A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback

Ummay Kulsum
North Carolina State University
Raleigh, USA
ukulsum@ncsu.edu

Haotian Zhu
Singapore Management University
Singapore, Singapore
htzhu@smu.edu.sg

Bowen Xu
North Carolina State University
Raleigh, USA
bxu22@ncsu.edu

Marcelo d'Amorim
North Carolina State University
Raleigh, USA
mdamori@ncsu.edu

ABSTRACT

Recent work in automated program repair (APR) proposes the use of *reasoning and patch validation feedback* to reduce the semantic gap between the LLMs and the code under analysis. The idea has been shown to perform well for general APR, but its effectiveness in other particular contexts remains underexplored.

In this work, we assess the impact of reasoning and patch validation feedback to LLMs in the context of vulnerability repair, an important and challenging task in security. To support the evaluation, we present VRpilot, an LLM-based vulnerability repair technique based on reasoning and patch validation feedback. VRpilot (1) uses a chain-of-thought prompt to reason about a vulnerability prior to generating patch candidates and (2) iteratively refines prompts according to the output of external tools (e.g., compiler, code sanitizers, test suite, etc.) on previously-generated patches.

To evaluate performance, we compare VRpilot against the state-of-the-art vulnerability repair techniques for C and Java using public datasets from the literature. Our results show that VRpilot generates, on average, 14% and 7.6% more correct patches than the baseline techniques on C and Java, respectively. We show, through an ablation study, that reasoning and patch validation feedback are critical. We report several lessons from this study and potential directions for advancing LLM-empowered vulnerability repair.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; • **Computing methodologies** → *Natural language processing*; • **Security and privacy** → *Software and application security*.

KEYWORDS

Automated Vulnerability Repair, Large Language Models

ACM Reference Format:

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 (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3664646.3664770

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3664770

1 INTRODUCTION

Automated vulnerability repair is an active field of research [3, 4, 13, 14, 16, 36, 41]. A variety of repair strategies have been proposed in the literature. Recent prior work [28, 36] applied large language models (LLMs) for vulnerability repair to mitigate the limitations of traditional non-LLM approaches, such as the inability to adapt to unseen circumstances or the difficulty in obtaining labeled data [27]. However, the effectiveness of these approaches is still questionable. We observe that there is an important *semantic gap* between what LLMs know and what they need to know to solve a challenging code-related task, such as vulnerability repair. Intuitively, the LLM knows little about the semantics of the code under analysis.

Recently, Xia et al. [38] proposed ChatRepair, an LLM-based technique for automated program repair. They demonstrate that ChatRepair can *bridge the semantic gap between the program under analysis and the LLM.* ChatRepair automatically repairs bugs by using (1) reasoning and (2) patch validation feedback. *Reasoning* enables the LLM to better understand the connection between the task and the code before producing answers whereas *Patch Validation Feedback* enables the LLM to refine its answer based on sensible static and dynamic information (e.g., compilation errors and tests failures due to violation of test case). Intuitively, ChatRepair tries to mimic how developers repair a bug in the real world by first attempting to understand the reasons for the problem (reasoning) and then iteratively refining the answers with the assistance of external tools (feedback).

This paper revisits the *reasoning-feedback* idea that Xia et al. introduced [38] in the context of vulnerability repair, which is a challenging class of problem in code repair. To enable that analysis, we present VRpilot, a LLM-based technique for vulnerability repair based on reasoning and feedback. In contrast to general code repair, VRpilot brings the kind of vulnerability (e.g., fix a "Heap Buffer Overflow" in a fragment of C code) into the LLM context for better reasoning. Recent prior work [20, 35, 40] have shown that LLMs are capable of performing complex tasks that require intermediate reasoning steps. VRpilot follows a design similar to those approaches to access such "reasoning" capability of the LLM. Complementary to reasoning, patch validation feedback helps the LLM to fix code that does not compile or code that breaks the program functionality [29]. Intuitively, the output of external tools provides guidance to the LLM remediate those problems [25]. **VRpilot iteratively incorporates the output of the compiler, test case, and also security sanitizer in prompts** under the assumption that the LLM

