to take more steps to repair a vulnerability. However, we believe the agent failed to use the tool effectively because the repair rate did not increase greatly.

3.5 Evaluation for Overfitting

Method. Considering the potential overfitting issues for the GPT-4 backend for the tested agents, we compare the repair rate between the repair cases on `CVE-Bench`'s pre-collected CVEs and newly collected CVEs after 2023 Nov (the end of GPT-4 training). We collected 50 Python CVEs after 2023 Nov to avoid the potential training sample involved. Then, we used the `CVE-Bench` under GPT-4 backend, with the information level set to white-box setting, to test the agents on the 50 Python CVEs.

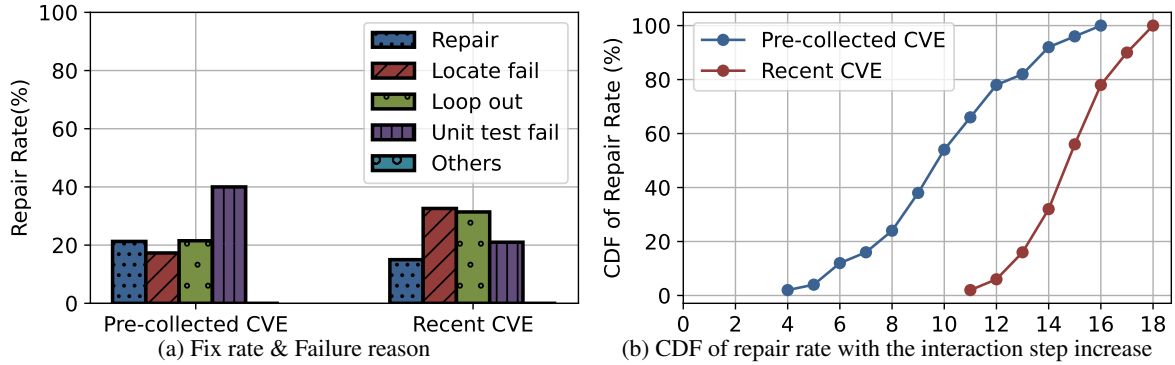
Results. As shown in Figure 7 (a), the repair rate on recent CVEs is lower than that of pre-collected CVEs. This indicated that the pre-trained samples strongly affected the agents' repair ability. If the related knowledge persists in the agents' backend foundation models' training samples, it may be easier for them to locate and repair the vulnerability issues.

The CDF in Figure 7 (b) further reveals the overfitting effect. For the pre-collected CVE, the agent mainly takes less than 12 steps to accomplish the repair tasks. Meanwhile, for the recent CVE, the agent must take more steps to accomplish the tasks.



(a) Patch rate (b) CDF of repair rate with the interaction step increase

Figure 6: We evaluate the tested agents across four different static analysis tool support: Pylint, Bandit, mypy, and Prospector. We compare the repair rate and the CDF of repair between steps.



(a) Fix rate & Failure reason (b) CDF of repair rate with the interaction step increase

Figure 7: We compare the repair rate between the repair cases on CVE-Bench's pre-collected CVEs and newly collected CVEs after 2023 Nov (the end of GPT-4 training) to avoid the effect of overfitting.

3.6 A quantitative analysis

We provide an example from CVE-2022-0577, a Python CVE from the Scrapy repository (see Figure 8). The CVE's description is "Exposure of Sensitive Information to an Unauthorized Actor in GitHub repository scrapy/scrapy."

In this example, the agents are given the 3-level information contained in the generated issue. The agent adds four lines of code in the agent-generated code blocks to check whether the header is from the two sensitive headers: "Cookie" and "Authorization," to repair the potential vulnerability. However, as we observe in Figure 8's ground truth repair code, the developers added another function to check the header detailedly. Furthermore, the developers call another function from the other script to solve it within the created function.

This highlights current LLM-based agents' drawbacks: First, it tends to repair the code directly in its original location. Compared to code written by human developers, it is not concise enough. It lacks maintainability (adding a lot of code in place is more likely to make code development more confusing than creating a specific function). Second, the LLM agents still lack enough domain to call the existing function to solve the problem. Still, they aim to fix the problem using their own generated code, which is inconsistent with the modular development characteristics of software engineering.

