All That Glitters is Not Gold: Vulnerability-Driven Repairing in ChatGPT-Generated C Source Code

Clodoaldo B. Leite Neto
Zhejiang University
Hangzhou, Zhejiang, China
clodoaldo@zju.edu.cn

Haitao Xu
Zhejiang University
Hangzhou, Zhejiang, China
haitaoxu@zju.edu.cn

Gabriel H. M. Oliveira
Universidade de Fortaleza
Fortaleza, Ceará, Brasil
habib@edu.unifor.br

Danyllo Albuquerque
Federal Institute of Paraíba (IFPB)
Campina Grande, Paraíba, Brazil
danyllo.albuquerque@ifpb.edu.br

Rohit Gheyi
Fed. Univ. of Campina Grande (UFCG)
Campina Grande, Paraíba, Brazil
rohit@dsc.ufcg.edu.br

Mirko Perkusich
VIRTUS/UFCG
Campina Grande, Paraíba, Brazil
mirko@virtus.ufcg.edu.br

Abstract

Large language models (LLMs) are increasingly used for source code generation, yet they may produce vulnerable code. This study investigates security vulnerabilities in GPT-4-generated C programs and evaluates whether they can be automatically repaired. We generated solutions for 974 C programming tasks using GPT-4, collected four independent generations per task, and analyzed them with the *GCC* static analyzer. Detected issues were mapped to the Common Weakness Enumeration (CWE) taxonomy, and an iterative self-healing framework was used to repair vulnerable code based on static-analysis feedback. Vulnerabilities were detected in approximately 17% of the generated programs, spanning 10 CWEs grouped into three CWE pillars, with CWE-690 (Unchecked Return Value to NULL Pointer Dereference) and CWE-758 (Reliance on Undefined, Unspecified, or Implementation-Defined Behavior) being the most frequent. The proposed framework eliminated all vulnerabilities detected by the static analyzer within 4–7 repair rounds across the four samples, without human intervention. These results demonstrate that iterative LLM-assisted repair guided by static-analysis feedback is an effective strategy for mitigating vulnerabilities in GPT-4-generated C code.

Keywords

Vulnerability Analysis, Large Language Models, Source Code Generation, Automatic Program Repair

1 Introduction

Large Language Models (LLMs) have emerged as a transformative AI technology with applications across a wide range of domains. According to a 2024 report by the International Monetary Fund, 40% of global employment is exposed to AI [3]. ChatGPT, one of the most widely used LLM-based systems, can generate coherent text and source code across a wide range of tasks [6]. These technologies are reshaping work practices by assisting users with writing, problem solving, and other knowledge-intensive tasks. In computer science, LLMs have proven valuable for generating code snippets, explaining APIs, and assisting with software development tasks across multiple programming languages [6].

Despite these capabilities, the security of LLM-generated code remains an open challenge. While proficient in generating code, LLMs may lack inherent awareness of the intricate security considerations that surround software development [21]. They are unaware of the multifaceted risks associated with programming, which usually requires proper resource management, understanding of the complexities of operating systems, and interactions with other systems [18]. These challenges are formidable even for human developers, necessitating in-depth analysis and expertise [11].

Motivated by these concerns, this paper aims to evaluate and rectify code generated by LLMs in response to code-related prompts. Employing a static analysis tool, we systematically identified and categorized potential weaknesses within the generated source code. Our objective is to scrutinize these weaknesses for further analysis and ensure that the LLM-generated code adheres to essential security principles. Furthermore, we propose an iterative automatic repair method that integrates vulnerability warnings generated by the static analysis tools with the potentially vulnerable sources and then prompts the LLM for fixes, as illustrated in Figure 1. Our devised approach strategically leverages the non-determinism inherent in LLMs to advantageously induce the generation of fixes for the source code through detected vulnerability reports. Remarkably, in most repair iterations, the source code outputs exhibit divergence from the source code inputs. Our design facilitates the LLM in circumventing previously encountered mistakes.