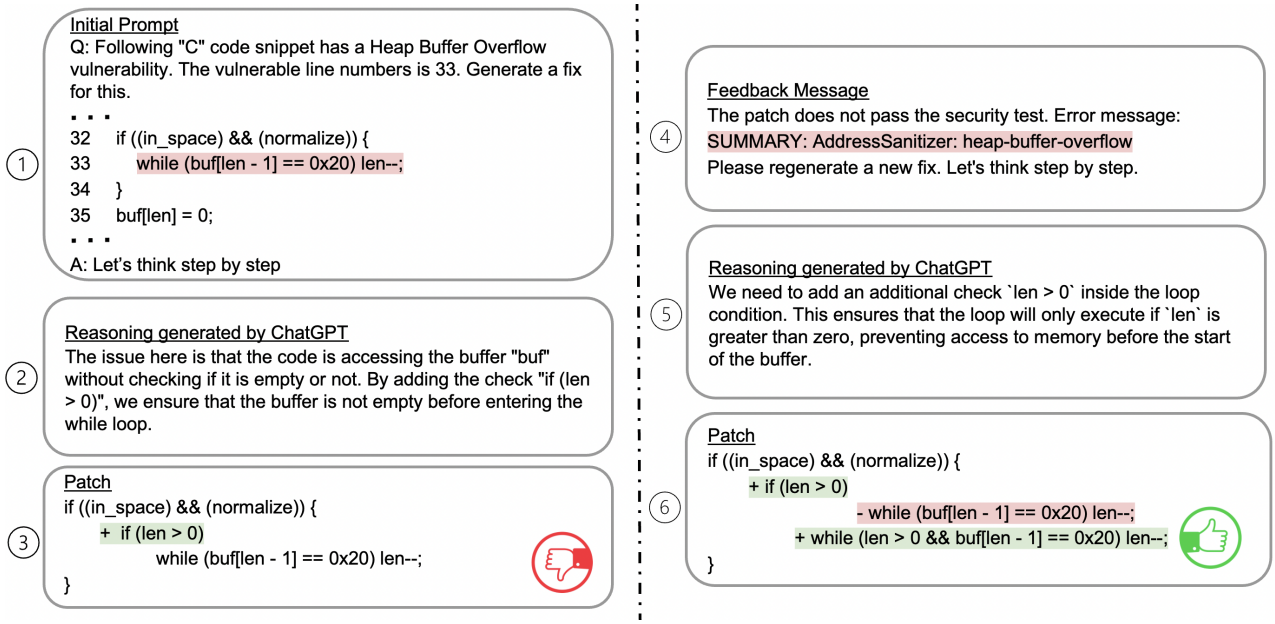

Figure 1: An example of correct patches generated by VRpilot. ① VRpilot queries ChatGPT using a prompt with a high-level description of the vulnerability ("Heap Buffer Overflow") and a fragment of the vulnerable code. The prompt also includes the trigger "Let's think step by step" to access ChatGPT's reasoning feature [20]. ② In response, ChatGPT explains the problem and suggests a solution. ③ VRpilot queries the model again combining the initial prompt and reasoning information, and a patch is produced. ④ However, this patch does not pass the security tests (👎). VRpilot leverages the output of external tools (e.g., compiler, functional and security tests) to circumvent the problem. In this case, VRpilot repeats the previous process after incorporating the error message generated by the address sanitizer (introduced by the compiler). ⑤ ChatGPT updates its reasoning and generates another patch. ⑥ This patch passes all tests (👍).

learns how to avoid them. It is worth noting that Xia et al. focuses on fixing bugs in general and it does not consider security tests.

Figure 1 illustrates VRpilot on a running example, consisting of a "Heap Buffer Overflow" vulnerability in file `parser.c` from Libxml2, a well-known software library for parsing XML documents.[1] VRpilot starts by requesting an explanation for the problem to the LLM. The answer from the LLM appears in the middle box on the left-hand side of the figure. Then, VRpilot uses that explanation to request a patch to the LLM. The answer from the LLM appears in the box with a "thumbs down" icon (👎). Unfortunately, although the explanation is coherent with the actual problem, the patch still fails the security test. As such, VRpilot extracts the error message from test execution logs and provides it to the LLM. VRpilot requests the LLM to reason about the problem again. Note from the explanation that, this time, the LLM spots the need to introduce an extra check in the loop conditional. VRpilot uses the revised explanation to request another patch and the LLM produces a plausible patch this time. The answer from the LLM appears in the box with a "thumbs up" icon (👍). Indeed, we find that this patch is equivalent to the ground truth, which replaces the original condition in the while statement with the condition `len > 0 && buf[len -1] == 0x20`.

We conduct three experiments to assess the impact of reasoning and feedback in LLM-based vulnerability repair.

First, we evaluate the effectiveness of CodexVR [28],[2] a recently-proposed technique for LLM-based vulnerability repair. CodexVR uses Codex [26], a coding-specific LLM developed by OpenAI, but it is no longer available. For that reason, we evaluate CodexVR with ChatGPT and various prompts. This experiment evaluates both robustness and effectiveness of CodexVR. Based on our results, we find that CodexVR with ChatGPT is capable of generating more plausible patches than with Codex by a large margin, i.e., 20.2%, showing that the performance of CodexVR does not degrade. However, we find that, despite the improvement, CodexVR only generates a relatively small amount of plausible patches. On average, only **29.6%** of the patches it generates are plausible.

Second, we compare VRpilot against an optimized version of CodexVR finding that our proposed technique outperforms the baseline. In particular, we find that VRpilot generates, on average, **14%** and **7.6%** more correct patches than the baseline technique on C and Java datasets, respectively.

Third, we conduct an ablation study to measure the contribution of each component of VRpilot: reasoning and feedback. We find that both components contribute to VRpilot's performance. For example, considering the percentage of plausible patches produced, we find that the performance of VRpilot falls from **64%** to **34%** and to **35%** when feedback is disabled and when reasoning is disabled, respectively. These results show that both components contribute to the performance of VRpilot and the performance of the full-fledged combination achieves the best or equally the best performance in 90% cases.

We learned several lessons from this work. For example, our results indicate that more work is needed to devise automated techniques to mine code contexts to be incorporated in LLM prompts. It is worth noting that some vulnerability repair tasks require domain

[1] https://gitlab.gnome.org/GNOME/libxml2/-/commit/6a36fbe
[2] Pearce and others [28] did not name their technique. We use the name CodexVR for their technique, reflecting the use of the Codex model applied to vulnerability repair.

knowledge (e.g., design choices, environment factors, etc.), posing a challenge in automating this task. It is important to differentiate those cases in benchmarks. As seed for future work, our experience with patch validation suggests that integrating large language models to improve productivity in manual inspection –as opposed to replacing humans altogether– may offer a good balance between accuracy and scalability.

Our work makes the following contributions: (1) We examine the idea of *reasoning-and-feedback* in LLMs for vulnerability repair by presenting VRpilot, a tool follows the spirit of the idea; (2) We evaluate VRpilot on datasets of C and Java programs showing that the approach outperforms the state-of-the-art baselines; (3) We discuss lessons learned and the implications of our study; (4) We provide the replication package to facilitate future research: http://tinyurl.com/vrpilot-artifacts.

2 METHODOLOGY

In this section, we first introduce the problem formulation and then detail the design of the method.

2.1 Problem Formulation

The primary objective of automated vulnerability repair is to fix software vulnerabilities without human intervention. A repair may be unacceptable because the code fails the *security tests* introduced by the vulnerability detection tool (e.g., ASAN [31] for memory safety issues and UBSAN for integer overflows and division by zero [32]). Likewise, a patch may be unacceptable if the patched code does not pass the *functional tests*.