4 Related Work

Code generation benchmark. HumanEval (Chen et al., 2021) has become the benchmark for the enduring challenge of generating code from natural language specifications (Yu et al., 2018; Austin et al., 2021; Hendrycks et al., 2021a,b; Zan et al., 2023). Over the past year, various benchmarks have aimed to enhance HumanEval by incorporating support for multiple programming languages (Cassano et al., 2022; Athiwaratkun et al., 2023; Orlanski et al., 2023), diverse editing scopes (Yu et al., 2023; Du et al., 2023). Innovative code completion challenges (Muennighoff et al., 2023), along with an increased emphasis on testing (Liu et al., 2023b,a). In parallel, other studies have endeavored to introduce innovative coding methodologies (Yin

Figure 8: We show an example of a formatted task instance, the original code, ground-truth repair code (extracted from the database, made by the developers), the 3-level information, generated issue, and the agent-generated code. In the code blocks, grey highlights are additions.

et al., 2022; Yang et al., 2023) or to develop problems tailored to specific libraries (Lai et al., 2022; Zan et al., 2022). Instead of general code generation problems, CVE-Bench focuses on vulnerability repair problems. Vulnerability repair requires the agents to understand the vulnerability exploit rationale from the attackers' view. Based on such understanding, the agents can generate high-quality repair patches. Furthermore, real-world vulnerability repair involves black-box testing reports, which only provide vague information for the agents.

ML for software engineering. Machine Learning (ML) is revolutionizing Software Engineering by automating software development processes using neural networks and Large Models (LMs) (Zheng et al., 2023; Hou et al., 2023). This includes automating commit messages (Jung, 2021; Liu et al., 2023a), enhancing PR reviews (Yang et al., 2016; Li et al., 2022b), bug localization (Kim et al., 2019; Chakraborty et al., 2018), software testing (Kang et al., 2023; Xia et al., 2023), and program repair (Gupta et al., 2017; Allamanis et al., 2017). For CVE-Bench, relevant works focus on applying LMs to automated program repair (Xia and Zhang, 2022; Fan et al., 2023) and guiding code editing through commits (Chakraborty et al., 2018; Zhang et al., 2022). However, the previous fixing mainly follows the one-time input-to-output paradigm and ignores the interactive nature of the LLM-based agents. CVE-Bench simulates a real-world environment for the program repair by providing real-world-level information and tool use.

5 Conclusion

Automated vulnerability repair is a popular and valuable software engineering and security research field. Large language models (LLMs) and LLM-based agents have shown sizeable potential application value in this area. LLMs can understand natural language described vulnerability rationale and generate formal code to repair it. However, evaluation benchmarks modeling real-world vulnerability repair environments are still lacking. To this end, we introduce CVE-Bench, an evaluation framework consisting of 509 CVEs from four programming languages and 120 popular open-source repositories. Unlike previous vulnerability repair benchmarks, which only involve the code input and output, we provide LLM agents with a real-world vulnerability repair environment. In this environment, multiple levels of CVE information modeling, such as black-box and white-box testing, are provided to rate agents' ability to repair vulnerabilities. Furthermore, we enable the agents to use static analysis tools in this environment to assist their repair process and see how the repair rate changes with the tool's use. Our evaluation shows that the agent's ability differs greatly from white to black-box settings. And the agent can still not use the static analysis tool effectively.

6 Limitation

Limited programming language coverage. Because there are too many programming languages, our benchmark cannot cover too many types. At the same time, due to the high complexity of some programming languages, it is difficult for us to build a more robust unit test set.

Limited tested agent types. Since there are relatively few existing agent types, we only conducted experiments on SWE-Agent (Yang et al., 2024). Furthermore, Open-Devin (OpenDevin Contributors, 2024) does not provide a sufficient API for us to call. It only supports using its API, which limits `CVE-Bench` to establish large-scale experiments.

Limited tested foundation models. Until our paper is submitted, we have only performed the experiments on the GPT -4, GPT-3.5, and Llama-3 models. We plan to add more experiments before the publication.