Specifically, to shed light on the usability and security implications of LLM-generated code, we applied static analysis over a large corpus of source code generated by LLMs from a set of representative programming tasks and confirmed that LLMs might introduce various security bugs in their code outputs. Then, by applying our proposed repair framework, our experiments showed that GPT-4 can repair reported vulnerabilities when static-analysis warnings are incorporated into the source code and the annotated program is resubmitted to the model. The main contributions of this work are threefold. First, we propose a fully automated repair framework that combines static analysis with LLMs to eliminate vulnerabilities detected in generated C code iteratively. Second, we characterize the security weaknesses present in GPT-4-generated C code through a systematic CWE-based analysis. Third, we investigate the non-deterministic behavior of LLMs by analyzing multiple generations for the same programming tasks.

2 Methodology Overview

To improve the security of LLM-generated source code, we propose the vulnerability-driven repair workflow illustrated in Figure 1.

The workflow comprises five steps. First, a programming task is submitted to the LLM (①), which generates C source code (②). The generated code is then analyzed by the *GCC* static analyzer

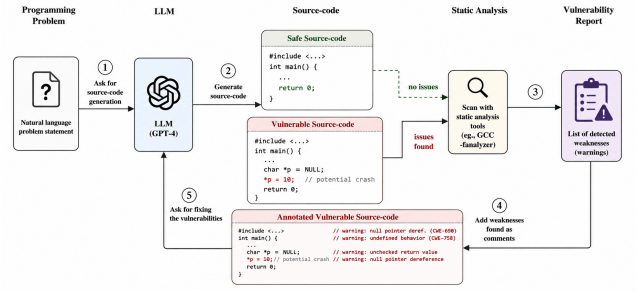

Figure 1: Overview of the proposed repair workflow.

(③). If no vulnerabilities are reported, the code is considered *safe* within the scope of this study. Otherwise, the reported warnings are embedded as comments at their respective locations in the vulnerable source code (④). The resulting annotated code is submitted to the LLM through a repair prompt (⑤), instructing it to address the reported vulnerabilities. The repaired code is then returned to the static-analysis step, forming an iterative repair cycle that continues until no vulnerabilities are reported or the maximum number of repair iterations is reached. Thus, rather than relying on manually crafted repair rules, the workflow uses static-analysis feedback to iteratively guide the LLM toward safer implementations. Different prompts are used for code generation and repair (steps ② and ⑤), as described in the following sections.

3 Source Code Generation

To generate source code for vulnerability analysis, we curated the *Basic C Tasks* (BCT) dataset based on the Mostly Basic Programming Problems (MBPP) benchmark [2]. BCT comprises 974 entry-level programming tasks adapted to produce C implementations, each including a problem statement, a reference solution, and test cases. We selected entry-level tasks because they represent common scenarios in which developers may rely on LLM-generated code with little or no modification [16].

For each task, GPT-4 generated four independent implementations to account for the variability introduced by LLM non-determinism, resulting in 3,896 source files. Code generation followed established prompt engineering practices [19], combining the *Persona* and *Output Automater* patterns to request complete C source code without additional explanations. Any Markdown formatting or explanatory text included in the responses was automatically removed before analysis.

The generated source files were then submitted to static analysis for vulnerability detection. Successful compilation was not required, since static analyzers can identify many security issues directly from source code. This choice also reflects practical scenarios in which developers may reuse LLM-generated code snippets before verifying their compilability [6, 10]. Accordingly, our analysis focuses on vulnerabilities detectable in generated source code independently of whether the programs successfully compile.

4 Experimental Setup

All experiments were conducted using *GPT-4* (*gpt-4-0613*) with the default *temperature* and *top_p* settings (t=1, top_p=1). Static analysis was performed using the *GCC* 13.2.0 analyzer (*-fanalyzer*). All experiment scripts were implemented in *Python* 3.11 and are publicly available (see Artifact Availability Section).

Only source files with at least one reported vulnerability entered the repair stage. Repair experiments were conducted in a zero-shot setting without conversation history. At each iteration, the static-analysis warnings were embedded into the corresponding source code as comments and submitted back to the LLM using the repair prompt described in Section 6. This process continued until no vulnerabilities were reported by the analyzer.

5 Vulnerability Analysis

This section characterizes the vulnerabilities identified in the generated source code, describing the analysis process, the CWE-based classification, and the observed vulnerability patterns.