Vulnerability Repair (VR): Given a vulnerable program P, the corresponding security specification SS that makes the vulnerable program P fail, the functional specification FS, we define VR as a function taking the triple (P, SS, FS) as input and producing a patch for P as output. More precisely, the problem of VR is to find a patch P' (i.e., a safe variant of P) that passes both SS and FS. We refer to the patch P' as *plausible patch*. In the literature, some of the prior works (such as [23]) on VR assume example patches are available. Therefore, they define the example patches as part of the input of the task. In this work, considering VR is often time-critical, we follow a more realistic task definition in this work, *zero-shot setting*, i.e., none of the patch examples is given.

2.2 Overview

An LLM (e.g., InCoder [12], Polycoder [15], and ChatGPT [34]) is trained on massive amounts of data using general-purpose tasks[3]. Users interact with an LLM through a *prompt*, which describes a task. For Software Engineering tasks, a prompt typically includes code and text [2]. Users of LLMs often can control the *temperature*, a variable that sets the level of (un)predictability of answers.

VRpilot is an LLM-based vulnerabilty repair technique that employs *reasoning* and (patch validation) *feedback* in producing plausible patches. Conceptually, reasoning enables the LLM to better understand the task that needs to be solved prior to solving the task; it helps an LLM think logically and draw a conclusion based on premises [17] (Section 2.3). Feedback helps an LLM to obtain information from external sources [25]; information that is not present in the models' training data. These two mechanisms contribute to bridging the semantic gap between the LLM and the programs under analysis (Section 2.4).

[3]For example, predicting a masked token and predicting the following sentence, which is the approach used in ChatGPT [34].

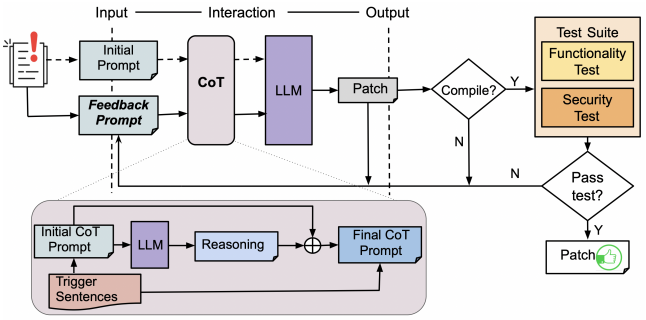

Figure 2: Overview of VRpilot: The process begins with an initial prompt based on vulnerability information, passed to the Chain-of-Thought (CoT) block. The CoT block adds a trigger sentence and queries the LLM for reasoning. The final CoT prompt, combining initial prompt, reasoning, and another trigger sentence, is used by the LLM to generate a repair patch. The patch is compiled and tested. If the patch passes, it is considered a plausible patch. If not, VRpilot refines it using feedback and repeats the steps.

Figure 2 details the workflow of VRpilot. The input, highlighted with an exclamation mark, consists of information about the vulnerability, such as the statements in the vulnerable function and description of vulnerability (e.g., "heap buffer overflow"). The initial prompt includes the description of the vulnerability and the vulnerable function. VRpilot then obtains a patch through a zero-shot chain-of-thought prompting [20], highlighted in the box at the bottom-left corner of the figure (Section 2.3). If the patch is not compilable, VRpilot updates the prompt with an indication of the compilation error and requests ChatGPT for a better solution. Otherwise, VRpilot runs the test suite, including both functional and security tests. If a patch passes both tests, VRpilot reports this plausible patch and the process terminates. Otherwise, VRpilot extracts the useful part of the error message, and updates the prompt accordingly. VRpilot repeats these steps until reaching the maximum number of feedback iterations.

2.3 Chain of Thought

The optimal performance of LLMs relies on the proper design of prompts. Prior studies have shown that various LLMs possess the capacity to generate reasoning and effectively address complex multi-step reasoning tasks with the help of advanced prompt techniques [20, 35, 39]. For example, Wei et al. [35] demonstrated that providing a series of intermediate reasoning steps explained through a few examples in the prompt increases the performance of LLMs in solving complex mathematical and commonsense tasks, naming that approach as *chain-of-thought prompting*. On followup work, Kojima et al. [20] showed that the success of the chain of thought prompting depends on the explanation in a few-shot manner and proposed a new zero-shot approach where the explanations are generated by the LLM.

Vulnerability repair is undeniably a complex problem that may require intermediate reasoning steps, like comprehending the code, comprehending the vulnerability present in the code, and devising a suitable solution. Consequently, we hypothesize that utilizing

LLMs in expressing the rationale for these intermediate steps can boost the performance of LLMs in vulnerability repair.

Kojima et al. [20] proposed a prompt template to be used in solving a given task through zero-shot chain-of-thought (CoT) prompting. It involves two stages of prompting: (1) reasoning extraction and (2) answer extraction. The initial CoT prompt takes the form $X' = Q : [X]\ A : [T]$, where $[X]$ is a slot for a problem X and $[T]$ is a sentence that triggers reasoning generation, such as "Let's think step by step". The symbols Q : and A : initiating a sentence is a short form of "Question" and "Answer", respectively. It is assumed that the LLM is trained on completion tasks; as such, it will generate the answer that follows the triggering signal for reasoning T. The final CoT prompt takes the form $[X'][Z][A]$, where $[X']$ is a slot for the initial CoT prompt, $[Z]$ is for the response generated from the initial CoT prompt, and $[A]$ is for the trigger sentence to extract the solution of the problem.

In our case, we define X as a vulnerability repair task, the initial prompt contains the vulnerability information, vulnerable code block, and an instruction to repair the vulnerability. For the trigger sentence $[T]$, we choose to use *"Let's think step by step"* as it achieves the best performance in multiple reasoning task datasets [20]. Then we construct the final prompt by combining the initial prompt, the reasoning and appending a trigger sentence "Therefore the fixed code is" to extract the generated repair patch.