References

Jagmeet Singh Aidan, Harsh Kumar Verma, and Lalit Kumar Awasthi. 2017. Comprehensive survey on petya ransomware attack. In *2017 International conference on next generation computing and information systems (ICNGCIS)*, pages 122–125. IEEE.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, and Yuchen Tian et. al. 2023. Multi-lingual evaluation of code generation models. *Preprint*, arXiv:2210.14868.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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 (PROMISE '21)*, page 10. ACM.

Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 464–468, New York, NY, USA. Association for Computing Machinery.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

US CERT. 2017. Indicators associated with wannacry ransomware.

Saikat Chakraborty, Yujian Li, Matt Irvine, Ripon Saha, and Baishakhi Ray. 2018. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, and Jared Kaplan et. al. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. *Preprint*, arXiv:2205.10583.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021a. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021b. Measuring coding challenge competence with apps. *NeurIPS*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *Preprint*, arXiv:2308.10620.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231.

Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and explaining large language model-based code generation: A causality-centric approach. *arXiv preprint arXiv:2310.06680*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *Preprint*, arXiv:2105.14242.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. *Preprint*, arXiv:2209.11515.

Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–34.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

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

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022b. Automating code review activities by large-scale pre-training. *Preprint*, arXiv:2203.09095.

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.

Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2023a. Commitbart: A large pre-trained model for github commits. *Preprint*, arXiv:2208.08100.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, and Hanyu Lai et. al. 2023b. Agentbench: Evaluating llms as agents. *Preprint*, arXiv:2308.03688.

Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *Preprint*, arXiv:2308.07124.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenDevin Contributors. 2024. Opendevin: Code less, make more. `https://github.com/OpenDevin/OpenDevin`.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. *Preprint*, arXiv:2302.01973.

Akanksha Verma, Amita Khatana, and Sarika Chaudhary. 2017. A comparative study of black box testing and white box testing. *International Journal of Computer Sciences and Engineering*, 5(12):301–304.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. 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, page 472–483, New York, NY, USA. Association for Computing Machinery.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748*.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Preprint*, arXiv:2306.14898.

Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the modern code review repositories: A dataset of people, process and product. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 460–463.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. 2022. Natural language to code generation in interactive data science notebooks. *Preprint*, arXiv:2212.09248.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *Preprint*, arXiv:2302.00288.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. *Preprint*, arXiv:2206.06888.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. *Preprint*, arXiv:2212.09420.

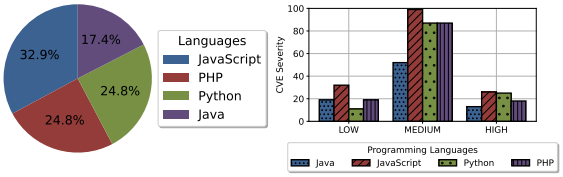
Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. *Preprint*, arXiv:2208.05446.

Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396*.

Yixin Zou, Abraham H Mhaidli, Austin McCall, and Florian Schaub. 2018. " i've got nothing to lose": Consumers' risk perceptions and protective actions after the equifax data breach. In *Fourteenth Symposium on Usable Privacy and security (soups 2018)*, pages 197–216.

A Dataset Details

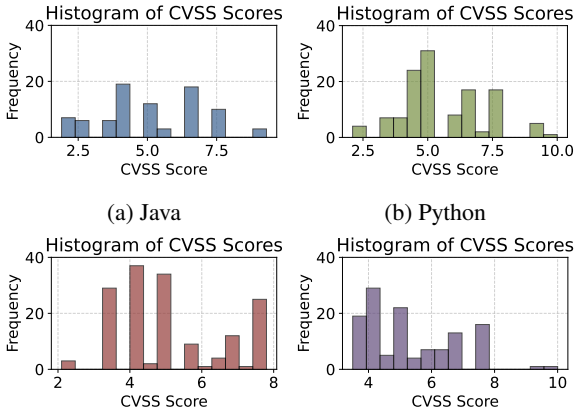
In this section, we discussed the details of the CVE dataset.



(a) Programming Languages (b) CVE Severity

Figure 9: The programming languages and the CVE severity of the selected CVEs.

Programming languages. The CVE dataset contains 509 vulnerability issues selected from four programming languages (Java, Python, JavaScript, and PHP). The portion of each programming language is shown in Figure 9(a) (JavaScript: 164, PHP: 124, Python: 124, Java: 87).



(a) Java (b) Python

(c) JavaScript (d) PHP

Figure 10: The distribution of the CVSS scores across the four programming languages CVEs.