Analysis Tools. Source code was analyzed using the *GCC* static analyzer (*-fanalyzer*), which detects a broad range of programming defects, including null-pointer dereferences, memory leaks, buffer overflows, use-after-free, double free, invalid memory accesses, and uninitialized variables [5]. We selected *GCC* because it is widely available, fully automatable, and reports warnings that can be directly integrated into our repair pipeline. The analyzer reports the source location, warning type, associated CWE (when available), and the corresponding analysis rule. Representative outputs are shown in Table 1.

Table 1: Sample outputs from GCC analyzer.

Sample Output
`problem-102-1.c:13:25: warning: dereference of possibly-NULL 'camelStr' [CWE-690] [-Wanalyzer-possible-null-dereference]`
`problem-114-1.c:23:23: warning: leak of '<unknown>' [CWE-401] [-Wanalyzer-malloc-leak]`
`problem-231-1.c:7:25: warning: stack-based buffer over-read [CWE-126] [-Wanalyzer-out-of-bounds]`

CWE Classification. To standardize vulnerability analysis and enable comparisons across different warning types, all analyzer reports were mapped to the Common Weakness Enumeration (CWE) taxonomy. For warnings without explicit CWE identifiers, we created a mapping dictionary based on the *GCC* documentation and the semantic meaning of each warning. The complete mapping dictionary is publicly available in our replication package (see Artifact Availability Section).

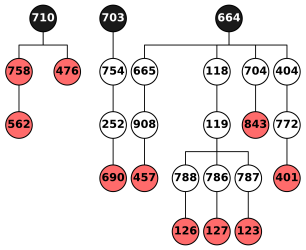

Figure 2: CWE hierarchy containing the detected weaknesses.

Analysis Results. We first quantify the prevalence of vulnerabilities in the generated programs and then characterize the identified weaknesses using the CWE taxonomy.

How many generated programs contained detected vulnerabilities, and how many vulnerabilities were identified overall?

Across the four samples, an average of 162.25 out of 974 generated programs per sample (approximately 17%, 15.7–17.8% across samples) contained at least one detected vulnerability, as shown in the "No Repair" category of Figure 3. In total, the analyzer reported an average of 385.75 weaknesses, corresponding to 2.38 weaknesses per vulnerable program.

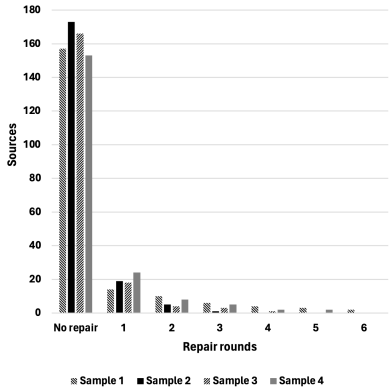

Figure 3: Number of vulnerable source files per repair round.

Figure 4 shows that more than 90% of vulnerable tasks contained at most five detected weaknesses, whereas an average of only two tasks contained ten or more detected weaknesses. This distribution indicates that most vulnerable programs contain only a small number of detected weaknesses, while a few tasks consistently accumulate multiple vulnerabilities.

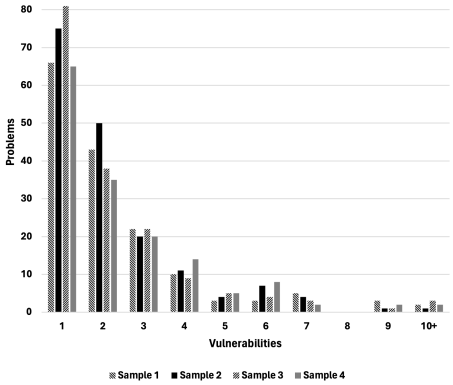

Figure 4: Frequency of vulnerabilities per problem.

Table 2 lists the ten tasks with the highest number of detected vulnerabilities. The substantial variation observed for Tasks 570 and 108 across samples illustrates the non-deterministic behavior of GPT-4, which may generate either secure or vulnerable implementations for the same programming task.

