PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise

Zheng Yu[◇◁] Ziyi Guo[◇◁] Yuhang Wu[◇◁] Jiahao Yu[◇] Meng Xu[○◁] Dongliang Mu[•] Yan Chen[◇◁] Xinyu Xing[◇◁]
[◇] *Northwestern University* [○] *University of Waterloo*
[•] *Huazhong University of Science and Technology* [◁] *42-b3yond-6ug Team* [*]

Abstract

Automated program repair (APR) techniques, which aim to triage and fix software bugs autonomously, have emerged as powerful tools against vulnerable code. Recent advancements in large language models (LLMs) have further shown promising results when applied to APR, especially on patch generation. However, without effective fault localization and patch validation, APR tools specialized in patching alone cannot handle a more practical and end-to-end setting—given a concrete input that triggers a vulnerability, how to patch the program without breaking existing tests?

In this paper, we introduce PATCHAGENT, a novel LLM-based APR tool that seamlessly integrates fault localization, patch generation, and validation within a single autonomous agent. PATCHAGENT employs a language server, a patch verifier, and interaction optimization techniques to mimic human-like reasoning during vulnerability repair. Evaluated on a dataset of 178 real-world vulnerabilities, PATCHAGENT successfully repairs over 90% of the cases, outperforming state-of-the-art APR tools where applicable. Our ablation study further offer insights into the how various interaction optimizations contribute to PATCHAGENT's effectiveness.

1 Introduction

As programs grow in size and complexity, they also become increasingly vulnerable, as evident in reports [4] and CVE records [57]. For example, fuzz testing (a.k.a. fuzzing) alone has enabled the discovery of thousands of vulnerabilities in large and complex software [85, 91], not to mention other program analysis tools [17, 92] that continuously run on the software deployment pipelines. However, merely discovering vulnerabilities is not sufficient to eliminate the threat; these vulnerabilities must be mitigated promptly and effectively.

While generic program hardening techniques such as memory defense [51, 98, 109, 112] or software fault isolation (SFI) [55, 76, 79, 82] have been proposed to mitigate certain types of vulnerabilities, these techniques can run into performance or compatibility issues [89]. More importantly, they do not aim to fix the code logic that causes the vulnerability. In contrast, Automated Program Repair (APR) [46]—especially source code-based APR tools—aim to patch the buggy code directly without distorting functionalities nor introducing unnecessary overhead. Effective APR tools can significantly reduce if not eliminate manual effort in patching a security vulnerability [40], and hence, may help shorten the time frame between vulnerability discovery and fix rollout.

Over the past decade, APR has received much attention from researchers [46, 58, 114], especially on patch generation techniques. Briefly, a patch generator takes both the buggy code snippet and some form of bug description (a.k.a., bug metadata) as input and produces a patch that fixes this bug without violating generic requirements for patches (e.g., edit distance [13], idiomaticity [87], or functional specifications [37]). More recently, the research on patch generator has entered the era of large language models (LLMs) [23, 102, 103], especially when LLM-based patch generators have outperformed conventional ones in results [102].

However, patch generation is only a midstream task in APR. Most patch generators require effective fault localization (FL)—some even assume perfect FL [96, 103, 104] —to pinpoint the buggy code snippet. This introduces two challenges when applying end-to-end APR to real-world software: 1) FL techniques based on static analysis are prone to high false positive rates [52], and patching correct code is not only dangerous but also creates extra work for developers. 2) FL techniques based on dynamic execution of proof-of-concept (PoC) test cases face the challenge of slicing a real-world program into a small bug-enclosing snippet. Previous studies have shown that the execute traces of bug-triggering inputs are typically excessively long [11, 77], ruling out straightforward adoption of program slicing techniques [39, 59] .

The downstream task of patch generation is patch validation, which is often left to either manual review [100] or automated testing [78] where in the latter case a patch is

[*]The work is conducted while 42-b3yond-6ug members are under the support of DARPA AIxCC prize award.

considered "correct" when all the tests pass, including the mitigation of the PoC, if exists. While this is arguably the state-of-the-practice [28, 44, 100] treating patch generation and validation as separate steps forgoes the opportunity to harvest useful information in partially correct patch and the reasons of failure, which could be used as a feedback

for the next round of patch generation.

In this paper, we take a holistic view of APR and propose PATCHAGENT, a **push-button APR** tool that handles FL, patch generation, and patch validation in an integrated LLM agent which manages the entire context during APR. In particular, PATCHAGENT tackles a practical issue—patching a vulnerable program based on a single PoC test case (i.e., a guaranteed true positive bug report). This is modeled after realistic settings such as 1) a fuzzer finds a bug or 2) a community member files a bug report with a PoC test case included. More specifically, PATCHAGENT targets large and complex software with source code available and requires that:

- at least one PoC test case to trigger the vulnerability
- a (textual) description of the vulnerability triggered
- a functional test suite to validate integrity of core logic

Static analysis reports, on the other hand, are not required by PATCHAGENT although they can be integrated as additional metadata on the vulnerability triggered.

The key principle behind PATCHAGENT is to mimic how human developers might triage and patch a bug, which typically includes a mixed ordering of actions ranging from ① comprehending bug reports, ② comprehending code snippets, ③ resolving definitions of symbols, ④ writing a patch, and ⑤ applying the patch for validation. As most pre-trained LLMs only support ①, ②, and ④ natively, we additionally program a language server (for ③), and a patch verifier (for ⑤) as abilities into the LLM agent. Note that we do not claim generality nor optimality on the set of abilities provided in PATCHAGENT as they are based on self-reflection of how members in the author team patch bugs and we look forward to seeing a more principled approach in devising the set of abilities.

However, merely providing the abilities to the LLM agent does not empower the agent to "reason" like a developer (shown in §3), which could be caused by the contrast that bug triaging and patching usually involves heavy code analysis [31, 105] while LLMs do not have robust reasoning capabilities [32]. To address this issue, we introduce an assisted reasoning middleware between the LLM agent and the APIs for the provided abilities. The middleware contains four mechanisms: ❶ report purification to facilitate an LLM in interpreting bug reports; ❷ chain compression to shorten the reasoning chain of the LLM agent; ❸ auto correction to correct errors that occur during the interaction between LLM and ability APIs; and ❹ counterexample feedback to encourage the LLM agent to generate diversified patches. These optimizations bring remarkable improvements as shown in §7.3.

Simultaneously, we understand that even with the introduction of these four distinct optimization components in our PATCHAGENT framework, it does not imply that our system has achieved human-comparable capabilities in holistic program repair. Human experts remain unparalleled in their ability to address real-world vulnerabilities. Through PATCHAGENT, we aim to leverage insights inspired by human expertise to assist LLMs in improving program repair tasks. Looking ahead, we aspire to gradually uncover and integrate more nuanced patching techniques, practices, and tips from human experts into PATCHAGENT, further enhancing its effectiveness over time.

To demonstrate the effectiveness of PATCHAGENT in repairing real-world vulnerabilities, we created a dataset comprising 178 cases sourced from OSS-Fuzz [85], Huntr [34] and ExtractFix [24] on 9 distinct bug types: stack overflow, heap overflow, integer overflow, use-after-free, double free, global overflow, divide by zero, invalid free, and null dereference. PATCHAGENT is built upon the GPT-4 series from OpenAI [72] and the Claude-3 series from Anthropic [10]. PATCHAGENT exhibited remarkable performance on the dataset, successfully repairing 92.13% vulnerabilities. Each repairing solution passed both the security tests and functional tests. We also show that PATCHAGENT outperforms two state-of-the-art APR methods (ExtractFix [24] and Pearce et al. [78]) that are closely aligned with PATCHAGENT in the overall goal.

Contributions. In summary, the four main contribution of our works are as follows:

- We propose a novel LLM-based program repair agent that leverages a language server and patch verifier to analyze programs, generate patches, and validate them.
- We introduce four interaction optimizations to enhance the repair performance of PATCHAGENT. An ablation study demonstrates their effectiveness in improving repair performance.
- We evaluate our prototype and provide in-depth analysis, demonstrating the effective and efficient of PATCHAGENT, including on vulnerabilities that LLMs have never encountered before.
- **PATCHAGENT has made an impact in the real world.** We successfully used PATCHAGENT to repair numerous real-world vulnerabilities. Additionally, we employed PATCHAGENT to participate in the DARPA AI Cyber Challenge (AIxCC) [8], where we advanced to the finals as team 42-b3yond-6ug [1].