CVE severity&CVE score. The severity of CVEs (Common Vulnerabilities and Exposures) is classified using the CVSS (Common Vulnerability Scoring System). The CVSS provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes. Here are the general severity ratings based on the CVSS score:

1. None: 0.0
2. Low: 0.1 - 3.9
3. Medium: 4.0 - 6.9
4. High: 7.0 - 8.9
5. Critical: 9.0 - 10.0

The CVSS score is calculated based on several metrics, which fall into three main groups:

- **Base metrics**: These represent a vulnerability's intrinsic and fundamental characteristics that are constant over time and across user environments. Base metrics include the attack vector, attack complexity, privileges required, user interaction, scope, confidentiality, integrity, and availability impacts.
- **Temporal metrics**: These reflect the characteristics of a vulnerability that may change over time but not across user environments. Temporal metrics include the exploit code maturity, remediation level, and report confidence.
- **Environmental metrics**: These represent the characteristics of a vulnerability that are relevant and specific to a particular user's environment. Environmental metrics include collateral damage potential, target distribution, security requirements, and modified versions of the base metrics.

We summarized the collected CVEs severity distribution in Figure 9(b) and the CVSS score distribution in Figure 10.

The overall CVSS score combines these metrics to provide a standardized view of a vulnerability's severity.

CWE type of the selected CVEs. The Common Weakness Enumeration (CWE) is a community-developed list of common software and hardware security weaknesses. Maintained by the MITRE Corporation, CWE aims to serve as a standard reference point for identifying, mitigating, and preventing weaknesses in software and systems throughout their lifecycle. By providing a unified language for discussing software security issues, CWE helps improve the quality of software and hardware products by reducing the prevalence of vulnerabilities.

CWE uses a hierarchical classification system to categorize vulnerabilities based on their nature, cause, and impact. This classification system is

CWE name	description	count
Cross-site Scripting	The software does not neutralize or incorrectly neutralize user-controllable input before it is placed in output used as a web page served to other users.	103
Insufficient Information	There is insufficient information about the issue to classify it; details are unknown or unspecified.	28
Path Traversal	The software uses external input to construct a pathname intended to identify a file or directory located underneath a restricted parent directory. Still, the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location outside the restricted directory.	28
Exposure of Sensitive Information to an Unauthorized Actor	The product exposes sensitive information to an actor who is not authorized to access that information.	23
SR	The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.	22
Improper Input Validation	The product receives input or data, but it does not validate or incorrectly validate that the input has the properties that are required to process the data safely and correctly.	21
Incorrect Authorization	The software performs an authorization check when an actor attempts to access a resource or perform an action, but it does not correctly perform the check. This allows attackers to bypass intended access restrictions.	13
SR	The web server receives a URL or similar request from an upstream component and retrieves the contents of this URL, but it does not sufficiently ensure that the request is being sent to the expected destination.	13
Open Redirect	A web application accepts a user-controlled input that specifies a link to an external site and uses that link in a Redirect. This simplifies phishing attacks.	12
SQL Injection	The software constructs all or part of an SQL command using externally-influenced input from an upstream component. Still, it does not neutralize or incorrectly neutralize special elements that could modify the intended SQL command when it is sent to a downstream component.	12
Uncontrolled Resource Consumption	The software does not properly control the allocation and maintenance of a limited resource, thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.	12
Injection	The software constructs all or part of a command, data structure, or record using externally influenced input from an upstream component. Still, it does not neutralize or incorrectly neutralize special elements that could modify how it is parsed or interpreted when sent to a downstream component.	12
Deserialization of Untrusted Data	The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.	11
Other	NVD is only using a subset of CWE for mapping instead of the entire CWE, and the weakness type is not covered by that subset.	10
Improper Authentication	When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct.	10
Command Injection	The software constructs all or part of a command using externally influenced input from an upstream component. Still, it does not neutralize or incorrectly neutralize special elements that could modify the intended command when sent to a downstream component.	10

Table 2: The selected CWE name, description, and the count of the selected CVEs.