These observations suggest that vulnerability occurrence depends not only on the programming task itself but also on the stochastic nature of LLM generation, motivating the evaluation of multiple independent samples.

Table 2: Top 10 programming problems with the most detected vulnerabilities.

Problem	S1	S2	S3	S4	Avg
450	13	18	17	13	**15.25**
827	9	7	7	9	**8**
536	7	7	7	4	**6.25**
570	22	1	0	1	**6**
175	7	5	5	6	**5.75**
966	3	6	6	6	**5.25**
231	5	5	5	5	**5**
958	2	2	2	14	**5**
108	0	4	10	5	**4.75**
469	6	6	0	6	**4.5**

Types of Weaknesses Identified. Overall, we identified 10 distinct CWEs belonging to three major CWE pillars (Figure 2), according to the Research Concepts view (CWE-1000) of CWE version 4.20 [12], namely resource-management weaknesses (CWE-664), coding-standard violations (CWE-710), and improper check or handling of exceptional conditions (CWE-703). These three pillars encompass all vulnerabilities detected throughout the experiment.

- *CWE-664: Improper Control of a Resource Through its Lifetime*, which comprises resource-management issues such as memory leaks, invalid memory access, and buffer overflows.
- *CWE-710: Improper Adherence to Coding Standards*, which includes violations that may lead to undefined behavior, such as returning the address of local variables.
- *CWE-703: Improper Check or Handling of Exceptional Conditions*, which includes weaknesses related to the inadequate verification and treatment of error conditions, such as unchecked return values that may lead to NULL pointer dereferences.

This hierarchy provides a higher-level view of the detected security issues, allowing individual weaknesses to be interpreted within broader vulnerability categories. Figure 5 summarizes the distribution of detected weaknesses across the identified CWEs.

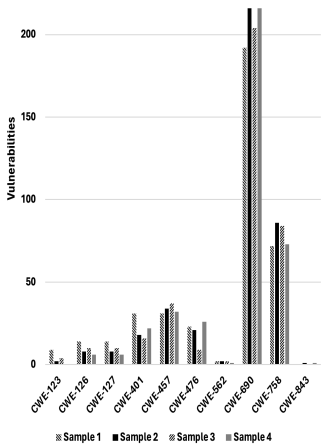

Figure 5: Detected vulnerabilities per CWE before applying the repair mechanism.

Table 3: Prompt template of the repair process.

Prompt Template
Acting as an experienced C developer, analyze the following source-code: `[source-code]` Re-write the source-code, paying attention to the comments to check for fixes for the possible weaknesses identified. Don't forget to add a main function, and proper includes and function definitions. Don't write any comments; just write the code.

CWE-690 was the most frequent weakness in all four samples, followed by CWE-758, which ranked second in every sample. Furthermore, eight of the ten identified CWEs also appeared consistently across all samples, suggesting that GPT-4 reproduces similar classes of security weaknesses despite its non-deterministic outputs. This consistency indicates that the observed vulnerabilities are not isolated artifacts of individual generations, but recurring patterns that can be systematically detected and subsequently repaired.

6 Iterative Automatic Repair Method

Approximately 17% of the generated programs contained at least one detected vulnerability, motivating the development of an automated repair strategy. The objective of this stage is to investigate whether an LLM can iteratively repair its own generated code using only feedback provided by a static analysis tool, without relying on handcrafted repair rules or human intervention.

LLM Feedback Strategy. Potentially vulnerable source code was annotated with the warnings reported by the *GCC* analyzer, which were inserted as C comments immediately before the corresponding source-code lines, as illustrated in Figure 6. When multiple warnings referred to the same location, they were grouped into a single multi-line comment to preserve readability. The annotated source code was then submitted to GPT-4 using the repair prompt shown in Table 3.

Unlike previous approaches that rely on manually crafted repair hints, our framework directly exploits the feedback produced by the static analyzer. This design enables the repair process to be fully automated while remaining independent of vulnerability-specific repair rules.

Repair Analysis. Overall, the iterative repair process generated 780 additional source files, resulting in 4,676 generated programs throughout the experiment. After the first repair round, the average number of vulnerable programs decreased from 162.25 to 18.75, corresponding to an 88% reduction. Likewise, the average number of detected weaknesses dropped from 385.75 to 62.5 (84% reduction), as shown in Figures 3 and 7. These results indicate that the majority of reported vulnerabilities can be eliminated with a single repair iteration.