2.4 Patch Validation Feedback

ChatGPT is trained with reinforcement learning and human feedback to follow conversations. We leverage this observation by iteratively providing previously unused compiler and test suite error messages as conversation feedback. These feedback messages assist ChatGPT in refining incorrect patches. Intuitively, error messages are designed to guide developers to fix vulnerabilities. ChatGPT generates code using the code context provided in the prompt and its internal knowledge. However, it lacks the external knowledge, i.e., security and functionality requirements of the project. VRpilot provides this external information through feedback and guides ChatGPT to iteratively improve generated patches. Even though incorrect patches might seem like failures, error logs contain valuable information that can help understand the cause of the problem and subsequently modify the patch to fix the vulnerability. To construct comprehensive feedback, we sequentially leverage 3 types of error messages to instruct ChatGPT effectively, namely:

(1) Feedback from compilation test. When a generated patch does not compile, VRpilot parses the compilation error message and extract the relevant error messages. The extracted relevant error messages are then selected based on their proximity to the vulnerable lines of code within a range of 100 lines. Hence, VRpilot focuses on the specific errors directly related to the problematic code, allowing for more accurate feedback.

(2) Feedback from functionality test. VRpilot runs functional tests on the generated patch to ensure it implements the desired functionality. For that reason, VRpilot automatically executes the test suite included in the projects. If the patch fails to pass these functionality tests, VRpilot parses the corresponding log to extract the names of the failed test cases that carry the high-level semantic of the testing purpose. These failed test cases are then used as additional feedback for ChatGPT, providing further information about the specific functional deficiencies of the patch.

(3) Feedback from security test. Similarly, we run security tests by enabling different sanitizer flags (ASAN/UBSAN) according to the original setting of the projects. The security error message is parsed from the log generated by the test and used as feedback to guide ChatGPT.

Based on the above feedback messages, VRpilot constructs the initial CoT prompt for feedback iteration (feedback prompt) with chain-of-thought. We design the feedback prompt as $F = Q : [X][C][E]A : [T]$ where $[C]$ is the slot for code changes suggested by LLM in previous iteration; $[E]$ is the slot for the feedback error message described above; $[X]$ and $[T]$ are slots for the vulnerability repair task and trigger sentence respectively as described in section 2.3. The final CoT prompt for feedback iteration is constructed as $[X'][Z][A]$ where $[X']$ is a slot for the feedback prompt F and $[Z]$ and $[A]$ are slots for the reasoning and trigger sentence for answer extraction respectively as describe in section 2.3. The above iterative process aims to enhance the quality and accuracy of the generated patches by incorporating information from compilation errors, failed functionality tests, and security tests.

2.5 Implementation

VRpilot is implemented in Python and it accesses ChatGPT via API provided by OpenAI [33]. We use *gpt-3.5-turbo* as the underlying model for ChatGPT. VRpilot initializes ChatGPT with a system message; "You are a chatbot for vulnerability repair" for the vulnerability repair task. We iterate through 5 different temperature levels (0.0, 0.25, 0.5, 0.75, 1.0). It is worth noting that we did not identify an optimal temperature that consistently outperformed others across all scenarios in our experiments. Furthermore, we run the feedback iteration loop 4 times. Therefore, in total, we query 5 times in each of the temperatures to generate a patch. Specifically, we add line number at the beginning of each line of the extracted function block starting from 1. We add the line numbers with the code block to instruct ChatGPT regarding the vulnerable line numbers at the beginning of the prompt. It is inspired by the best practices for prompt engineering, suggesting adding instructions at the beginning of the prompt [30].

3 EXPERIMENTAL SETTING

In this section, we first introduce the state-of-the-art approach as our baseline. And then we provide details of our three research questions. Last, we describe the evaluation metrics and the dataset used in our experiment.

3.1 Comparison Baseline

VRpilot is most related to CodexVR, a vulnerability repair technique using zero-shot learning [28]. Unlike VRpilot, CodexVR does not leverage the output of test runs or employ a chain of thought prompt design to improve LLM performance. It is worth noting that the original version of CodexVR uses the OpenAI's Codex model (codex-cushman-001) to produce patches. However, Codex has been deprecated[4] in March 2023 and, for that reason, we cannot replicate the result of the paper as is [28]. For a fair comparison, we configure CodexVR with GPT-3.5, which is a more recent model developed by OpenAI. Moreover, there are six different prompts investigated in [28] as shown in Table 1. Therefore,

[4]https://platform.openai.com/docs/guides/code

Table 1: Prompt templates used in CodexVR [28].

ID	Description
n.h.	No Help - deletes the vulnerable code/function body and provides no additional context for regeneration.
s.1	Simple 1 - deletes the vulnerable code/function body and adds a comment 'bugfix: fixed [error name]'.
s.2	Simple 2 - deletes the vulnerable code/function body and adds a comment 'fixed [error name] bug'.
c.	Commented Code - After a comment 'BUG: [error name]', it includes a 'commented-out' version of the vulnerable code/function body followed by the comment 'FIXED:'. After this it appends the first token of the original vulnerable function.
c.a.	Commented Code (alternative) - same as c. , but commented in the alternative style for C /* and */ rather than //
c.n.	Commented Code (alternative), no token - same as c.a., but with no 'first token' from vulnerable code.

two configurations potentially impact the approach's performance, i.e., the base model and prompt selection.

3.2 Research Questions

Our experiment is designed to answer the following research questions (RQs):

RQ1: What is the impact of the base LLM and prompt selection on the state-of-the-art technique for VR?

Rationale This question aims to investigate the impact of different prompts and LLMs on the performance of CodexVR. In particular, we want to assess if the results do not degrade when using the newer model but for general purpose in a particular task, in our case, i.e., vulnerability repair. It is also important to observe if the opposite occurs —i.e., if the results of CodexVR configured with the new model improve significantly– as the motivation for proposing a new technique would be weaker in that case.

Setting. To answer this question we reproduced the setup documented in the CodexVR paper [28]. The only difference between the original experiment and our experiment is the use of ChatGPT (instead of Codex) as the base LLM. The CodexVR paper reported results on various LLMs from the Codex family, with codex-cushman-001 performing best overall. For that reason, and in the interest of space, we restricted the comparison of ChatGPT to codex-cushman-001. We evaluate the performance of CodexVR on every combination of base model, and all the six prompts proposed in [28]. For each combination, we query the model 50 times, as in the CodexVR evaluation, and collect the performance on the evaluation metrics (Section 3.3).