We have strictly followed ethical guidelines when developing PATCHAGENT. We also plan to make our code, dataset, and evaluation artifacts publicly available to promote transparency and follow-up works.

2 Background on Automated Program Repair

Automated Program Repair (APR) aims to reduce the manual effort required to fix vulnerabilities. In this work, we focus on

scenarios where a proof-of-concept (PoC) input is available, accompanied by a vulnerability description and a functional test suite to ensure the integrity of core logic, thus eliminating the need for static analysis. It is important to note that not all APR approaches adhere to this setting; many rely on static analysis [31, 105] or exact fault localization [96, 103, 104]. Our PoC-driven approach streamlines integration with fuzzing which provides PoC inputs, and boosts practicality especially considering the sheer volume of bugs found in industry-scale fuzzing campaigns like OSS-Fuzz [85] and syzkaller [91].

2.1 Workflow for PoC-driven APR

Under this setting, the APR process typically involves three key steps, as described below:

Fault localization. Fault localization (FL) aims to identify the root cause of a vulnerability and to provide an optimal code location to apply patches. Previous works [16, 30, 108] have utilized program analysis techniques such as data flow analysis and symbolic execution to verify program entities against manually crafted rules to uncover potential root causes. However, these analysis rules are typically bound to certain vulnerability types and are limited by high computational overhead. Other works [11, 77, 86, 106] employ a statistics-based method, which involves scoring elements in the program to perform root cause analysis. To enhance the statistics-based method, fuzzing techniques are often employed to explore both crashing and non-crashing inputs. These methods are also time-consuming due to the extensive use of fuzzing. Additionally, they fail to provide the exact root cause location and cause, instead offering a list of potential candidates.

Patch generation. Broadly categorized, a patch generator can be ① search-based [45, 80], which search for a correct patch in a predefined patch space scoped by heuristics; ② constraint-based [24, 31], which employ advanced constraint solvers or program synthesis techniques to generate candidate patches that toggle the bug-triggering condition; ③ pattern-based [54, 97], which applies program fixed templates (a.k.a., transformation schema) to buggy code to generate patches, where the templates can be either manually defined or mined automatically; ④ learning-based [36, 111], which learns a mapping between a buggy code snippet (with optional metadata) and the corresponding patch via training and applies the learned model to generate patches. It is different from pattern-based APRs primarily because fix templates are never *explicitly* defined in the process.

Patch validation. Fixation [28] uses distance-bounded weakest preconditions to identify partially fixed exceptions in Java programs. Le and Pattison [44] introduced a novel program representation called the multi-version interprocedural control flow graph, which integrates and compares the control flow of multiple versions of programs. They also developed a demand-driven, path-sensitive symbolic analysis that traverses the graph to detect bugs related to software changes.

```
const M3OpInfo c_operations[] = { /* ... */ };
const M3OpInfo c_operationsFC[] = { /* ... */ };

static inline const M3OpInfo*
GetOpInfo(m3opcode_t opcode) {
    switch (opcode >> 8) {
    case 0x00:
        return &c_operations[opcode];
    case 0xFC:
        return &c_operationsFC[opcode & 0xFF];
    default:
        return NULL;
    }
}

M3Result
Compile_BlockStat(IM3Compilation o) {
    m3opcode_t opcode;
    Read_opcode (&opcode, &o);
    IM3OpInfo opinfo = GetOpInfo(opcode);
    _throwif(unknownOpcode, opinfo == NULL);
    if (opinfo->compiler) { // global overflow
        (*opinfo->compiler) (o, opcode)
    } else {
        Compile_Operator (o, opcode);
    }
}
```

Listing 1: An out-of-bound access vulnerability in WASM3.

KLAUS [100] leverages abstract interpretation to extract modified read and write operations caused by patches in the Linux kernel. It combines these modifications with branch-resolving mechanisms to guide a kernel fuzzer toward relevant code and contexts. However, these methods focus on specific programs and vulnerabilities and are not adaptable to the diversity of real-world programs.

Holistic solution. ExtractFix [24] is a holistic APR solution that employs symbolic execution for fault localization. It extracts a crash-free constraint from a PoC and identifies suitable fix points within the program's code. This constraint is then propagated to these points, where a pattern-based method generates patches that satisfy the constraint for all possible inputs. The PoC is subsequently replayed to verify the correctness of the patch. However, a notable drawback of this approach is the significant computational overhead associated with symbolic execution, which often suffer from path explosion issues. Additionally, the pattern-based patch generation approach may struggle with more complex vulnerabilities, such as use-after-free scenarios.

2.2 LLMs and Their Applications in APR

Large Language Models (LLMs) have demonstrated exceptional capabilities in various natural language processing tasks, including text classification and generation [12, 81]. By leveraging their sophisticated language modeling abilities, LLMs can generate coherent text by predicting subsequent to-

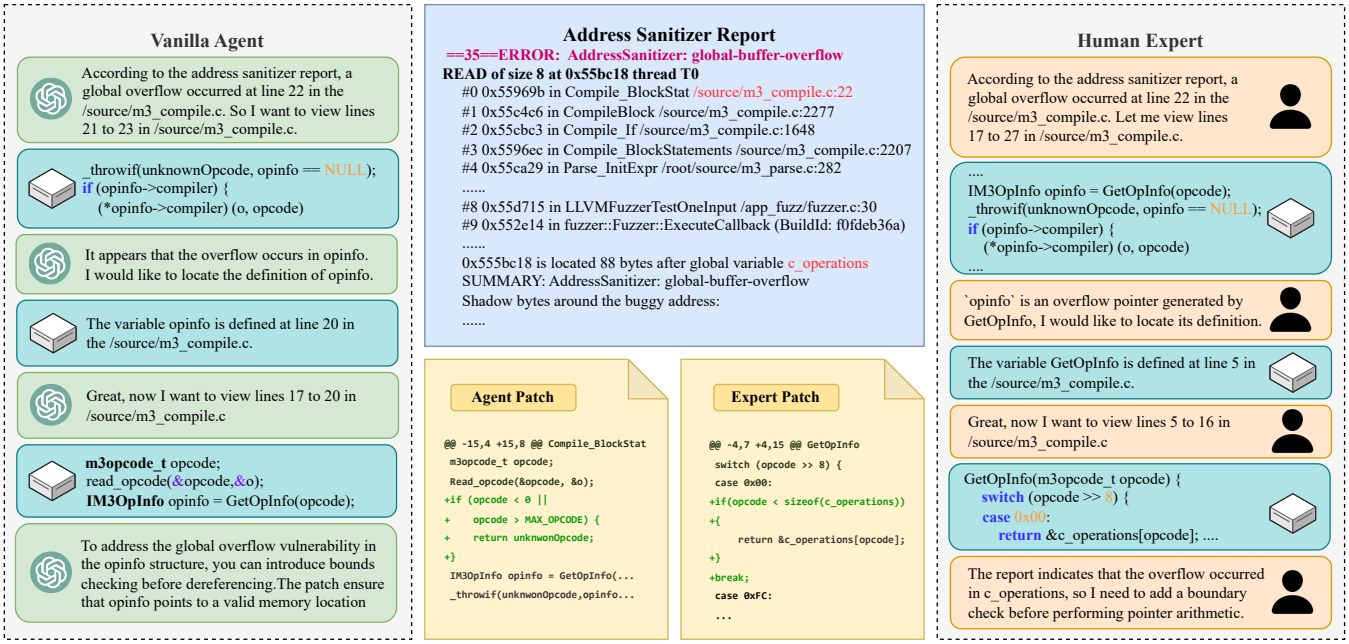

Figure 1: **Comparison between vanilla agent and human expert vulnerability repair processes.** The center part displays the AddressSanitizer report alongside the patches generated by both human expert and vanilla agent. The left side illustrates the repair process of vanilla agent, while the right side shows the repair process of human expert.

kens or words based on a given input. Their potential extends beyond natural languages, showing significant proficiency in code generation as well. Previous research has highlighted the effectiveness of LLMs in aiding human developers by generating functional code and addressing security vulnerabilities within software programs [14, 78, 83].