Although highly effective, the first repair round did not eliminate all vulnerabilities. In several cases, GPT-4 successfully removed the reported weakness but simultaneously introduced new vulnerabilities. Consequently, only the remaining vulnerable programs were resubmitted for additional repair iterations, while repaired programs were excluded from subsequent rounds.

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

void removeDigits(char *str) {
    char *i = str;
    char *j = str;

    while (*j != '\0') {
        if (!isdigit(*j)) {
            *i = *j;
            i++;
        }
        j++;
    }

    *i = '\0';
}
```

(a) Zero-shot generated source code.

```
collected_code/problem-27-0.c: In function 'removeDigits':
collected_code/problem-27-0.c:11:16: warning: write to string literal
[-Wanalyzer-write-to-string-literal]
   11 |             *i = *j;
      |             ~~~^~~~
'main': events 1-2
   |
   |   26 | int main() {
   |      |     ^~~~
   |      |     |
   |      |     (1) entry to 'main'
   |......
   |   30 |     removeDigitsFromList(list, size);
   |      |     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   |      |     |
   |      |     (2) calling 'removeDigitsFromList' from 'main'
   |
   +--> 'removeDigitsFromList': events 3-6
```

(b) Static analysis report.

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

void removeDigits(char *str) {
    char *i = str;
    char *j = str;

    /* Possible weaknesses found:
     * Cannot determine that '*j' is initialized
       [premium-bughuntingUninit]
     */
    while (*j != '\0') {
        if (!isdigit(*j)) {
            /* Possible weaknesses found:
             * write to string literal
               [-Wanalyzer-write-to-string-literal]
             */
            *i = *j;
            i++;
        }
        j++;
    }
```

(c) Annotated source code used for repair.

Figure 6: Example of static-analysis feedback incorporated into the repair prompt.

After the second repair round, an additional 64% of the remaining vulnerable programs were repaired, reducing the average number of vulnerable programs from 18.75 to 6.75 and the average number of detected weaknesses from 62.5 to 14.25 (77% reduction). Figure 7 shows that the number of vulnerabilities consistently decreased throughout successive repair rounds, indicating that the proposed feedback mechanism progressively improves the generated code.

Complete repair was achieved after 7, 4, 5, and 6 rounds for Samples 1–4, respectively, requiring approximately 5.5 rounds on average. Overall, the proposed framework eliminated all vulnerabilities detected by the static analyzer without human intervention.

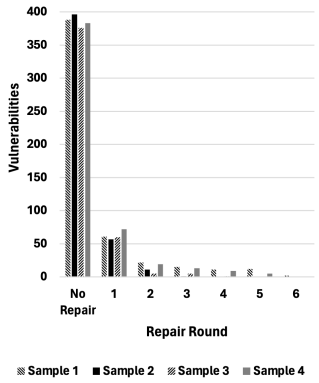

Figure 7: Detected vulnerabilities across repair rounds.

These findings demonstrate that iterative static-analysis feedback provides an effective mechanism for automatically repairing LLM-generated C code while exploiting the inherent variability of successive LLM generations.

7 Related Work

Recent studies have investigated the security of LLM-generated code from different perspectives, including secure code generation, vulnerability detection, and Automated Program Repair (APR). Existing work varies considerably in terms of programming languages, datasets, vulnerability detection techniques, and repair strategies, making direct comparisons challenging.

Several studies have evaluated the security of LLM-generated code across multiple programming languages [10, 13, 14], while others focus on specific domains such as Verilog [1] or C [17]. Our work concentrates exclusively on C because its explicit memory management, pointer manipulation, and undefined behaviors make it particularly susceptible to security vulnerabilities. Moreover, whereas previous studies typically analyze handcrafted examples, publicly disclosed vulnerabilities, or relatively small collections of generated programs, our evaluation comprises 974 programming tasks with four independent GPT-4 generations per task, providing a broader empirical basis for studying both vulnerability occurrence and generation variability.

