Not All Input Helps: What Information Should We Feed to LLMs for Vulnerability Repair?

Dongwook Choi
Department of Computer Science and Engineering
Sungkyunkwan University
Suwon, Republic of Korea
dwchoi95@skku.edu

Eunseok Lee
College of Computing and Informatics
Sungkyunkwan University
Suwon, Republic of Korea
leees@skku.edu

Abstract

Software vulnerabilities pose significant security risks, and timely repair is essential for mitigating potential exploits. Large Language Models (LLMs) have shown promise in automating vulnerability repair, but their effectiveness heavily depends on the input information provided. This paper systematically investigates the impact of different input types on LLM-based vulnerability repair performance. We surveyed 26 recent studies to categorize the input information used, including vulnerable code snippets, vulnerable line markers, CVE/CWE identifiers and descriptions, and additional metadata such as patch context. Through extensive experiments, we analyzed how these inputs, individually and in combination, influence the accuracy of vulnerability repair across various LLM architectures and datasets. Our findings reveal that not all input types contribute positively to repair performance; some may introduce noise or redundancy that hinders the model's ability to generate effective patches. Based on our analysis, we provide practical guidelines for selecting and structuring input information to optimize LLM-based vulnerability repair.

CCS Concepts

• **Security and privacy → Software security engineering**; **Web application security**.

Keywords

Vulnerability Repair, Software Security, Large Language Models

ACM Reference Format:
Dongwook Choi and Eunseok Lee. 2026. Not All Input Helps: What Information Should We Feed to LLMs for Vulnerability Repair?. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786847

1 Introduction

Modern software is characterized by massive dependencies and short release cycles, and even small defects can lead to serious security incidents. Vulnerabilities can be introduced throughout design, implementation, configuration, and deployment, and are exploited by malicious actors for intrusion, privilege escalation, and theft of sensitive information. According to the National Vulnerability Database (NVD[1]), more than 38,783 new Common Vulnerabilities and Exposures (CVE[2]) were reported in 2024, marking an annual record [3]. However, increases in submissions and infrastructure constraints have led to a substantial accumulation of unprocessed items [25]. In addition, 2024 vulnerability statistics report that the Mean Time to Remediate (MTTR) varies widely by type, averaging about 35 days (minimum 15 ~ maximum 274.8), with high-severity cases averaging 61 days [8]. Consequently, disclosure is outpacing vulnerability repair, and organizations struggle to manage risk as vulnerabilities accumulate.

Large Language Models (LLMs), trained on large code and natural language corpora, have shown competitive performance across diverse software engineering tasks [15], and their potential has also been demonstrated for automating vulnerability detection and vulnerability repair [32]. Accordingly, LLM-based vulnerability repair approaches that incorporate insights from conventional Automated Program Repair (APR) have been proposed, yet even state-of-the-art methods still report accuracy of about 20%, leaving a gap to real-world deployment [33]. To bridge this gap, a systematic analysis is needed of *which input clues LLMs rely on to understand vulnerabilities and to produce repair suggestions*. In particular, there is a lack of work that quantitatively analyzes how the input information used for vulnerability repair (e.g., *code snippets*, *vulnerable line*, *CVE/CWE IDs and descriptions*) affects performance.

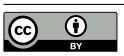
In this paper, we investigate the inputs provided to LLMs in LLM-based vulnerability repair studies and systematically analyze how they contribute to repair performance when fed individually or combinatively to the model. Our results empirically validate the hypothesis that "not all inputs helps", offering practical guidance on which information to select and arrange when designing LLM-based vulnerability repair for future research and practice. The main contributions are as follows:

- We survey 26 recent LLM-based vulnerability repair studies and systematize the categories and terminology of input information.
- We systematically analyze how diverse input combinations affect repair performance and derive practical guidelines.
- To enhance reproducibility, we release the prompts, scripts, datasets, and experimental results of this study [7].