Without doubt, LLMs fit perfect well with APR and many applications have been proposed. As a subset of ④ learning-based patch generation techniques, LLM-based tools [38, 102] have outperformed other patch generators as reported in recent surveys [33, 114]. On the fault localization (FL) front, SemiAutoFL [60] focuses on LLM-based fault localization by multiple interactions with LLMs. However, it is designed to be semi-automated, and human efforts have to be introduced during the repair process. This huge difference separates it from our fully automated APR system. As for patch validation, researchers explore the possibility of introducing patch validation feedback [41] into LLMs for improving APR.

However, instead of being a holistic APR tool, these LLM-based tools mostly focus on using LLMs to implement one component in the APR process. Pearce et al. [78] is an LLM-based APR tool that integrates both patch generation and validation but lacks FL, as it relies on developer-provided patches as an oracle for localizing the patch point. The corresponding code is then fed to the LLM, and the patch is validated by replaying the PoC and running a functional test suite. And yet, this is the most closely related work in our PoC-driven APR setting. Designing new APR workflow [107] without fault localization (free-FL), under the assistance of LLMs, is another potential way for APR. However, at this moment, we are unable to compare with it because (1) Their workflow is built for Java projects, which differs from our C/C++ targets. (2) Currently, their project is not open-sourced.

3 Motivation: Human vs Vanilla LLM Agent

In this section, through a concrete example, we show that a vanilla LLM agent that merely shadows the abilities of human developers produce only substandard patches. This hints at the importance of (subtle) human expertise in program repair that are yet to be provisioned into the vanilla LLM agent.

3.1 Motivating Example

Issue-33078. Listing 1 presents an out-of-bound (OOB) access bug that causes issue-33078 [35], which was discovered by OSS-Fuzz [85]. The OOB access is flagged by AddressSanitizer [84] at line 22 in function `Compile_BlockStat` when the OOB pointer `opinfo` is dereferenced. `opinfo` is produced in function `GetOpInfo`, which fails to validate the `opcode` properly—the root cause. However, AddressSanitizer cannot flag the root cause because `GetOpInfo` does not dereference `opinfo`. In fact, `GetOpInfo` does not even show up in the call trace of the AddressSanitizer report (see Figure 1).

Repair by human. The process by which a human expert repairs the vulnerability is illustrated in the right side of Fig-

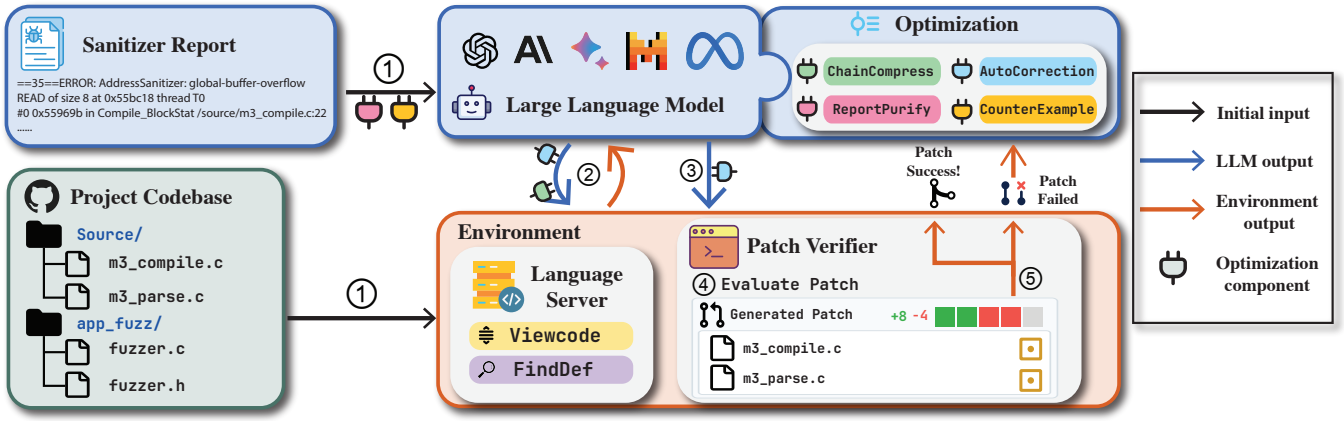

Figure 2: **Overview of PATCHAGENT.** The process begins with the sanitizer report and the project codebase (①). The LLM retrieves the code context using the *viewcode* and *find_definition* APIs (②) and then generates a patch (③). The patch is subsequently validated by the patch verifier (④). If the patch is incorrect, the agent will refine the patch or gather additional context (⑤), iterating until a correct patch is generated or the budget is exhausted. The optimization components enhance the agent's capabilities, bringing it closer to the level of human expertise.

ure 1. ① The expert first identifies that the crash occurs at line 22 in *m3_compile.c* and proceeds to examine the code around that line. ② After reviewing the code, the expert discovers that the overflow is caused by dereferencing `opinfo`, which is the return value of the function call to `GetOpInfo`. ③ The expert then locates the definition of `GetOpInfo` and examines the function body. ④ At this point, the expert notices in the sanitizer report that the overflow occurs near global variable `c_operations`, and more importantly, `GetOpInfo` returns a pointer based on this variable. ⑤ This inspires the expert to add a bounds check before performing the pointer arithmetic as the patch. ⑥ After replaying the PoC and running functional tests, the patch is deemed correct.

In this process, the human expert primarily relies on three abilities that are natively built into LLM: comprehending sanitizer report (in ①,④) and code (in ①,②,④), and generating code (in ⑤), and uses three additional abilities: retrieve code snippet (in ①, ③), locate a symbol's definition (in ③), and validate patch (in ⑥), to effectively complete the repair task.

Repair by a vanilla agent. To fairly compare how an LLM agent patches a bug with the human approach, we developed a vanilla agent equipped with the three additional abilities—retrieving code snippets by range, locating a symbol's definition, and validating a patch—by coupling a language server and patch verifier to the agent (see §4.1 for details).

One sample run of the repair process by the agent is shown on the left side of Figure 1. The agent identifies that the overflow occurs at line 22 in *m3_compile.c* but attempts to view the surrounding code with a very narrow range. Consequently, it does not directly find the definition of `opinfo` as a human expert might. Instead, the agent first locates the definition of `opinfo` and then examines the relevant code. Although the agent successfully retrieves the definition of `opinfo`, it fails to continue locating the definitions of the symbols on which `opinfo` depends. Instead, it arbitrarily assumes that the overflow is caused by the incorrect usage of `opcode` and add a "boundary check" before `GetOpInfo`, leading it to generate an incorrect patch. After the patch validation fails, we reset and rerun the agent, but it continues to generate similar patches.

3.2 Reflection on Both Processes

Comparing the program repair processes by the vanilla LLM agent and the human expert, we identified four challenges that need to be addressed to elevate the vanilla agent's capabilities to a level approaching human expertise.

❶ *Ineffective Ability Utilization*: The vanilla agent struggles with effectively utilizing the abilities at its disposal. In this example, the agent consistently limits its attention to narrow ranges of code snippets, costing it three rounds to fetch both the crash site code and the definition of `opinfo`. In contrast, the human expert adopts a broader perspective and gathers the same information in a single round.

❷ *Timing of Ability Application*: The agent can not use abilities at the appropriate time. In this example, after locating the definition of `opinfo`, although the code indicates that `opinfo` depends on `GetOpInfo`, the agent does not use the ability to find the definition of `GetOpInfo`, leading to an incorrect decision. In contrast, the human expert correctly identifies this dependency and applies the ability at the right time to obtain the necessary information.

❸ *Report Comprehension*: The sanitizer report highlights that `c_operations` is a key variable, but the agent did not mention this detail during the repair process. In contrast, the human expert identifies its importance in the final step. This oversight by the agent leads to an incomplete understanding of the vulnerability.