RQ2: How does VRpilot compare against the baseline?

Rationale This question aims to assess whether or not our proposed approach outperforms the optimal setting of our comparison baseline, CodexVR, identified through RQ1.

Setting To establish a strong comparison baseline, we use an *idealized optimal version* of CodexVR that selects the best prompt and base model identified in RQ1. We compare such an idealized version of CodexVR with VRpilot using the standard metrics from the literature.

RQ3: How do the different components affect the performance of VRpilot?

Rationale The goal of this question is to measure the contribution of different components of VRpilot on its performance.

Setting VRpilot incorporates two features: chain of thought (C) and (patch validation) feedback (F). We compare the performance of all four feature combinations of VRpilot: CF, $C\neg F$, $\neg CF$, $\neg C\neg F$. For $C\neg F$, VRpilot does not execute the feedback iteration, i.e., none of the feedback messages is fed from the compiler and test suite. For $\neg CF$, VRpilot does not run the CoT block but it performs the feedback iteration. In this setting, none of the trigger sentences have been added at the end of the initial and feedback prompt.

3.3 Metrics

We use 3 standard metrics from the literature: (1) percentage of compilable patches and (2) percentage of plausible patches (i.e., patches that are compilable and also pass the functional and security tests) and (3) percentage of patches which are semantically equivalent to the ground truth (modulo manual inspection). Functional tests are those available in the test suite. Security tests correspond to the same tests augmented with runtime monitors — referred to as sanitizers— to check violations of a security-related property, such as buffer overflows. These monitors (or sanitizers) are automatically introduced by compilers with specific flags (e.g., `-fsanitize=address` [31]).

3.4 Datasets

We evaluate VRpilot on multiple datasets in two popular programming languages, C and Java.

Dataset in C. We consider ExtractFix [1], it has been used in previous studies [14, 28]. This dataset consists of 10 real-world Common Vulnerabilities and Exposures (CVEs) from public open-source projects in C. All the examples include developer patches localized within a single file and have comprehensive test suites.

Dataset in Java. We also consider two datasets of Java programs (VjBench [36] and Vul4j [3]) including a total of 50 vulnerabilities with CVE labels. Due to the page limit, please refer to the original papers for more details about these datasets.

4 RESULTS

This section reports the results of the experiments to answer the research questions listed in Section 3.2.

4.1 RQ1: What is the impact of the base LLM and prompt selection on the state-of-the-art technique for VR?

Table 2 summarizes the results of CodexVR when configured with different LLMs (Codex and ChatGPT) and prompts[5]. We use the C dataset in this experiment by following [28]. We compare the performance of CodexVR with two base LLMs and all the six prompts that Pearce et al. proposed in [28].

Results show that among all the six prompts, ChatGPT performs the best when using the "c.a." prompt, generating 29.6% plausible patches on average. Differently, CodexVR performs the best when using the "c." prompt, yielding an overall average of 9.4% plausible patches. Overall, ChatGPT generates **+20.2%** more plausible answer than CodexVR in their respective best settings. Furthermore, we find that ChatGPT consistently produces more plausible

[5] In interest of space, we provide detailed results on each case in our replication package.

Table 2: *Results of RQ1.* Evaluation of CodexVR with different LLM models and prompts. We query each triple of example, model, and prompt 50 times. x/y indicates x % plausible patches out of y % patches that compile ((higher values are better). The last column shows if any prompts with a given model report a plausible patch.

EF#	Model	Prompt						Pass?
		n.h	s.1	s.2	c.	**c.a**	c.n	
Average	Codex	1.8/15.4	0.6/10.6	1.0/14.4	9.4/49.0	2.8/55.4	5.0/51.8	7/10
	ChatGPT	4.4/24.6	3.2/23.2	2.4/17.4	25.4/44.2	**29.6**/**50.6**	22.0/62.2	**10**/**10**

Table 3: Percentage of Compilable and Plausible Patches on C Dataset

EF#	CodexVR* on ChatGPT			VRpilot		
	Compilable%	Plausible%	Pass?	Compilable%	Plausible%	Pass?
Average	55.6	33.4	10/10	90.4	63.8	10/10

Table 4: Percentage of Correct Patches on C Dataset

EF#	CodexVR* on ChatGPT	VRpilot
	% Correct/ Inspected	% Correct/ Inspected
EF01	90	**100**
EF02_01	70	70
EF08	0	0
EF15	0	0
EF17	**100**	60
EF18	90	**100**
EF22	0	**70**
Average	50	57

patches in all the prompts. These results suggest that CodexVR performs better when configured with ChatGPT instead of Codex.

However, our results also show that the performance of ChatGPT is still quite low with only 29.6% plausible answers in the best case, which is far from desirable. Based on the column Pass?, We also observe that CodexVR with ChatGPT can generate plausible for all the cases while CodexVR with Codex fails to generate plausible patches for 3 cases. It is worth noting that the count of crosses, by definition, is a lenient performance indicator for techniques. Overall, our results indicate that simply using a newer LLM is insufficient to repair vulnerabilities effectively.

Answers to RQ1: The use of a newer yet general-purpose LLM, i.e., ChatGPT instead of Codex, does improve the results of CodexVR. However, the performance of CodexVR with ChatGPT is still far from desirable. This indicates that the adoption of a larger general-purpose language model alone is unlikely to be sufficient to effectively repair vulnerabilities.

4.2 RQ2: How does VRpilot compare against the baseline?

4.2.1 *Evaluation on C Dataset.*

Setting. Despite the improvement of CodexVR when configured with ChatGPT instead of Codex, results for RQ1 showed that CodexVR still performs poorly overall. Motivated by those results, RQ2 compares our proposed technique, VRpilot (Section 2), with an *idealized optimal version* of CodexVR that is configured with ChatGPT and selects the best prompt for a given example, i.e., the prompt that optimizes the metrics for that example (Section 3.3).