2 Related Work

To systematize the input information used in LLM-based vulnerability repair studies, we adopt the search strategy of Zhou et al. [33], who surveyed LLM-based vulnerability detection and

[1]https://nvd.nist.gov
[2]https://www.cve.org



ICSE-NIER '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2425-1/26/04
https://doi.org/10.1145/3786582.3786847

repair, but extend the search range to January 2018–September 2025. The target venues include major conferences and journals in Software Engineering, Security, and Artificial Intelligence. We leveraged multiple databases, including Google Scholar, the ACM Digital Library, and arXiv, to conduct a primary search using keywords [34] related to LLMs and vulnerability repair, followed by a secondary search through references. The inclusion criteria were: (i) addressing *vulnerable code repair or patch generation* using LLMs, and (ii) providing reproducible artifacts.

Table 1: Inputs used in each Approach for LLM-based Vulnerability Repair

Approach	CVE		CWE				ETC	
	I	II	III	IV	V	VI	VII	VIII
VRepair [5]			✓				✓	✓
SeqTrans [6]							✓	✓
VulRepair [13]								✓
SPVF [36]	✓	✓	✓					✓
llm-vul [29]							✓	✓
ChatGPT4Vul [14]								✓
SCAN [31]								✓
Pearce et al. [23]							✓	✓
VulRep [28]								✓
VQM [11]							✓	✓
SecureCode [16]								✓
AIBugHunter [12]								✓
VulMaster [35]			✓	✓		✓		✓
VSP [20]			✓		✓		✓	✓
DeepCodeAIFix [2]								✓
VerilogRepair [1]			✓	✓	✓		✓	✓
SecRepair [17]								✓
Appatch [21]			✓				✓	✓
NAVRepair [27]								✓
ContractTinker [26]								✓
San2Patch [18]								✓
Rupam et al. [22]							✓	✓
VRpilot [19]								✓
ACFix [30]							✓	✓
LLM4CVE [9]	✓	✓	✓	✓	✓		✓	✓
PatchLM [3]							✓	✓

Table 1 summarizes the input types provided to LLMs across 26 LLM-based vulnerability repair approaches. A ✓ is shown only when the item was explicitly provided as an *LLM prompt or input directly consumed by the LLM* (internal use by pre-search/classification modules is excluded). All approaches supplied Vulnerable Code, and Vulnerable Lines were used in 62% (16/26) of the studies. Common Weakness Enumeration (CWE) information (ID/Name/Description/Example) was utilized more frequently than CVE information (ID/Description), as a single CVE often maps to multiple CWEs, making CWE more direct for characterizing the weakness type. Meanwhile, some studies are domain-specific areas (e.g., smart contracts, Verilog), where input requirements may differ from those of general-purpose programming languages. This paper's input information uses the 8 types above as the primary axes while normalizing definitional differences across studies and pipeline heterogeneity for fair comparison.

I **CVE ID**: Unique identifier for the CVE.
II **CVE Description**: Natural-language description of the CVE.
III **CWE ID**: Unique identifier for the CWE.
IV **CWE Name**: Name of the CWE.
V **CWE Description**: Natural-language description of the CWE.
VI **CWE Example**: Diff-style vulnerable-fix example for the CWE.
VII **Vulnerable Lines**: Location hints for the vulnerable line(s).
VIII **Vulnerable Code**: Function-level vulnerable code snippet.

In summary, the vulnerable code itself is a mandatory input, whereas the choice of auxiliary inputs varies across approaches; in particular, location hints and CWE-based metadata are frequently combined. This observation motivates and grounds the design of our experiments that analyze the contribution of each input and their combinations.

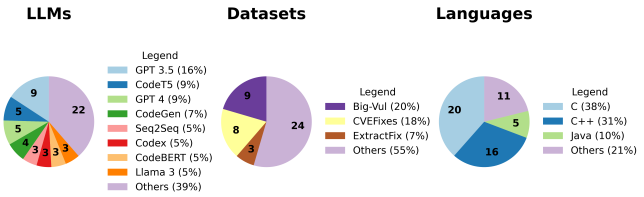


Figure 1: Distribution across 26 papers: LLMs & Datasets & Languages

Figure 1 summarizes the distribution of LLMs, datasets, and programming languages used in the surveyed 26 LLM-based vulnerability repair studies. LLMs from the GPT family (especially GPT-3.5/4) are the most widely used, accounting for over 25%, with code-specialized models like CodeT5 and CodeGen also frequently employed. The datasets BigVul and CVEfixes are the most commonly adopted, at 20% and 18%, respectively. Programming languages are dominated by C/C++ at 69%, followed by Java (10%) and others (Verilog, Solidity, etc.).