❹ *Lack of Variability*: The agent lacks sufficient randomness in its approach, leading to repetitive and ineffective solutions. When a new patching attempt is initiated, the previous effort, including the reason for failure, is not taken into account.

While a more formal behavioral analysis might uncover more gaps between human and the vanilla agent, we believe these four issues could be a starting point for improvement. To close the identified gaps, we design a series of interaction optimizations to guide or restrict the behavior of the vanilla agent, as detailed in §4.2. As an overview, we introduce a middleware that includes a report purifier, designed to transform sanitizer reports into a format that can be easily processed by the LLM. Additionally, the middleware monitors and adjusts the LLM's use of its abilities. It can also collect the failed patches generated by the LLM and provide feedback to prevent repeated generation of similar patches to some extent.

4 The PATCHAGENT Design

In this section, we present the design of PATCHAGENT. We start by outlining the basic framework (§4.1), followed by an overview of its optimization components (§4.2). Finally, we describe the prompt design (§4.3).

4.1 Framework

The overall framework of PATCHAGENT is illustrated in Figure 2. PATCHAGENT takes the sanitizer report and the project codebase (①) as the input. The sanitizer report includes details such as the bug type, stack trace, and other relevant information, while the project codebase contains the project's source code. PATCHAGENT builds a language server for the codebase to facilitate code retrieval and analysis and sets up the runtime environment necessary for the patch verifier. The language server is built on the Language Server Protocol (LSP), a universal standard that supports over 50 languages. The versatility of LSP is a key factor motivating PATCHAGENT to adopt it.

The patch verifier ensures the correctness of the generated patch. In this work, we consider a patch correct if it resolves bugs without disrupting functionality, as evaluated by test suites rather than program equivalence. The verifier applies the patch, checks for syntax errors, and determines whether the vulnerability can still be triggered. It also runs all functional tests to ensure functional integrity. For temporal bugs, we observed that LLMs may address them by simply removing the free operation, which prevents the sanitizer from detecting the issue. However, this approach is generally unacceptable in real-world scenarios. To avoid such problems, we employed LeakSanitizer [5] to check for any additional memory leaks introduced post-patch. If such leaks are detected, the patch is deemed invalid.

In each patch iteration, PATCHAGENT first analyzes the sanitizer report and interacts with the project codebase (②) to retrieve the relevant code context. Two retrieval APIs are available for the LLM agent to invoke: *viewcode* and *find_definition*. The *viewcode* API retrieves the code context by specifying file names and line numbers, while the *find_definition* API finds the definition location of symbols by specifying their names and reference locations. Once sufficient code context is retrieved, the agent generates a patch (③) and invokes the *validate* API to check the patch's correctness (④). If the patch is incorrect, it will provide feedback to the agent (⑤), and the agent will refine the patch (③) or retrieve additional code context (②) that is necessary for the patch generation. This process repeats until a correct patch is produced or the budget for each round is exhausted. PATCHAGENT conducts multiple patch rounds with a positive temperature. The agent is reset after each round, and the process continues until the total budget is exhausted or a correct patch is found.

4.2 Incorporating Human Expertise

While the framework discussed so far outlines the basic workflow of PATCHAGENT, it does not fully address the challenges mentioned in §3. To bridge this gap and incorporate human expertise, we introduce several optimization components into the agent, represented as the plugin object in Figure 2. These components are designed to emulate the problem-solving strategies of human expert, addressing the four challenges identified earlier. Optimization comprises four key components, each targeting a specific challenge.

1. **Report Purification:** This component tackles the challenge of *Report Comprehension* (❸). It transforms complex sanitizer reports into a format that the LLM can easily process, emulating how an experienced developer would focus on and interpret key details from error reports.
2. **Chain Compression:** Addressing the issue of *Timing of Ability Application* (❷), this component helps the agent make more optimal decisions about when and how to use available abilities. It assists in identifying dependencies and providing the necessary code context autonomously for the LLM agent, much like how a human expert would navigate through code relationships.
3. **Auto Correction:** This component addresses the challenge of *Ineffective Ability Utilization* (❶). It automatically corrects ineffective or invalid ability usage, mimicking an expert's proficiency in utilizing these abilities correctly. By doing so, it prevents the LLM from repeatedly adjusting parameters when calling ability APIs. This ensures that the LLM effectively uses these abilities and retrieves the necessary information.
4. **Counterexample Feedback:** Even with restarting the patching process, the agent may still generate similar ineffective patches repeatedly without self-reflection. Ad-

dressing the *Lack of Variability* (❹) in the agent's approach, this component makes the agent learn from past attempts. It saves failed patches and provides feedback to prevent the generation of similar ineffective patches repeatedly, mimicking a developer's ability to learn from mistakes and vary their approach.

These components work in concert to enhance the agent's performance by incorporating human-like problem-solving strategies. They form a series of interaction optimizations that guide and restrict the behavior of the native agent, elevating its capabilities to a level approaching human expertise. In §5, we will delve deeper into each component, explaining how they contribute to more effective program repair.

4.3 Prompt Design

The initial prompt includes both a system prompt and a user prompt. The system prompt, which remains constant across different repair tasks, provides a detailed overview of repair tasks, explains how LLMs can interact with the environment, and offers strategic suggestions for leveraging ability APIs. The user prompt, tailored from specific vulnerability information, is divided into three sections. The first section presents the purified content of the sanitized report, ensuring that all sensitive details are clarified. The second section provides in-depth explanations related to the sanitized report, offering additional context and insights. The final section includes precise instructions for the LLMs to follow in repairing the identified vulnerability.

5 Interaction Optimization

In this section, we introduce the optimization of the interaction with LLMs, which plays an important role in improving the repair performance of PATCHAGENT.

5.1 Report Purification

As we mentioned in §3, we identified issues when LLMs process initial prompts that include sanitizer reports. These reports often contain noisy symbols and complex information, which can obscure key details and reduce the effectiveness of the LLM. The report purification mechanism is designed to streamline and clarify the information from the original report, making it more suitable for LLM processing. Specifically, we implemented a parser to transform the original report into a concise and clear format. The parser first analyzes the report to identify the attributes of each symbols. Next, it removes unnecessary symbols, such as memory addresses, shadow memory bytes, and symbols intended solely for human readability. The parser then recalculates numerical data within the report, such as access offsets and object sizes in out-of-bounds bugs, to ensure accuracy and integrity. Additionally, it appends clear and concise explanations for complex data fields or technical terms, such as vulnerability types, stack traces, and other critical details. Finally, the parser appends the repair suggestions to the end of the report, thereby reducing ambiguities and enhancing the clarity of sanitizer reports.

5.2 Chain Compression

Integrating an LLM with a language server enables the LLM to analyze vulnerabilities in a manner similar to human developers. We expect PATCHAGENT to efficiently navigate code, locate symbol definitions, and conduct thorough vulnerability analyses. Once the necessary information is gathered, the LLM is expected to generate correct patches. However, we have observed that LLMs may sometimes halt the code retrieval process and stop interacting with the environment, resulting in incorrect patches due to incomplete information. This issue is particularly problematic for vulnerabilities involving numerous code segments and variables, as the LLM must gather complete information through multiple interactions. The need for the LLM to gather comprehensive information over several interactions introduces a long interaction chain, increasing the complexity of decision-making. This complexity can significantly reduce repair performance, especially in complex repair tasks, highlighting the importance of optimizing the interaction process.

To address this problem, we employ chain compression to optimize the interaction. We regard the interaction process as a chain, with each round of interaction representing a node on this chain. Each time, the LLM needs to infer the next action based on the current information. By studying how humans analyze vulnerabilities and observing the actions of LLMs, we found that some inference steps are trivial and can be handled by non-LLM algorithms. Chain compression optimization will automatically detect if the current inference step is trivial. If it is, the system bypasses the LLM, directly generates and executes the next action, and returns all the obtained information to the LLM. This approach compresses multiple nodes on the chain of interaction into a single node. From the perspectives of both the LLM and the environment, the number of interaction iterations is reduced.