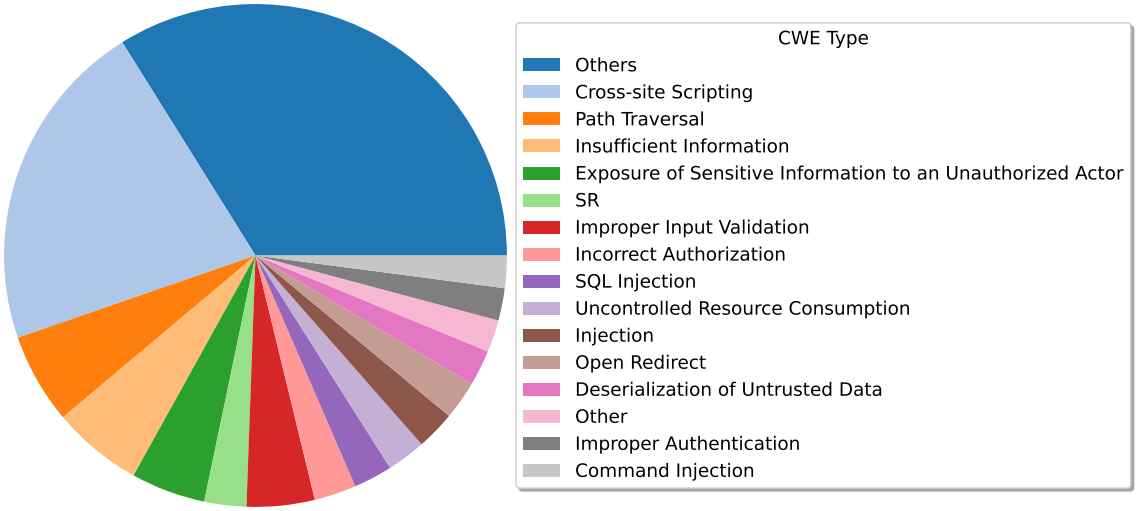


Figure 11: The CWE type that the CVEs belong to.

structured into several levels, making it easier for developers, security professionals, and researchers to understand and address security weaknesses. Here are the main components of the CWE classification system:

1. **Weaknesses (Base-Level)**: The most detailed classification level, each base-level CWE entry describes a specific type of vulnerability. Examples include buffer overflows, SQL injection, and cross-site scripting (XSS).

2. **Categories**: Category group-related weaknesses based on certain attributes or effects. They provide a higher-level view that helps understand commonalities between different weaknesses. For instance, "Resource Management Errors" is a category that includes various weaknesses related to resource handling issues.

3. **Views**: Views are specific perspectives or lenses through which the CWE list can be examined. They are designed to meet the needs of different stakeholders or purposes. Examples include the "Development View," which focuses on weaknesses from a software development perspective, and the "Research Concepts View," which categorizes weaknesses based on their conceptual relationships.

4. **Compound Elements**: These include chains and composites, which describe combinations of multiple weaknesses. Chains represent sequences of weaknesses that lead to an exploitable condition, while composites describe complex weaknesses best understood as aggregations of simpler ones.

5. **Root Causes**: Some CWE entries classify vulnerabilities by their root causes. Understanding the underlying cause of a weakness can help develop more effective mitigation strategies. Root causes might include insufficient input validation, improper resource management, or inadequate error handling.

6. **Modes of Introduction**: CWE also categorizes vulnerabilities based on how they are introduced into the software. This includes phases of the software development lifecycle, such as design, implementation, and deployment. Understanding when and how a weakness is introduced can help in preventing similar issues in the future.

The CWE classification system provides a structured way to analyze and address vulnerabilities, making it an invaluable tool for improving software security practices across the industry. We summarized the CWE type of our collected CVEs in Figure 11 and Table 2.

Repository of the selected CVEs. Our selected repositories of the selected CVEs are listed in Table

3. Our selection on the repositories follow the rule that their stars should be over at least 5000.

B Patch Generation Details

Database extraction. We built a database by following (Bhandari et al., 2021)'s CVE collection process. This comprehensive vulnerability dataset is automatically collected and curated from Common Vulnerabilities and Exposures (CVE) records in the public U.S. National Vulnerability Database (NVD). The goal is to support data-driven security research based on source code and source code metrics related to fixes for CVEs in the NVD by providing detailed information at different interlinked levels of abstraction, such as the commit, file, and method levels, as well as the repository and CVE levels. The database structure is shown in Figure 12. We selected the CVE description (cve.description), CVE scripts (file_change.file_name), and CVE method (method_change.name) from the database to construct the three-level inputs for the `CVE-Bench`.