Analysis. Table 3 shows the results of comparing VRpilot with the baseline on the C dataset. We use a star (*) to emphasize the use of such an idealized version of CodexVR. We observe that, both VRpilot and the baseline generates plausible answers in all the cases. **However, VRpilot can generate 63% more compilable patches and 91% plausible patches than the baseline**. Ideally, all generated patches should be compilable. However, results shows that nearly 10% of generated patches are not. Furthermore, we manually check the correctness of the generated plausible patches. However, manual validation is known be expensive. Hence, we select 7 (out of 10) cases from C dataset in which VRpilot have produced more than 10 plausible patches. For each case, we randomly select at most 10 patches among the generated plausible patches. One labeler carefully examines the selected patches and then discusses the result with the second labeler to derive the final decision on the correctness. Table 4 presents the result of manual inspection. We find that **VRpilot produces 14% more correct patches compared to the baseline**. However, our results indicate that there is still significant room for improving patch correctness.

Table 5: Performance of VRpilot on fixing Java vulnerabilities. The 'Plausible' column shows the number of vulnerabilities with at least one patch passing test cases, while the 'Correct' column shows those with at least one patch semantically equivalent to human patch

	VJFix		VRpilot	
	Plausible	Correct	Plausible	Correct
VjBench (15)	4.6	4.0	**6.0**	**5.0**
Vul4j (35)	10.9	6.2	**14.0**	**9.0**
Total (50)	15.5	10.2	**20.0**	**14.0**
Compilation Rate (%)	79.7		**86.0**	

4.2.2 *Evaluation on Java Dataset.*

Settings. We evaluate VRpilot on the VjBench [36] and Vul4j [3] datasets of Java vulnerabilities. We follow a similar experimental procedure that Wu et al. [36] use to evaluate the ability of the LLMs from the Codex family to generate correct patches. Wu et al. report that the davinci-002 model (we refer to it VJFix) performed the best, therefore we reuse the same configuration setting. Specifically, we run VRpilot by querying the LLM with a temperature 0.6 and generating 10 patches for each vulnerability. It is worth noting, however, that we run our pipeline once –for the sake of time and computational cost– whereas authors of VJFix run their pipeline 25 times, taking averages of the metrics of interest.

Analysis. Table 5 shows the results comparing VJFix and VRpilot. We observe an increase from 79.7% to 86% (6.3% gain) in terms of compilation rate and from 15.5 to 20.0 (9.0% gain) in terms of plausible patch when using VRpilot instead of the baseline. Finally, considering the number of cases in which a semantically correct patch is generated (out of 50), we observe an increase from 10.2 to 14 (7.6% gain). As usual, semantic correctness is based on manual inspection where one of the authors evaluates and discusses with other authors if the patch is behavioural-equivalent to the human patch. It is worth noting that there are 4 cases where VRpilot generates plausible patches for which we were unable to decide correctness because of the complexity of the code. For instance, the human patch for fixing vulnerability Vul4J-8 adds two new conditions to break one loop.[6] VRpilot generates a plausible patch for this vulnerability by adding different loop condition.[7] Although the patch seems logically correct, it is hard to fully ascertain without domain knowledge. We further discuss this on Section 5.1

[6]Human patch: https://github.com/apache/commons-compress/commit/4ad5d80a6272e007f64a6ac66829ca189a8093b9

[7]Diff between original and patch: https://www.diffchecker.com/7aePjrvI/

Table 6: *Results of RQ3.* The leftmost column shows the impact of VRpilot's prompt without any of the components (CoT and Feedback). The rightmost column shows the aggregated impact of all component. The middle two column show the impact of each component in isolation.

Id	w/o Chain of Thought				w/ Chain of Thought			
	w/o Feedback		w/ Feedback		w/o Feedback		w/ Feedback	
	Compilable	Plausible	Compilable	Plausible	Compilable	Plausible	Compilable	Plausible
EF01	60	0	96	16	76	68	88	80
EF02_01	100	8	100	16	96	56	100	74
EF07	80	8	92	44	52	20	88	56
EF08	56	4	88	32	60	20	76	64
EF09	92	28	100	32	68	36	96	88
EF15	92	0	100	0	80	4	100	40
EF17	92	92	96	96	88	32	96	92
EF18	96	96	100	100	92	80	100	100
EF20	100	0	100	4	76	0	92	4
EF22	96	0	100	12	56	28	68	40
Average	86	23	97	35	74	34	90	64

Answers to RQ2: VRpilot consistently outperforms the baselines on both C and Java datasets in terms of compilable, plausible and correct patch rate. However, generating the correct patch remains challenging.

4.3 RQ3: How do the different components affect the performance of VRpilot?

Table 6 shows the results of the four different combinations of these two components. This RQ considers the percentage of plausible patches as our main evaluation metric. We observe that the baseline without both Chain-of-Thought and Feedback produces 23% plausible patches. Column "Plausible" under "Without Chain-of-Thought" and "Without Feedback" shows that information. We construct the prompt for this combination by removing the chain-of-thought and the feedback components from VRpilot.

Adding chain-of-thought to the baseline increases the rate of plausible patches generated from 23% to 34% (+47% gain). This result shows the positive impact of the chain-of-thought prompt. Adding feedback to the baseline increases the rate of plausible patches from 23% to 35% (+52% gain). When combining chain of thought with feedback, the rate of plausible patches increases to 64%, reflecting the importance of these two components to VRpilot.

Results show that combining Chain-of-Thought with Feedback (i.e., our method VRpilot) yields the best result. In this setting, the percentage of plausible patches increases to 64%. Moreover, the distribution of green cells across the table suggests that Feedback and Chain-of-Thought synergistically cooperate. In four of the cases (EF02_01 [6], EF08 [11], EF09 [8], and EF15 [7]), the rate of plausible answers in the full-fledged combination is higher than the sum of the rates from the combinations with a single component.

Answers to RQ3: Each component of VRpilotis essential to its performance and the full-fledged combination achieves the best or equally the best performance, suggesting that the components synergistically cooperate.