PATCHAGENT employs two types of predefined rules to identify trivial inference steps: **Dominator Action** for deterministic scenarios and **Heuristic Exploration** for non-deterministic ones. Here are the details of these two mechanisms:

- **Dominator Action**: When LLMs need to retrieve complete information, they may require multiple related actions. For example, after locating the definition of a variable, the LLM may need to examine the associated code or recursively identify other symbols that the variable depends on to ensure the definition is comprehensive. The initial action in this sequence is called the dominator action, while all subsequent actions necessary to gather complete information must be executed; otherwise, the

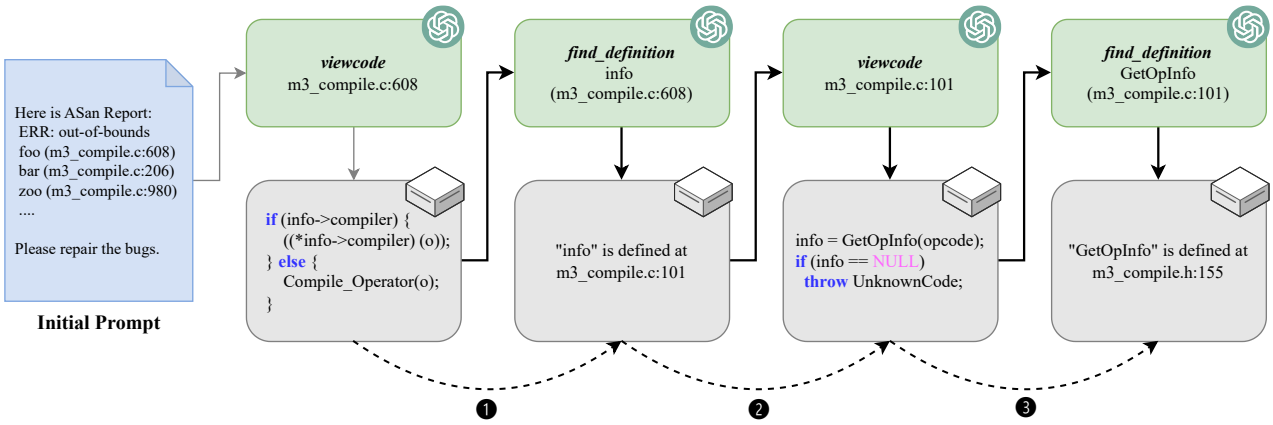

Figure 3: **Example of Chain Compression.** The LLM takes the initial prompt as input and starts interacting with the language server. The black bold arrows illustrate the interaction without chain compression, while the black dashed arrows represent the compressed interaction process. The original interaction chain of length four was compressed into a single interaction.

LLM can only obtain incomplete information. When chain compression identifies an action as a dominator action, it automatically generates and executes the subsequent required actions, ensuring the LLM has access to complete data.

- **Heuristic Exploration**: LLMs typically need to explore various symbols (e.g., functions and variables) in the codebase, requiring many interaction iterations. This exploration helps LLMs gather contextual information about vulnerabilities. To optimize this process, we designed a heuristic exploration strategy to select these symbols. We observed that symbols near the lines mentioned in the sanitizer report are chosen more frequently. Therefore, after LLMs review a code snippet, our system directly samples symbols near the lines mentioned in the sanitizer report. The system then finds their definition locations and returns this information to the LLM.

To better understand chain compression, we use Figure 3 to illustrate how the optimization works. The LLM takes the initial prompt as input and starts interacting with the language server. The actions taken by the LLM are shown in the green box, while the responses from the language server are displayed in the gray box. Without chain compression, the LLM requires four iterations to gather the information displayed in the gray box, as indicated by the bold black arrows. With the optimization applied, only one iteration is needed, as shown by the dashed black arrows. The chain compression mechanism is activated three times during this process.

❶ represents the **Heuristic Exploration** mechanism. After the LLM sends a *viewcode* action, the mechanism determines that the crash is caused by the dereference of `info` and the line where `info` located appears in both the viewed code snippet and the sanitizer report. This indicates that it is a valuable symbol to explore. Consequently, the chain compression mechanism automatically triggers an additional *find_definition* action to locate the definition of `info`. ❷ and ❸ represent the **Dominator Action** mechanism. Using only the *find_definition* action to locate the definition of `info` is insufficient to reveal its complete information. Therefore, the mechanism first generates another *viewcode* action to obtain the definition code snippet of `info`. Then, it identifies that the variable relies on another symbol, `GetOpInfo`, and recursively finds its definition location. To prevent infinite action generation, we set a maximum count for the chain compression optimization.

5.3 Auto Correction

The LLMs need to frequently generate numbers to interact with the environment, which usually involves calculations. However, LLMs often struggle with numerical tasks and often generate incorrect numbers. This can cause the environment to deem the actions generated by LLMs as invalid, requiring the LLMs to consume many iterations to fix the actions. While prompt engineering can provide some improvement, we found that this solution is not stable enough. Alternatively, we design a correction mechanism for each action:

***viewcode*.** The *viewcode* action requires the LLM to specify both the filename and the range of code lines to be viewed. We observed that LLMs usually provide a narrow range, leading the generated content to rely heavily on local information while overlooking the broader context. To address this issue, our system automatically expands the range whenever it detects that the specified range is below a predetermined threshold. For example, if the threshold is n and the range provided by the LLM is $[l, r)$ and $r - l < n$, the range will be expanded to $[l - \frac{n-(r-l)}{2}, r + \frac{n-(r-l)}{2})$.

***find_definition*.** The *find_definition* action requires the LLM to provide the row and column numbers corresponding to the reference symbol. If LLMs fail to supply the exact num-

bers, the language server may consider it an invalid action. Our observations indicate that LLMs tend to focus on symbols appearing in the most recently viewed code and those closely related to the vulnerability. Consequently, our correction algorithm operates based on this principle. It sequentially examines the result code snippets from *viewcode* actions, starting with the most recent and moving in reverse chronological order. This backward traversal ensures that the most recently viewed code snippets are analyzed first. Once the algorithm detects the symbol in the currently examined code snippet, it selects the reference locations closest to those provided by the action. This approach is effective because, although an entire codebase may contain many symbols with the same name but different definitions, it is uncommon for a single code snippet to have multiple definitions of the same symbol. Therefore, the algorithm can almost always correctly identify the symbol that the LLM intends to locate.

***validate*.** The *validate* action requires LLMs to provide patches in a multi-hunk format [33], which includes the line numbers of the modified lines. Incorrect line numbers prevent the patch from being applied to the source code. To address this, we observed that the redundant information within the multi-hunk patch offers the potential for error correction. Specifically, each hunk in the patch presents the modified code lines with unchanged contextual lines to help users understand their placement within the file. Consequently, we designed an algorithm based on the minimal edit distance [88] to correct errors in line numbers and contextual lines. The algorithm fixes each hunk in three steps: ❶ Scans all unchanged and deleted lines, merging them into the original code snippet. ❷ Calculates the edit distance between the original code snippet and possible code ranges in the changed file, selecting the range with the minimal edit distance. If multiple ranges have the same edit distance, the algorithm chooses the range closest to the one originally indicated by the patch. ❸ Corrects the contextual lines based on the selected code range and updates the line numbers in the patches. This approach allows for efficient error correction in patches without requiring complete regeneration by LLMs.