3 Study Design

3.1 Datasets

Taking the distribution in Figure 1 as a starting point, we prioritized *security-domain suitability* and *input information retention* as the top criteria: (i) covering as many of the 8 input types as possible, including security metadata such as CVE/CWE; (ii) ensuring reproducibility via publicly available artifacts; and (iii) including an *external validation set* to check for potential pre-training leakage.

Base sets. We adopt the widely used BigVul [10] and CVEfixes [4]. BigVul consists of CVE-linked vulnerable commits (3,754 instances, 91 CWE types) and includes *CVE ID & Description, CWE ID, Vulnerable Lines & Code*. CVEfixes comprises CVE-linked vulnerable commits (5,365 instances, 180 CWE types) and contains 7 of the 8 input types, excluding *CWE Example*.

Supplementing missing values. To supplement missing *CWE Example*, we refer to the supplementary materials from VulMaster [35] (which included *human evaluation* of LLM-generated patches) and accept *only items that passed human verification.*

External validation set. The base sets (1999–2021) may overlap with the pre-training data of recent LLMs. Accordingly, we add ZeroDay (Appatch) [21], reconstructed from vulnerabilities after April 2024, and use it as a small external validation set with the same 8 input types.

Final scale. After filtering duplicates and supplementing missing values, the base sets comprise 997 (BigVul) and 1,593 (CVEfixes) instances, while the external validation set contains 62 (ZeroDay), for a total of 2,652 instances.

3.2 Prompt Design

To isolate the effect of input information without bias, we use a minimal, fixed prompt schema (Role, Task, Input, Output) from Table 2 consistently across all experiments. The design principles are: (i) *Role* provides minimal expertise context while avoiding style instructions or examples that could bias outputs; (ii) *Task* uses imperative commands specifying "no explanation" and "preserve functionality" to constrain output form and scope; (iii) *Input* wraps data in XML-like tags to mark clear boundaries and minimize linguistic hints from the prompt itself, ensuring the model focuses on the actual data; and (iv) *Output Format* restricts responses to a fixed schema, suppressing free-form text and ensuring consistent parsing for fair comparison.

Table 2: Prompt Template for Vulnerability Repair

Section	Context
Role	You are an expert in security, specialized in vulnerability repair.
Task	Given a vulnerable code snippet and additional vulnerability information, generate fixed code snippet by fixing the vulnerability in it. Do not provide explanations or comments. Preserve the original functionality.

Input	`Vulnerable Code Snippet:`
	`<vulnerable_code>`
	`{vulnerable_code}`
	`</vulnerable_code>`
	`Additional Vulnerability Information:`
	`{selected_information}`

Output Format	`Fixed Code Snippet:`
	`<fixed_code>`
	`...fixed code here...`
	`</fixed_code>`

3.3 Evaluation Metrics

To evaluate vulnerability repair, we employ CodeBLEU [24], Exact Match (EM), and LLM Evaluation [26], following prior work [3, 16, 21, 35]. CodeBLEU extends BLEU by incorporating code syntax and semantics, providing greater precision than simple text matching. EM is a strict metric that checks character-level identity with the ground-truth patch. LLM Evaluation judges whether the generated patch equivalently resolves the vulnerability. For efficiency, we measure response time (seconds) and token counts (#Input Tokens, #Output Tokens), which are computed separately due to different API pricing.

3.4 Experimental Setup

The goal of this paper is to systematically analyze performance under a full-factorial combination of inputs. Comparing multiple LLMs simultaneously would cause the factor space and dataset instances to grow exponentially, $F_{full} \times N_{instance} \times N_{LLM}$, leading to exploding costs and confounding. Therefore, to *hold the model factor constant* and manipulate *only the input factors*, we select **gpt-3.5-turbo**[3] as the representative LLM, since it has the highest usage in Figure 1. This choice prioritizes the *internal validity* of input effects; generalization across multiple LLMs is complemented on the external validation set by additionally evaluating **claude-3-haiku**[4]. Both models have a pre-training data cutoff before April 2024. The model temperature is fixed at 0.0 to induce deterministic outputs. We allow up to 5 retries for API calls.