The approaches used to identify vulnerabilities also differ substantially. Prior work has employed manual inspection, prompt-based assessment, compiler diagnostics, and static analysis [14, 15]. Similar to these studies, we rely on static analysis, but additionally normalize all reported weaknesses according to the CWE taxonomy. This enables a systematic characterization of vulnerability classes rather than simply counting analyzer warnings, facilitating comparisons across different categories of security weaknesses.

Research on LLM-assisted repair has likewise explored different strategies. Existing approaches rely on handcrafted repair prompts [1, 10], direct regeneration without explicit feedback [13], or analyzer-guided repair using tools such as CodeQL [15]. In contrast, our framework directly incorporates *GCC* analyzer warnings into the repair prompt and iteratively repairs only the remaining vulnerable programs until no vulnerabilities are reported. This design eliminates the need for manually engineered repair rules and exploits the non-deterministic behavior of LLMs to improve the generated code progressively.

Beyond security-oriented APR, recent research has discussed broader aspects of AI-assisted software development, including security risks [4], ethical implications [7], developer overconfidence [16], conversational repair [22], compiler-assisted repair [9], static-analysis-guided repair [8], and data leakage in repair benchmarks [23]. Collectively, these studies highlight both the opportunities and the challenges of integrating LLMs into software engineering workflows.

Compared with prior work, our study advances the state of the art in three ways. First, it provides large-scale empirical evaluation of LLM-generated C programs for security analysis. Second, it systematically maps detected weaknesses to the CWE taxonomy, enabling vulnerability-oriented analyses beyond raw analyzer outputs. Finally, it introduces a fully automated vulnerability-driven repair framework that relies exclusively on static-analysis feedback and successfully eliminates all vulnerabilities detected by the analyzer without requiring handcrafted repair hints.

8 Threats to Validity

Following the classification proposed by Wohlin et al. [20], we discuss the main threats to the validity of this study and the measures adopted to mitigate them.

Construct validity. Our study evaluates security through vulnerabilities reported by the *GCC* static analyzer. Although *GCC* is a mature and widely adopted tool, static analysis cannot detect all vulnerability classes and may produce false positives or false negatives. Consequently, repaired programs should be interpreted as free of vulnerabilities detected by the selected analyzer rather than free of all security flaws. To improve transparency and reproducibility, we systematically mapped analyzer warnings to the CWE taxonomy and publicly released the mapping in our replication package.

Internal validity. Threats to internal validity are mainly related to the experimental configuration. All experiments were performed using GPT-4 with its default parameters, and repair relied exclusively on static-analysis feedback. Different prompting strategies, model configurations, or repair policies could influence repair effectiveness. To mitigate this threat, we adopted a fully automated experimental pipeline, generated four independent implementations for each task to account for LLM stochasticity, and applied the same repair procedure to every generated program.

External validity. Our findings are based on GPT-4-generated C programs derived from 974 entry-level programming tasks. Although the dataset covers a broad range of introductory problems, the results may not generalize to larger software systems, other programming languages, industrial applications, or newer LLMs. Likewise, different static analysis tools may report different vulnerability classes. Nevertheless, the proposed methodology is largely independent of the underlying LLM, programming language, and analyzer, facilitating replication and extension in future studies.

Conclusion validity. Our conclusions are based primarily on descriptive statistics obtained from four independent generations per task. Although this sampling strategy captures part of the variability introduced by LLM non-determinism, additional generations or complementary statistical analyses could further strengthen the results. To facilitate independent verification, we publicly release the dataset, generated source code, analyzer mappings, experimental scripts, and repair artifacts.

9 Conclusion

This paper investigated the security of GPT-4-generated C code through a large-scale empirical study involving 974 programming tasks and four independent generations per task. We analyzed the generated programs using static analysis, classified the detected weaknesses according to the CWE taxonomy, and proposed an iterative repair framework guided by static-analysis feedback.

Our results show that around 17% of the generated programs contained vulnerabilities, with CWE-690 and CWE-758 emerging as the most frequent weakness classes. We also observed that the non-deterministic behavior of LLMs influences both the occurrence of vulnerabilities and the effectiveness of automated repair. The proposed repair framework successfully eliminated all vulnerabilities detected by the static analyzer without human intervention, demonstrating that static-analysis-guided feedback is an effective mechanism for improving the security of LLM-generated code.