5.4 Counterexample Feedback

The workflow of PATCHAGENT runs multiple patch rounds until the system found the correct patch, relying on the LLM's ability to generate diverse patches each round. However, even with the *temperature* set to 1 (indicating a higher probability of varied output), the LLM might still produce patches with similar logic across different rounds. This issue demonstrates the lack of ability of LLMs to self-reflect and improve their performance within multiple rounds. We demonstrate this issue in Listing 2, where the original code bug arises from mishandling a null pointer. Specifically, the program attempts to pass `t->text`, which may be null, to the `gf_strdup` function, which cannot handle null inputs, resulting in a null pointer

```
  if (t) {
    time[i] = (u32) s->DTS;
-   name[i] = gf_strdup(t->text);
+   if (t->text)
+     name[i] = gf_strdup(t->text);
+   else
+     name[i] = NULL;
    gf_isom_delete_text_sample(t);
  }

  if (t) {
    time[i] = (u32) s->DTS;
-   name[i] = gf_strdup(t->text);
+   name[i] = t->text ? gf_strdup(t->text): NULL;
    gf_isom_delete_text_sample(t);
  }

  if (t) {
    time[i] = (u32) s->DTS;
-   name[i] = gf_strdup(t->text);
+   name[i] = gf_strdup(t->text ? t->text : "");
    gf_isom_delete_text_sample(t);
  }
```

```
memset(&tx, 0, sizeof(tx));
tx.text = p2->value.name[i];
tx.len = (u32) strlen(p2->value.name[i])+1;
samp = gf_isom_text_to_sample(&tx);
```

Listing 2: **Example of Counterexample Feedback.**

dereference when accessing the string. The first two patches address the bug by adding checks to ensure `t->text` is not null before calling `gf_strdup`. The first patch uses an if-else structure: if `t->text` is not null, it duplicates the text; otherwise, it sets `name[i]` to `NULL`. The second patch uses a ternary operator to achieve the same logic. Both patches set `name[i]` to `NULL`, which poses a potential issue since `name[i]` is required to remain non-null in other parts of the program. This is evident in the second code snippet in Listing 2, where `name[i]` is passed directly to the `strlen` function without any null checks. Similar patterns are observed throughout the codebase, suggesting that numerous null checks would need to be added to ensure the program's correctness if we were to follow the logic of the first two patches. This could make the patch overly complex and introduce redundancy.

In our experiments, we consistently observed that LLMs tend to generate similar patches, like the first two incorrect examples, when the workflow is run repeatedly with little variation between rounds. This pattern likely arises because adding a null check is a common real-world fix for null pointer issues, leading the LLM to heavily favor this approach. To guide the LLM to generate the correct patch, as shown by the third patch in Listing 2, the solution assigns an empty string to `name[i]` instead of `NULL` when `t->text` is null. We introduce a counterexample feedback mechanism, where patches

Model	Temporal Error	Spatial Error	Null Dereference	Numeric Error	Total
GPT-4o	13/23 (56.52%)	96/125 (76.80%)	23/23 (100.00%)	7/7 (100.00%)	139/178 (78.09%)
GPT-4 Turbo	11/23 (47.83%)	87/125 (69.60%)	21/23 (91.30%)	7/7 (100.00%)	126/178 (70.79%)
Claude-3 Opus	14/23 (60.87%)	108/125 (86.40%)	22/23 (95.65%)	7/7 (100.00%)	151/178 (84.83%)
Claude-3 Sonnet	8/23 (34.78%)	77/125 (61.60%)	17/23 (73.91%)	6/7 (85.71%)	108/178 (60.67%)
Claude-3 Haiku	9/23 (39.13%)	93/125 (74.40%)	19/23 (82.61%)	7/7 (100.00%)	128/178 (71.91%)
Union	20/23 (86.96%)	114/125 (91.20%)	23/23 (100.00%)	7/7 (100.00%)	164/178 (92.13%)

Table 1: **Effectiveness Comparison of Vulnerability Repair Across Various Models.** This table compares the effectiveness of PATCHAGENT when utilizing different LLMs to repair vulnerabilities. The results are classified into four main types of errors: **Temporal Errors** (including stack overflow, global overflow, and heap overflow), **Spatial Errors** (including use-after-free, double free, and invalid free), **Null Dereference**, and **Numeric Errors** (including integer overflow and division by zero). The **Union** row represents the combined results of PATCHAGENT across all models, demonstrating the overall improvement in repair accuracy achieved through the collaborative use of multiple models.

that fail validation are treated as counterexamples. This mechanism samples counterexamples after the first workflow iteration and includes them in subsequent prompts, instructing the LLM not to generate similar patches again. This method ensures the LLM is aware of its previous shortcomings, preventing it from repeatedly generating similar, ineffective patches. By integrating these counterexamples into the prompt, we encourage the LLM to explore a broader range of solutions, increasing the likelihood of producing a correct patch.

6 Implementation

We implemented PATCHAGENT using LangChain [43] to facilitate the integration of LLM and prompt engineering. Below, we provide an overview of the environment support.

Language Server. The language server front-end is based on the Language Server Protocol (LSP) [71], while the back-end utilizes clangd. We employ a customized compiler wrapper to collect the compilation commands for each project, which are then used to initialize the clangd server. Upon receiving an action command from the LLMs, we generate an LSP packet containing all pertinent details and transmit it to the clangd server. The clangd server processes the request, allowing the LLMs to perform a thorough analysis of the codebase.

Patch Verifier. The patch verifier begins by applying the proposed patch to the program and compiling the modified code to ensure that no errors are introduced during the patching process. Upon successful compilation, the verifier replays the PoC to confirm that the identified security vulnerability has been effectively addressed. If the patch passes the security tests, the verifier then runs a series of functional tests within the program to ensure that the patch does not inadvertently disrupt any other functionality.

7 Evaluation

In this section, we assess PATCHAGENT with the following research questions.

- **RQ 1:** How effectively can PATCHAGENT repair vulnerabilities in real-world programs? (§7.2)
- **RQ 2:** What is the impact of individual interaction optimization mechanisms on repair performance? (§7.3)
- **RQ 3:** How does PATCHAGENT perform when repairing vulnerabilities that LLM has never seen before? (§7.4)
- **RQ 4:** How efficient is PATCHAGENT in repairing vulnerabilities? (§7.5)

Additionally, we present and analyze several case studies in §A.1 to offer a comprehensive understanding of the effectiveness and limitations of PATCHAGENT.

7.1 Setup

Hardware Environment. All experiments were conducted on an AMD EPYC 7763 64-core processor running at 2.445 GHz with 512 GB of RAM and 15 TB of SSD storage.

Large Language Model. The large language models used in the experiment include GPT-4 Turbo and GPT-4o from OpenAI, as well as Claude-3 Opus, Claude-3 Sonnet, and Claude-3 Haiku from Anthropic. The specific versions were *gpt-4-0125-preview*, *gpt-4o-2024-05-13*, *claude-3-opus-20240229*, *claude-3-sonnet-20240229*, and *claude-3-haiku-20240229*, respectively.

Dataset. We select 178 vulnerabilities from 30 programs, covering 9 distinct bug types: stack overflow, heap overflow, integer overflow, use-after-free, double free, global overflow, divide by zero, invalid free, and null dereference. Of these, 28 vulnerabilities are sourced from ExtractFix [24]. We excluded two ffmpeg cases from the original 30 ExtractFix examples due to reproducibility issues stemming from outdated code. The remaining 150 cases were collected from OSS-Fuzz [85] and Huntr [34]. We manually collected both security and

functional test scripts for all cases, ensuring that our patch verifier can validate that the generated patches meet both security and functional requirements.

Notably, no existing datasets provide verifiers for both security and functional tests. For example, only a subset of test cases in Magma [29] and FixReverter [116] are reproducible, while CGC [18] can reproduce all cases but does not include functional test scripts. To emphasize the effectiveness of PATCHAGENT on various complex vulnerabilities, we have detailed all relevant sample information of the dataset in §A.3.

Evaluation Criteria. In our evaluation, we primarily compare our approach with ExtractFix [24] and a zero-shot method based on LLMs proposed by Pearce et al. [78]. While attempting to reuse the ExtractFix code on 150 cases from OSS-Fuzz and Huntr, we encountered compatibility issues due to the outdated nature of the code, which prevented it from functioning with the current version. We have confirmed this limitation with the authors. Consequently, our comparison with previous work is limited to 28 cases from ExtractFix.

We also do not compare PATCHAGENT with zero-shot methods [78] or other related works [96, 103, 104] because they require accurate fault localization results, which are not available in our dataset. Additionally, it is non-trivial to adapt PATCHAGENT to fit the settings of these methods, as PATCHAGENT does not take fault localization results as input; it only utilizes the corresponding report and codebase.

For each case and configuration combination, including experiments in the ablation study, we will ensure that the generated patch passes both security and functional tests. Additionally, we will run our system through 15 iterations to minimize the impact of randomness. Since ExtractFix did not perform functional tests in their original evaluation, we manually ran their patches through our functional scripts to ensure proper handling.