Issue generation. Then, we perform an issue generation to generate the natural language-described issue as the input to the tested agents.

For the CVE desc level, we only provide the CVE description queried from the database. As for the CVE script level, the ground truth fix commit's modified script filename and path will be added to the issue. Finally, the CVE method level allows us to add the ground truth fix commit's modified method name and path to the issue.

Then, the generated issue will be input to the agents for code generation. The agents will create the patch to fix the vulnerability issues.

C Analysis Tool Details

The details of the four distinct analysis tools are as follows:

- Pylint: Pylint is a highly versatile Python static analysis tool that helps improve code quality and conform to coding standards. It checks for errors, enforces a coding standard, and looks for code smells. Pylint offers support for customizing the rules during the analysis, allowing developers to tailor the tool to their needs. It provides detailed reports on code quality, which can be integrated into development environments and continuous integration systems.
- Bandit: Bandit is a tool specifically designed for finding common security issues in Python code. It scans Python programs to identify vulnerabilities like SQL injection, cross-site scripting, hard-coded passwords, and more. Bandit is highly configurable, allowing users to exclude certain tests or files, and can be easily integrated into CI/CD pipelines for automated security testing.
- Mypy: Mypy is a static type checker for Python. By adding type annotations to Python code (using Python's typing module), mypy checks code and detects type errors before runtime, which can significantly improve code reliability and maintainability. Mypy supports Python's dynamic typing features while adding the benefits of static type checking, making it a powerful tool for large codebases.
- Prospector: Prospector is a comprehensive linting tool that bundles multiple Python static analysis tools such as Pylint, Bandit, and Mypy, among others. Running these tools simultaneously provides an aggregate view of code quality and potential issues. The prospector aims to simplify the setup and configuration of multiple linters and static analysis tools, providing a unified output that helps developers focus on improving code quality.

D Limitation

Limited programming language coverage. Because there are too many programming languages, our benchmark cannot cover too many types. At the same time, due to the high complexity of some programming languages, it is difficult for us to build a more robust unit test set.

Limited tested agent types. Since there are relatively few existing agent types, we only conducted experiments on SWE-Agent (Yang et al., 2024). Furthermore, Open-Devin (OpenDevin Contributors, 2024) does not provide a sufficient API for us to call. It only supports using its API, which limits `CVE-Bench` to establish large-scale experiments.

Limited tested foundation models. Until our paper is submitted, we have only performed the experiments on the GPT -4, GPT-3.5, and Llama-3 models. We plan to add more experiments before the publication.

repo name	description	stars
WordPress	WordPress, Git-ified. This repository is just a mirror of the WordPress subversion repository. Please do not send pull requests. Submit pull requests to https://github.com/WordPress/wordpress-develop and patches to https://core.trac.wordpress.org/ instead.	16559
showdoc	ShowDoc is a tool greatly applicable for an IT team to share documents online	10631
jenkins	Jenkins automation server	19376
symfony	The Symfony PHP framework	27375
october	Self-hosted CMS platform based on the Laravel PHP Framework.	10791
drawio	Source to app.diagrams.net	30936
zulip	Zulip server and web app—powerful open source team chat	16305
gocd	Main repository for GoCD - Continuous Delivery server	6561
cpython	The Python programming language	47121
etherpad-lite	Etherpad: A modern really-real-time collaborative document editor.	13151
onedev	Self-hosted Git Server with CI/CD and Kanban	9509
grav	Modern, Crazy Fast, Ridiculously Easy and Amazingly Powerful Flat-File CMS powered by PHP, Markdown, Twig, and Symfony	13415
netty	Netty project - an event-driven asynchronous network application framework	29819
synapse	Synapse: Matrix homeserver written in Python 3/Twisted.	9837
qutebrowser	A keyboard-driven, vim-like browser based on PyQt5.	8091
django	The Web framework for perfectionists with deadlines.	65901
jquery-ui	The official jQuery user interface library.	11076
parse-server	API server module for Node/Express	19696
Pillow	The friendly PIL fork (Python Imaging Library)	10080
salt	Software to automate the management and configuration of any infrastructure or application at scale. Get access to the Salt software package repository here:	12675
yii2	Yii 2: The Fast, Secure and Professional PHP Framework	13980
mongoose	MongoDB object modeling designed to work in an asynchronous environment.	24696
ansible	Ansible is a radically simple IT automation platform that makes your applications and systems easier to deploy and maintain. Automate everything from code deployment to network configuration to cloud management, in a language that approaches plain English, using SSH, with no agents to install on remote systems. https://docs.ansible.com.	54296
guzzle	Guzzle, an extensible PHP HTTP client	22038
pipenv	Python Development Workflow for Humans.	23211
PHPMailer	The classic email sending library for PHP	18596
typed_ast	Modified fork of CPython's ast module that parses '# type:' comments	212
handlebars.js	Minimal templating on steroids.	16719
fail2ban	Daemon to ban hosts that cause multiple authentication errors	7393
NodeBB	Node.js based forum software built for the modern web	12887