These findings have important implications for both industry and research. For practitioners, they reinforce that LLM-generated code should not be assumed to be secure by default and that automated static analysis should become an integral part of AI-assisted software development workflows. Moreover, our results show that vulnerability detection and repair can be seamlessly integrated into code generation pipelines with minimal human intervention. For researchers, this work provides empirical evidence that vulnerability-driven feedback is a promising direction for secure code generation and establishes a reproducible benchmark for evaluating future automated program repair techniques.

Future work includes extending the proposed framework with additional static and dynamic analysis tools, evaluating more complex programming tasks and real-world software systems, and investigating its applicability to other programming languages and state-of-the-art LLMs. We also plan to investigate the impact of prompt engineering strategies, model configurations, and alternative forms of analyzer feedback on repair effectiveness, and compare the proposed framework with existing automated program repair approaches.

Artifact Availability

A replication package is available at https://doi.org/10.5281/zenodo.10985772, providing the source code and generated assets for reproducing and validating the study.

Generative AI Use

Generative AI tools were used to support language editing and figure preparation. All scientific content was reviewed and approved by the authors, who take full responsibility for the manuscript.

Acknowledgments

This work was supported by CAPES No. 88887.313474/2026-00, through the BRICS Network University (BRICS-NU) Program, under Call No. 25/2025.

References

[1] Baleegh Ahmad et al. 2024. On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4043–4057.

[2] Jacob Austin et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021). https://arxiv.org/pdf/2108.07732v1.pdf

[3] Mauro Cazzaniga et al. 2024. Gen-AI: Artificial Intelligence and the Future of Work. *Staff Discussion Notes* 2024, 001 (2024). doi:10.5089/9798400262548.006

[4] Erik Derner and Kristina Batistič. 2023. Beyond the Safeguards: Exploring the Security Risks of ChatGPT. *arXiv preprint arXiv:2305.08005* (2023).

[5] Free Software Foundation. 2023. Static Analyzer Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Static-Analyzer-Options.html Accessed: August 11, 2026.

[6] 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.

[7] Maria Joseph Israel and Ahmed Amer. 2023. Rethinking data infrastructure and its ethical implications in the face of automated digital content generation. *AI and Ethics* 3, 2 (2023), 427–439.

[8] Matthew Jin et al. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*. 1646–1656. doi:10.1145/3611643.3613892

[9] Harshit Joshi et al. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140. doi:10.1609/aaai.v37i4.25642

[10] Raphaël Khoury et al. 2023. How Secure is Code Generated by ChatGPT?. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. doi:10.1109/SMC53992.2023.10394237

[11] Sam Lau and Philip Guo. 2023. From Ban it till we understand it to Resistance is futile: How university programming instructors plan to adapt as more students use AI code generation and explanation tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 106–121. doi:10.1145/3568813.3600138

[12] MITRE Corporation. 2024. CWE VIEW: Research Concepts (CWE-1000), CWE Version 4.20. https://cwe.mitre.org/data/definitions/1000.html Accessed: August 24, 2026.

[13] David Noever. 2023. Can large language models find and fix vulnerable software? *arXiv preprint arXiv:2308.10345* (2023).

[14] Hammond Pearce et al. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768. doi:10.1109/SP46214.2022.9833571

[15] Hammond Pearce et al. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356. doi:10.1109/SP46215.2023.10179420

[16] Neil Perry et al. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799. doi:10.1145/3576915.3623157

[17] Gustavo Sandoval et al. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2205–2222.

[18] Krzysztof Wach et al. 2023. The dark side of generative artificial intelligence: A critical analysis of controversies and risks of ChatGPT. *Entrepreneurial Business and Economics Review* 11, 2 (2023), 7–30.

[19] Jules White et al. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[20] Claes Wohlin et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[21] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. 2023. DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions. *arXiv preprint arXiv:2312.04730* (2023).

[22] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[23] Quanjun Zhang et al. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879* (2023).