3.5 Research Questions

RQ1 Which input combinations are most effective for LLM-based vulnerability repair? We evaluate the accuracy and efficiency of vulnerability repair across all full-factorial input combinations on the base sets.

RQ2: How generalizable are the input combinations? To verify whether the RQ1 results generalize to unseen data and different LLMs, we evaluate repair performance on the external validation set using both GPT-3.5-turbo and Claude-3-haiku.

4 Preliminary Evaluation

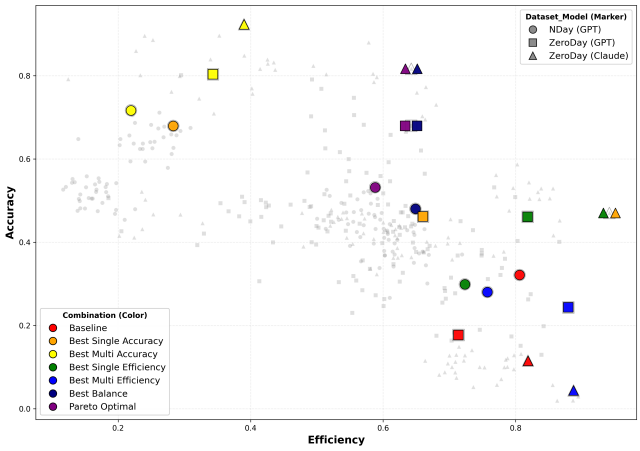


Figure 2: Results of All 128 Combinations

4.1 RQ1: Accuracy & Efficiency

Figure 2 presents results for all 128 (= 2^7) input combinations on the base set (NDay) and the validation set (ZeroDay). In Figure 2, "Baseline" denotes the reference setting that uses only the vulnerable code without any auxiliary inputs. Accuracy is the normalized mean of CodeBLEU, Exact Match, and LLM Evaluation, while Efficiency is the normalized mean of response time and input/output token counts.

[3] https://platform.openai.com/docs/models/gpt-3.5-turbo
[4] https://www.anthropic.com/news/claude-3-haiku

Accuracy. The baseline achieves CodeBLEU 0.733, EM 0.0%, and LLM Evaluation 25.0%. Among single-input settings, using *CWE ID* yields the best performance (0.772, 0.3%, 31.3%), while among multi-input settings, *CVE ID + CWE ID + CWE Name + CWE Description* performs best (0.770, 0.3%, 34.3%). Notably, 96.1% (123/128) of all combinations outperform the baseline in accuracy; among these, *CVE ID* appears in 52% (64/123), suggesting that it is a particularly informative signal for improving accuracy. In contrast, *Vulnerable Lines* is included in 100% (4/4) of the combinations that underperform the baseline, indicating a negative impact on accuracy.

Efficiency. The baseline consumes 1,056 input tokens (T_{in}) and 730 output tokens (T_{out}) per instance, with an average response time of 14.22 seconds. The most efficient single-input setting is *Vulnerable Lines* (1,466 T_{in}, 635 T_{out}, 13.71 s). *Vulnerable Lines* uses more T_{in}, but reduces the number of T_{out} and the response time by focusing the repair on specific lines. The most efficient multi-input setting is *CWE ID + Vulnerable Lines* (1,473 T_{in}, 640 T_{out}, 12.95 s), exhibiting a similar trend. Notably, all combinations except the baseline show lower efficiency, primarily due to increased token counts. Among these, *CWE Example*—which consumes the most tokens—exhibits the lowest efficiency, while *Vulnerable Lines*—which uses the fewest tokens—demonstrates the highest efficiency.