5 DISCUSSION

5.1 Lessons and Implications

Lesson #1 The use of a reason-feedback mechanism can enhance the ability of Large Language Models (LLMs) in repairing vulnerabilities, though addressing complex vulnerabilities remains a significant challenge. This observation is supported by the result of RQ3 (Section 4.3). We found that combining a chain-of-thought approach for reasoning with error messages from tests for feedback can improve the performance of LLMs. Specifically, our manual verification confirmed all plausible patches randomly selected for vulnerabilities EF01 [9] and EF18 [10] are correct. These vulnerabilities, categorized as *out-of-bounds* and *null pointer dereference*, respectively, are often resolved by adding a simple conditional check. **However, the complexity of a vulnerable code fix cannot be determined only by the corresponding vulnerability category**. For example, in our experiments, EF15 [7] is also categorized as *out-of-bounds* vulnerability. However, as shown in Table 4, none of the randomly selected plausible patches were found to be correct upon manual examination. Figure 3 presents the human-generated patch for EF15. To fix this vulnerability, it requires a significant amount of code change. A new variable, `curLength`, is created and initialized. That change requires a deeper understanding about the data structure `xmlParserCtxtPtr`.

parser.c

```
...  ...  @@ -9825,6 +9825,7 @@ static void
9825 9825  xmlParseEndTag2(xmlParserCtxtPtr ctxt, const xmlChar *prefix,
9826 9826                 const xmlChar *URI, int line, int nsNr, int tlen) {
9827 9827      const xmlChar *name;
     9828 +    size_t curLength;
9828 9829
9829 9830      GROW;
9830 9831      if ((RAW != '<') || (NXT(1) != '/')) {
...  ...  @@ -9833,8 +9834,11 @@ xmlParseEndTag2(xmlParserCtxtPtr ctxt, const xmlChar *prefix,
9833 9834      }
9834 9835      SKIP(2);
9835 9836
9836      -    if ((tlen > 0) && (xmlStrncmp(ctxt->input->cur, ctxt->name, tlen) == 0)) {
9837      -        if (ctxt->input->cur[tlen] == '>') {
     9837 +    curLength = ctxt->input->end - ctxt->input->cur;
     9838 +    if ((tlen > 0) && (curLength >= (size_t)tlen) &&
     9839 +        (xmlStrncmp(ctxt->input->cur, ctxt->name, tlen) == 0)) {
     9840 +        if ((curLength >= (size_t)(tlen + 1)) &&
     9841 +            (ctxt->input->cur[tlen] == '>')) {
9838 9842              ctxt->input->cur += tlen + 1;
9839 9843              ctxt->input->col += tlen + 1;
9840 9844              goto done;
```

Figure 3: A Complex Patch for EF15 *out-of-bounds* vulnerability

Implication #1: Context is the key for LLM in vulnerability repair. Additional information, such as vulnerability description and test cases outcomes, is helpful but insufficient for fixing complex vulnerabilities.

Lesson #2 Fixing vulnerabilities that are inherently linked to project design poses a significant challenge. An example of this is EF08 (CVE-2017-7601 [24]), a vulnerability arising from LibTIFF's handling of undefined behavior "shift exponent too large for 64-bit type long." Remote attackers can exploit this vulnerability to create a denial of service attack (making the application to crash) or possibly other unspecified behavior via a crafted image. A feasible solution to mitigate this vulnerability involves implementing a condition that checks whether the shift exponent size surpasses a certain bit threshold. Aligning with standard JPEG specifications, thresholds of 8 or 12 bits per sample are often considered. However, an important part of the human patch for EF08 [11] is adding a line of code `if(td->td_bitspersample > 16)`, the threshold is intentionally set to 16 bits. The choice of this threshold value reflects a design decision by the developers, influenced by factors such as their application's specific requirements, the anticipated range of TIFF files to be processed, and their strategy for balancing flexibility with safety and adherence to standards. This vulnerability underscores the complex challenge of addressing security issues that necessitate consideration of the project's design context. Moreover, we find that the key reason of VRpilot fails to generate

a correct patch is because it is "misled" by the vulnerability description, i.e., "shift exponent too large for 64-bit type long." Patches generated by VRpilot set the threshold to 64 instead 16 bits. Our finding reveals the challenge that **a deeper understanding on project/environment-related information beyond vulnerability description may be required for repairing complex vulnerabilities**.

Implication #2: Domain knowledge (e.g., project design) is critical to repair vulnerabilities. However, extracting this information is challenging. Investigating these aspects through LLMs presents a promising avenue for future research.

Lesson #3 The pressing need for a comprehensive and high-quality dataset for vulnerability repair is evident. Constructing such a dataset is a complex task. It should encompass both functionality and security tests to ensure thoroughness. Additionally, the dataset must be designed for easy replication of the execution environment. For instance, in our work with the ExtractFix dataset, Docker was utilized to establish a consistent executable environment. Furthermore, comprehensive documentation of the dataset's usage is crucial. Currently, locating datasets in vulnerability repair with *functionality-and-security test offered*, *well-documented*, *reproducible*, *well-maintained*, and *scalable* is a formidable challenge. These difficulties are similarly highlighted in a recent study from Croft et al. [5]. Therefore, addressing these challenges is essential for advancement in this field and significantly reducing the engineering burden for researchers.

Implication #3: High-quality datasets of vulnerability repair are rare. We urgently advocate more community involvement in developing reliable, reproducible and easy-to-use datasets with minimum human efforts.

Lesson #4 Fully manual patch validation for vulnerable code sometimes can be extremely difficult. This process, while crucial for distinguishing plausible from correct patches, demands highly skilled labelers with not only extensive programming experience but also a profound understanding of specific vulnerabilities. Such expertise makes it difficult to find suitable candidates. Moreover, comprehending vulnerabilities, often subtle and deliberately crafted by attackers, tends to be more complex than understanding general bugs. Additionally, while human-generated patches can offer insights into the logic of fixes, they sometimes present their own difficulties in interpretation. For instance, the human patch for the *Divide By Zero* vulnerability (EF09) [8] exemplifies this. It needs deep comprehension of the code context, including specific variables (`horizSubSampling` and `vertSubSampling`) and function calls (`usage(-1)`). This complexity adds another layer of challenge to the manual validation process.