Table 3: The selected repository, description and its star count.

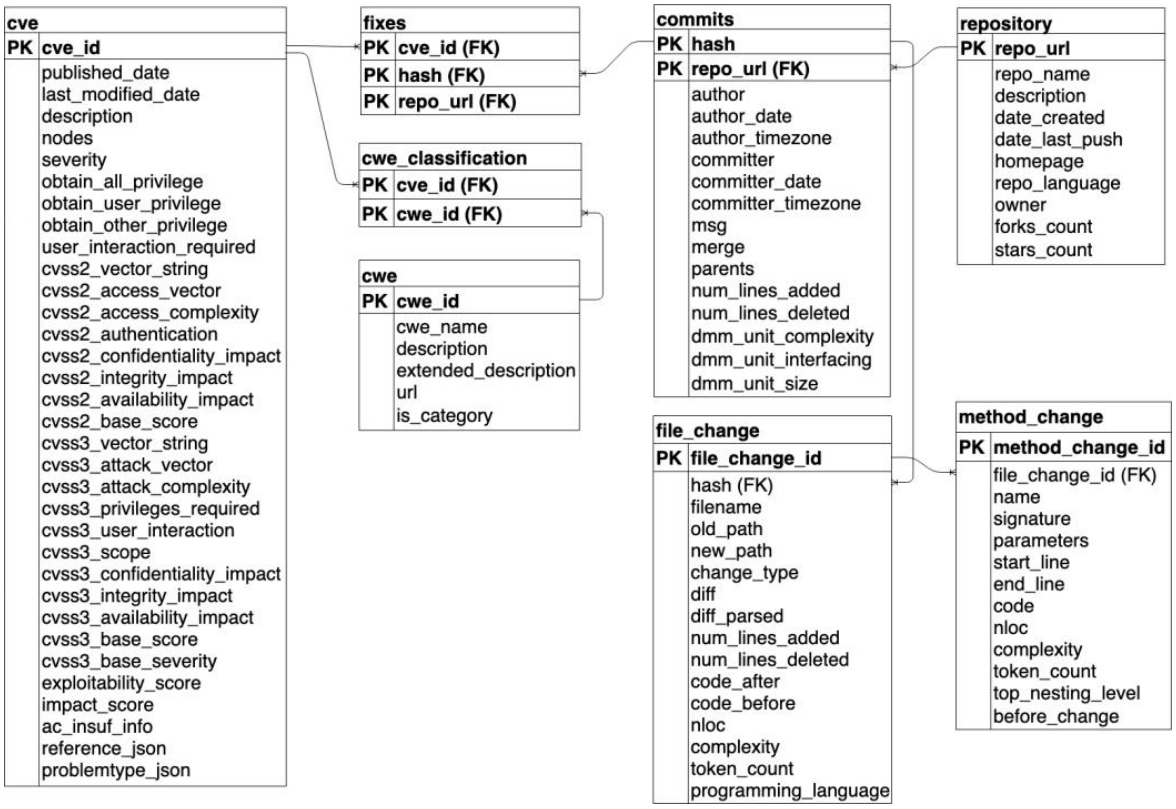


Figure 12: The database structure of the CVEFixes database, database contains recent CVE information and its corresponding fix commits information.