Trade-Off. To identify practical combinations, we compute a balanced score that combines normalized accuracy and efficiency. The best balanced (highest score) trade-off is achieved by *CVE ID + CWE Name + Vulnerable Lines*, with Accuracy: 0.480 (CodeBLEU=0.680, EM=0.6%, LLMEval=26.5%) and Efficiency: 0.649 (#Input=1486, #Output=715, Time=13.60s). The Pareto optimal combination is *CVE ID + CVE Description + CWE ID + Vulnerable Lines*, yielding Accuracy: 0.532 (CodeBLEU=0.685, EM=0.7%, LLMEval=25.8%) and Efficiency: 0.588 (#Input=1549, #Output=734, Time=13.86s). The best balanced attains substantial efficiency improvements at the cost of some accuracy, whereas the Pareto optimal maintains a more balanced accuracy–efficiency profile but exhibits markedly lower efficiency than the best balanced trade-off setting.

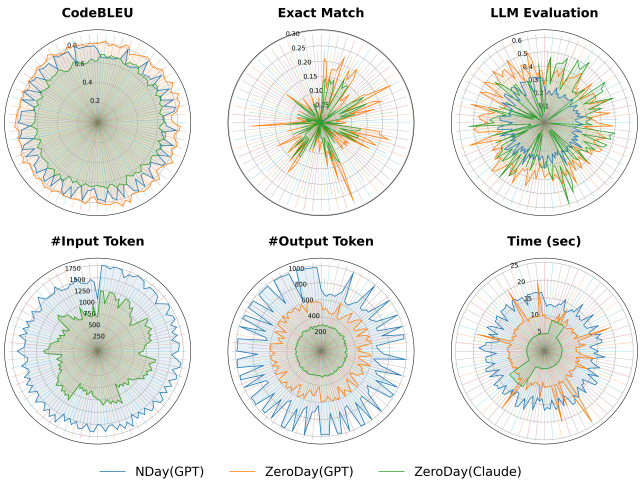


Figure 3: Comparison on the NDay and ZeroDay dataset between GPT and Claude models

4.2 RQ2: Generalizability

Figure 3 compares GPT and Claude models on the NDay and ZeroDay datasets. For CodeBLEU and Exact Match, NDay(GPT) and ZeroDay(GPT) exhibit higher similarity than ZeroDay(Claude), as both use the same model. Notably, ZeroDay(GPT) and ZeroDay(Claude)—which differ only in model choice while using the same dataset—show approximately 84% similarity. In contrast, LLM Evaluation exhibits lower similarity (69–75%), suggesting this metric is more sensitive to model-specific characteristics. For efficiency metrics, #Input Tokens match exactly because the experiments use identical datasets. However, for #Output Tokens and Time (sec), GPT tends to generate more tokens and exhibit longer response times than Claude. This may stem from differences in model architecture and optimization strategies.

5 Future Plans

Evaluate with more diverse LLMs and datasets. The number of LLMs selected in this study is limited, and the work focuses only on C/C++. In future work, we plan to broaden the generalizability of our findings by evaluating the most effective input combinations across diverse LLMs and programming languages.

Explore prompting methods to improve the performance. This study used a fixed prompt template in a zero-shot setting; we will investigate performance improvements by applying prompt engineering techniques (e.g., few-shot prompting, Chain-of-Thought, Retrieval-Augmented Generation).

Evaluate the robustness of LLM outputs. While this study focused on accuracy and efficiency, it is also important to evaluate the robustness of LLM outputs (e.g., consistency for identical inputs, sensitivity to noise). In future work, we plan to introduce robustness metrics to assess the reliability of LLM-based vulnerability repair.

6 Conclusion

This paper systematically analyzes how input information affects LLM-based vulnerability repair. By reviewing 26 prior studies, we categorize eight input types and evaluate all 128 full-factorial combinations across 2,652 real-world vulnerabilities from two base datasets and one external validation set. Our findings reveal that not all inputs contribute positively: 96.1% of combinations outperform the baseline. *CVE ID* emerges as the most critical signal for accuracy, appearing in 52% of best performing combinations, while *Vulnerable Lines* proves essential for efficiency despite degrading accuracy. We identify two practical combinations balancing accuracy and efficiency: *CVE ID + CWE Name + Vulnerable Lines* for optimal efficiency and *CVE ID + CVE Description + CWE ID + Vulnerable Lines* for Pareto-optimal accuracy. These patterns generalize across models with 84% similarity. This work provides actionable guidelines for input selection in LLM-based vulnerability repair.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438686, Development of software reliability improvement technology through identification of abnormal open sources and automatic application of DevSecOps)

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215* (2023).