Implication #4: Manually inspecting the correctness of a generated security patch is expensive. Utilizing large language models for semi-automatic patch inspection offers potential promise for achieving an optimal balance between accuracy and scalability.

5.2 Threats to Validity

The first threat to validity comes from the correctness of the plausible patches compared with the reference developer patch. To address this threat, we carefully examined and discussed each patch following prior work [19, 28, 37, 38]. However, manually measuring the correctness is expensive and sometimes requires domain specific knowledge. We sought second opinions and conservatively labeled patches as non-equivalent, when needed. The second threat to validity comes from the potential data leakage issue of developer patches being part of the original training data of ChatGPT. ChatGPT is not open-sourced and can only be accessed through API, so we cannot verify its training data. However, our approach and the baseline (RQ2) are based on the same version of ChatGPT, and our approach outperforms the baseline. This improvement gain is doubtful to be by memorizing training data.

6 RELATED WORK

Researchers have proposed vulnerability repair techniques by leveraging recent advances in deep learning. For instance, Chen et al. [4] proposed VRepair based on transformer and transfer learning. They showed that pre-training improves repair performance compared to training from scratch. Fu et al. [13] propose VulRepair, a technique that uses sub-word tokenization and pre-training for vulnerability repair, and show that that VulRepair outperforms VRepair. Huang et al. [18] applied pre-trained models for vulnerability repair to overcome the shortcomings of learning-based APR techniques. They demonstrated that these pre-trained models outperform learning-based APR techniques (e.g., CoCoNut [22] and DLFix [21]) and more data-dependent features help repair complex vulnerabilities. More recently, Wu et al. [36] proposed a Java vulnerability dataset, VJBench, by enhancing Vul4J [3]. They demonstrate that existing LLMs and APR models fix very few Java vulnerabilities. These works use large language models trained for code-related tasks. In contrast to prior work, this paper focuses on assessing effectiveness –for the task of vulnerability repair– of using a general-purpose LLM (e.g., ChatGPT) in combination with techniques that have been shown effective to reduce the LLM-to-code semantic gap.

7 CONCLUSION

This paper examines the idea of combining reasoning and patch validation feedback for vulnerability repair. The idea was originally proposed by Xia et al [38] in the general context of automated program repair. We present VRpilot, a method that queries the LLM first to reason about the code's specific vulnerability and then to respond with a repair patch using a chain of thought prompt. Furthermore, VRpilot iteratively refines the incorrect patch using error messages produced by the compiler and test suite as feedback to the LLM. We evaluate VRpilot on multiple datasets for Java and C. The results we obtained indicate that the idea of using reasoning and patch validation is still valid in the context of vulnerability repair, however, we find that the performance of LLM for this task still is far from desirable. This paper discusses the lessons we learned from this study and highlights the challenges for advancing LLM-based vulnerability repair.

ACKNOWLEDGMENTS. This work was supported and funded by the National Science Foundation Grant No. 2026928. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of any of the funding organizations.

REFERENCES

[1] 2023. ExtractFix Benchmark Download Link. https://drive.google.com/drive/folders/1xJ-z2Wvvg7JSaxfTQdxayXFEmoF3y0ET.

[2] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). https://doi.org/10.48550/arXiv.2206.01335 arXiv:2206.01335

[3] 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*. 464–468.

[4] 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.

[5] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[6] CVE-2014-8128. [n. d.]. EF0201. https://github.com/vadz/libtiff/commit/3206e0c

[7] CVE-2016-1838. [n. d.]. EF15. https://gitlab.gnome.org/GNOME/libxml2/-/commit/db07dd6

[8] CVE-2016-3623. [n. d.]. EF09. https://github.com/vadz/libtiff/commit/bd024f0

[9] CVE-2016-5321. [n. d.]. EF01. https://github.com/vadz/libtiff/commit/d9783e4

[10] CVE-2017-5969. [n. d.]. EF18. https://gitlab.gnome.org/GNOME/libxml2/-/commit/94691dc8

[11] CVE-2017-7601. [n. d.]. EF08. https://github.com/vadz/libtiff/commit/0a76a8

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022). https://doi.org/10.48550/arXiv.2204.05999 arXiv:2204.05999

[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] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *(TOSEM)* 30, 2 (2021), 1–27.

[15] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. 2013. A systematic mapping study of web application testing. *Information & Software Technology* 55, 8 (2013), 1374–1396.

[16] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Neural Information Processing Systems*.

[17] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403* (2022).

[18] Kai Huang, Su Yang, Hongyu Sun, Chengyi Sun, Xuejun Li, and Yuqing Zhang. 2022. Repairing Security Vulnerabilities Using Pre-trained Programming Language Models. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 111–116.

[19] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[20] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[21] Yi Li, Shaohua Wang, and Tien Nhut Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 602–614.

[22] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020).

[23] Siqi Ma, Ferdian Thung, D. Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *European Symposium on Research in Computer Security*.

[24] NIST. 2023. CVE vulnerability CVE-2017-7601. https://nvd.nist.gov/vuln/detail/CVE-2017-7601

[25] OpenAI. 2023. ChatGPT Plugins. https://openai.com/blog/chatgpt-plugins

[26] OpenAI. 2023. OpenAI Codex. https://openai.com/blog/openai-codex.

[27] OWASP. 2023. Vulnerability Disclosure Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html

[28] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1–18.

[29] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A comparative study of automatic program repair techniques for security vulnerabilities. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 196–207.

[30] Jessica Shieh. 2023. Best practices for prompt engineering with OpenAI API. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api.

[31] LLVM team. 2023. CLang 17 git documentation: Address Sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[32] LLVM team. 2023. CLang 17 git documentation: Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[33] OpenAI team. 2022. ChatGPT API. https://platform.openai.com/docs/api-reference/introduction.

[34] OpenAI team. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt.

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[36] 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. *arXiv preprint arXiv:2305.18607* (2023).

[37] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of ICSE 2023. Association for Computing Machinery*.

[38] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[39] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[40] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. arXiv:2305.04369 [cs.FL]

[41] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.

Received 2024-04-05; accepted 2024-05-04