Prog.	CVE/Issue	Bug Type	E.F.	Z.S.	P.A.
binutils	CVE-2017-15025	Divide By Zero	●	/	●
binutils	CVE-2018-10372	Heap Overflow	◐	/	●
coreutils	GNUBug 19784	Heap Overflow	○	/	●
coreutils	GNUBug 25003	Heap Overflow	●	/	●
coreutils	Bugzilla 26545	Heap Overflow	●	/	●
coreutils	Bugzilla 25023	Global Overflow	○	/	●
jasper	CVE-2016-8691	Heap Overflow	●	/	●
jasper	CVE-2016-9387	Integer Overflow	●	/	●
libjpeg	CVE-2018-19664	Heap Overflow	○	●	●
libjpeg	CVE-2017-15232	Null Dereference	●	/	●
libjpeg	CVE-2012-2806	Stack Overflow	○	○	●
libjpeg	CVE-2018-14498	Heap Overflow	◐	/	●
libtiff	CVE-2016-5321	Heap Overflow	●	●	●
libtiff	CVE-2017-7595	Divide By Zero	●	●	●
libtiff	CVE-2017-7601	Integer Overflow	●	●	●
libtiff	CVE-2016-9273	Heap Overflow	○	/	●
libtiff	CVE-2016-10094	Heap Overflow	◐	●	●
libtiff	CVE-2014-8128	Heap Overflow	●	○	●
libtiff	Bugzilla 2611	Divide By Zero	●	/	●
libtiff	CVE-2016-5314	Heap Overflow	○	/	●
libtiff	CVE-2016-3186	Heap Overflow	●	/	●
libtiff	CVE-2016-3623	Divide By Zero	●	●	●
libtiff	Bugzilla 2633	Heap Overflow	◐	/	●
libxml2	CVE-2017-5969	Null Dereference	●	○	●
libxml2	CVE-2012-5134	Heap Overflow	●	●	●
libxml2	CVE-2016-1834	Heap Overflow	●	/	●
libxml2	CVE-2016-1838	Heap Overflow	◐	●	●
libxml2	CVE-2016-1839	Heap Overflow	○	/	●

Table 2: **Comparison of vulnerability repair results between ExtractFix (E.F.), Zero-Shot (Z.S.), and PATCHAGENT (P.A.).** ● indicates that the patch successfully fixed the vulnerability and passed the functional test. ◐ denotes a patch that fixed the bug but failed the functional tests. ○ represents a patch that failed to fix the bug. For cases where results are unavailable, a '/' is used to denote this.

7.2 Effectiveness of PATCHAGENT

Table 1 provides a comprehensive comparison of PATCHAGENT's effectiveness in repairing vulnerabilities across different large language models. The vulnerabilities are classified into four main error types based on their nature and the success rate of repair: **Temporal Error**, **Spatial Error**, **Null Dereference**, and **Numeric Error**. These categories encompass nine specific bug types: use-after-free, double free, and invalid free (under Temporal Error); stack overflow, global overflow, and heap overflow (under Spatial Error); null dereference (under Null Dereference); and integer overflow and division by zero (under Numeric Error). The table details the number and percentage of vulnerabilities successfully repaired by PATCHAGENT for each error type across various models. The **Union** row aggregates the results from all models, showcasing PATCHAGENT's repair performance through the collaborative use of multiple models.

From Table 1, we can find that PATCHAGENT delivers strong performance across all bug types by leveraging diverse models, excelling particularly in numeric errors and null dereference with a perfect 100% success rate. For temporal and spatial errors, the success rates are slightly lower, at 86.96% and 91.20%, respectively. These outcomes are consistent with previous studies [24, 31, 33, 105], which suggest that most null dereference and numeric errors can often be resolved with a simple if-check, whereas temporal and spatial bugs typically require more complex solutions. Across individual models, the repair performance for different bug types closely mirrors the trends observed in the union row, with higher success rates for null dereference and numeric errors compared to temporal and spatial errors. On the full dataset, Claude-3 Opus stands out with the highest repair rate, successfully addressing 84.83% of the total vulnerabilities. GPT-4o also performs impressively, achieving a 78.09% repair rate. GPT-4 Turbo and Claude-3 Haiku demonstrate comparable effectiveness.

Configuration	Spatial	Temporal	Other	Total
Disable RP	63.64%	27.27%	88.89%	64.00%
Disable CC	60.00%	36.36%	88.89%	62.67%
Disable AC	41.82%	9.09%	55.56%	38.67%
Disable CF	65.45%	54.54%	**100.00%**	70.67%
PATCHAGENT	**72.73%**	**63.64%**	**100.00%**	**77.33%**

Table 3: **Ablation Study of PATCHAGENT.** Based on GPT-4o (RP: Report Purification, CC: Chain Compression, AC: Action Correction, CF: Counterexample Feedback), **Other** include both null dereference and numeric errors.

CVE/Issue	Project	Bug Type	Fix Date	P.A.
2024-6064 [70]	gpac	Use After Free	Jun. 13rd	✓
2024-27530 [94]	wasm3	Use After Free	N/A	✓
2024-41965 [68]	vim	Double Free	Aug. 1st	✓
2024-3204 [63]	c-blosc2	Heap Overflow	Apr. 4th	✓
2024-34459 [67]	libxml2	Heap Overflow	May 8th	✓
2024-34249 [65]	wasm3	Heap Overflow	N/A	✓
2024-34252 [66]	wasm3	Global Overflow	N/A	✓
Issue-471 [74]	wasm3	Heap Overflow	N/A	✗
2024-6063 [69]	gpac	Null Dereference	Jun. 12nd	✓
2024-34246 [64]	wasm3	Null Dereference	N/A	✗

Table 4: **Summary of unseen vulnerability repair results. P.A.** indicates the repair result of PATCHAGENT (Based on GPT-4 Turbo). The **Fix Date** represents the date when the patch was applied, with all dates in 2024. Data are current as of August 22nd, 2024.

In contrast, Claude-3 Sonnet lags behind, with a significantly lower repair rate of 60.67%.

Table 2 presents a comparative analysis of vulnerability repair results among three approaches: ExtractFix (**E.F.**) [24], LLM-based Zero-Shot (**Z.S.**) [78], and PATCHAGENT (**P.A.**). The results clearly indicate that PATCHAGENT outperforms the other two methods in almost all the listed vulnerabilities. Specifically, PATCHAGENT successfully fixed the vulnerability and passed the functional test in all cases. In contrast, ExtractFix and Zero-Shot show mixed results, with many instances of ○, indicating failed fixes, or ◐, indicating fixes that failed functional tests. This comparison underscores the superiority of PATCHAGENT in effectively patching vulnerabilities while maintaining functional correctness, making it the most reliable approach among the three.

To better understand the unique contributions of each model, we analyzed the number of vulnerabilities each one repaired that others could not. Our findings indicate that Claude-3 Opus had the highest number of unique repairs, addressing 7 vulnerabilities that no other models could fix. In contrast, Claude-3 Sonnet did not contribute any unique repairs, while GPT-4o, GPT-4 Turbo, and Claude-3 Haiku demonstrated 3, 1, and 1 unique repairs, respectively. During the effectiveness evaluation, PATCHAGENT generated 33,336 incorrect patches. Among these, 45.02% failed due to syntax errors, 49.22% did not pass the security test, and 5.76% failed to pass functional tests, highlighting the importance of functional tests.

7.3 Ablation Study

To evaluate the impact of our key idea, applying multiple interaction optimizations for LLM, we conducted an ablation study. In this study, we systematically deactivated each optimization within PATCHAGENT, focusing on report purification (RP), chain compression (CC), auto correction (AC) and counterexample feedback (CF). We sampled 75 cases, primarily due to the high cost associated with running the full dataset. Running these 75 cases alone costs over $1500, highlighting the financial constraints. This approach is also consistent with previous best practices [110]. Furthermore, we ensure that the distribution of vulnerability types in the sampled cases is consistent with the original dataset. The result is shown in Table 3, we illustrate the impact of each optimization component by examining the repair ratios across three categories: Spatial, Temporal, and Other, which include null dereference and numeric errors. The complete PATCHAGENT system achieved a repair ratio of 77.33%.