[2] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. 2024. Deepcode AI fix: Fixing security vulnerabilities with large language models. *arXiv preprint arXiv:2402.13291* (2024).

[3] Guru Bhandari, Nikola Gavric, and Andrii Shalaginov. 2025. Generating vulnerability security fixes with Code Language Models. *Information and Software Technology* (2025), 107786.

[4] 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*. 30–39.

[5] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[6] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 564–585.

[7] Dongwook Choi. 2026. Github - Not All Input Helps: What Information Should We Feed to LLMs for Vulnerability Repair? https://github.com/dwchoi95/BILVR.git.

[8] Edgescan. 2024. *2024 Vulnerability Statistics Report* (9 ed.). Industry report. Edgescan. https://www.edgescan.com/wp-content/uploads/2025/04/2024-Vulnerability-Statistics-Report.pdf

[9] Mohamad Fakih, Rahul Dharmaji, Halima Bouzidi, Gustavo Quiros Araya, Oluwatosin Ogundare, Mst Ayesha Siddika, and Mohammad Abdullah Al Faruque. 2025. Llm4cve: Enabling iterative automated vulnerability repair with large language models. In *2025 28th Euromicro Conference on Digital System Design (DSD)*. IEEE, 592–599.

[10] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories*. 508–512.

[11] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.

[12] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering* 29, 1 (2024), 4.

[13] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.

[14] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we?. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 632–636.

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

[16] Nafis Tanveer Islam, Mohammad Bahrami Karkevandi, and Peyman Najafirad. 2024. Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031* (2024).

[17] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Mohammad Bahrami Karkevandi, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. 2024. Llm-powered code vulnerability repair with reinforcement learning and semantic reward. *arXiv preprint arXiv:2401.03374* (2024).

[18] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. 2025. Logs In, Patches Out: Automated Vulnerability Repair via {Tree-of-Thought}{LLM} Analysis. In *34th USENIX Security Symposium (USENIX Security 25)*. 4401–4419.

[19] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. 103–111.

[20] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230* (2024).

[21] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {Real-World} software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*. 4481–4500.

[22] Rupam Patir, Qiqing Huang, Keyan Guo, Wanda Guo, Guofei Gu, Haipeng Cai, and Hongxin Hu. 2025. Towards LLM-Assisted Vulnerability Detection and Repair for Open-Source 5G UE Implementations. (2025).

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

[24] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[25] VulnCheck. 2024. Danger is still lurking in the NVD backlog. https://vulncheck.com/blog/nvd-backlog-exploitation-lurking. Accessed: 16 December 2024.

[26] Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. 2024. Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2350–2353.

[27] Ruoke Wang, Zongjie Li, Chaozheng Wang, Yang Xiao, and Cuiyun Gao. 2024. Navrepair: Node-type aware c/c++ code vulnerability repair. *arXiv preprint arXiv:2405.04994* (2024).

[28] Ying Wei, Lili Bo, Xiaoxue Wu, Yue Li, Zhenlei Ye, Xiaobing Sun, and Bin Li. 2023. VulRep: vulnerability repair based on inducing commits and fixing commits. *EURASIP Journal on Wireless Communications and Networking* 2023, 1 (2023), 34.

[29] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294.

[30] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2025. ACF ix: Guiding LLMs with Mined Common RBAC Practices for Context-Aware Repair of Access Control Vulnerabilities in Smart Contracts. *IEEE Transactions on Software Engineering* (2025).

[31] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2023), 2507–2525.

[32] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).

[33] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–31.

[34] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large language model for vulnerability detection and repair: Literature review and the road ahead. https://docs.google.com/document/d/18-UrkfH35CNMGRjjsDYZGK6L1aC9wP3GsKCtrIekcUQ/edit?tab=t.0 Keywords.

[35] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13.

[36] Zhou Zhou, Lili Bo, Xiaoxue Wu, Xiaobing Sun, Tao Zhang, Bin Li, Jiale Zhang, and Sicong Cao. 2022. SPVF: security property assisted vulnerability fixing via attention-based models. *Empirical Software Engineering* 27, 7 (2022), 171.