When report purification (RP) was disabled, the overall repair ratio dropped significantly to 64.00%, with a marked decline in effectiveness across all bug types, particularly in Temporal errors, where the ratio fell to 27.27%. This highlights the crucial role RP plays in refining reports. Disabling counterexample feedback (CF) also led to a decreased overall repair ratio of 70.67%. The CF component proved essential for handling Temporal errors, reducing the repair ratio from 63.64% to 54.54%. Turning off chain compression (CC) resulted in a drop in the repair ratio to 62.67%, with a notable impact on Spatial errors, where the ratio decreased from 72.73% to 60.00%. Disabling auto correction (AC) caused significant degradation, as without this optimization, we observed that the LLM even struggled to generate correctly formatted patches.

7.4 Repair Unseen Vulnerability

To assess PATCHAGENT's capability in repairing vulnerabilities that LLMs have never encountered before, we collected 10 newly discovered vulnerabilities spanning 5 different bug types across 5 distinct projects: (1) `c-blosc2` [2], a fast binary compressor; (2) `GPAC` [27], a multimedia framework; (3) `libxml2` [9], an XML toolkit library; (4) `wasm3` [3], a WebAssembly interpreter; and (5) `vim` [95], an improved version of the UNIX editor Vi. These cases were selected based on the following criteria: (1) the vulnerabilities were reproducible and disclosed in 2024; and (2) the projects have functional tests. We utilized the GPT-4 Turbo-based system, PATCHA-

Model	Avg. Token # Input	Avg. Token # Output	Avg. $ Cost	% Repaired
GPT-4o	234,533	38,331	$1.75	78.09%
GPT-4 Turbo	89,856	13,005	$1.29	70.79%
Claude-3 Opus	83,051	12,111	$2.15	84.83%
Claude-3 Sonnet	180,944	18,574	$0.82	60.67%
Claude-3 Haiku	254,609	26,476	$0.10	71.79%
Union	842,993	108,498	$6.11	92.13%

Table 5: **Token & Money Cost of PATCHAGENT under Different LLM.**

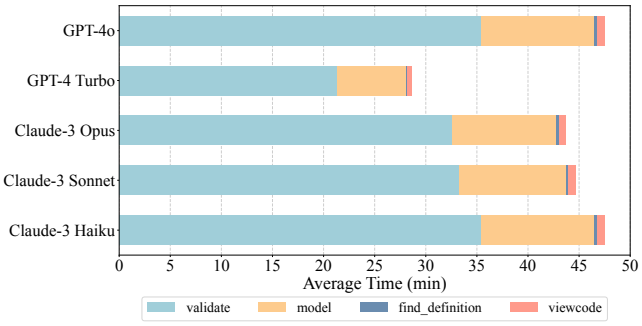

Figure 4: **Average Time Cost of PATCHAGENT.**

GENT, specifically the *gpt-4-0125-preview* version, to repair these vulnerabilities. According to the OpenAI API documentation [73], the training data of this model includes information only up to December 2023. Consequently, the patches for these vulnerabilities are not part of *gpt-4-0125-preview*'s training dataset. This allows us to evaluate PATCHAGENT's ability to repair newly discovered vulnerabilities without concerns about prior knowledge or memorization of the patches.

The results are summarized in Table 4, demonstrating that PATCHAGENT successfully repaired 8 out of 10 previously unseen vulnerabilities. These findings align with the effectiveness evaluation discussed in §7.2. We note that PATCHAGENT failed to repair CVE-2024-34246, a null dereference bug. Given PATCHAGENT's otherwise perfect performance in repairing null dereference bugs during the effectiveness evaluation, we will conduct an in-depth analysis of this failure in §A.1. For another null dereference bug, PATCHAGENT effectively repaired it by inserting a null check. For three temporal memory bugs, PATCHAGENT successfully repaired two by inserting validity checks and mitigated the risks of the third by nullifying the dangling pointer. For spatial memory bugs, the generated patch recalculated the object size, performed advance checks, and added error handling code when necessary. We plan to manually review and measure these patches generated by PATCHAGENT. Once verified to avoid unexpected outcomes, we submit PRs and maintain ongoing communication with developers. The PRs that received responses from maintainers are summarized in A.2.

7.5 Efficiency of PATCHAGENT

Token Cost. Table 5 summarizes the average cost of successfully repairing a vulnerability using different models. Claude-3 Opus achieves the highest repair success rate at 84.83%. However, it also incurs the highest average cost per task at $2.15. Claude-3 Haiku offers the most cost-effective solution at $0.10 per task, though with a lower repair rate of 71.79%. GPT-4o and GPT-4 Turbo present different trade-offs, with GPT-4o using significantly more tokens (234,533 input, 38,331 output) compared to GPT-4 Turbo (89,856 input, 13,005 output), resulting in a higher cost ($1.75 vs $1.29) but also a better repair rate (78.09% vs 70.79%). Claude-3 Sonnet falls in the middle range, repairing 60.67% of vulnerabilities at $0.82 per task. The combined use of all models yields a substantial 92.13% repair rate. In the worst scenario, to achieve such a high repair rate, we would need to run each case through all models, which would cost $6.11. However, we believe that this cost can be reduced through model scheduling. We will consider this as a potential area for future work.

Time Cost. Figure 4 presents a detailed breakdown of the average time costs associated with PATCHAGENT when using different models. The time costs are divided into several subcategories: *validate*, *viewcode*, *find_definition*, and the Model component. They represents the time spent on validating the patch, viewing the code, finding the definition, and coummunicating with the LLM, respectively. As for overall repair time (the total time across the four subcategories), GPT-4 Turbo is the fast, only 28.7 minutes, while the other three models take approximately 45 minutes. This suggests that GPT-4 Turbo may be a preferred choice for time-sensitive program repair tasks. Analyzing the time distribution among the subcategories, for all models, the *validate* accounts for between 74% and 82%, followed by the model time which is between 17% and 27%, *viewcode* is around 1%, and *find_definition* is lower than 0.7%. Notably, across all models, the majority of the time is spent in the *validate*. Optimizing the time spent on *validate* could be a key focus of our future work.

8 Discussion and Limitation

Scope & Generalization. The current evaluation demonstrates PATCHAGENT's repair performance for C/C++ programs. We focus exclusively on C/C++ because memory errors in these programs represent some of the most prevalent and dangerous vulnerabilities. According to statistical data reported by Google [26], approximately 70% of high-severity bugs are memory errors, often resulting from issues with C/C++ pointers. It is also worth noting that a significant body of research [49, 50, 53, 113] has proposed various methods to manipulate memory and compromise real-world programs. Moreover, by analyzing critical control information and structures [15, 48, 99], researchers have explored the auto-

generation of exploitation code, leading to outcomes such as denial-of-service (DoS), information leaks, and privilege escalation. Consequently, we chose C/C++ as the initial target for PATCHAGENT. However, We believe PATCHAGENT can handle various types of vulnerabilities and languages. Supporting new languages only requires replacing the LSP (a universal protocol for 50+ languages) backend. In fact, the versatility of LSP is why PatchAgent uses it. Supporting new vulnerability types requires implementing new parsers to purify vulnerability-specific reports.

Limited Validation. Our patch validation method employs security and functional tests, a widely adopted practice in software development, such as github CI [25]. While this method is effective and scalable for addressing many vulnerability repair scenarios, it has notable limitations. From a security standpoint, prior works [100] have revealed that approximately 5% of security patches written by human in the Linux kernel may not fully mitigate the vulnerabilities they aim to address, which suggests that patches generated by AI agents may also contain similar issues. Regarding functionality, some projects often update or expand their test cases alongside patches, which hints that simply using existing functional tests may not be sufficient. Additionally, patch correctness is influenced by factors beyond security and functionality, such as performance optimization, system compatibility, long-term maintainability, and specific requirements of downstream applications or users. These additional considerations highlight the complexity of comprehensive patch validation. While our approach provides a solid foundation, we acknowledge that it may not capture all critical aspects, underscoring the need for complementary methods and more holistic evaluation frameworks in certain contexts. We will consider these aspects in future work.

Future Improvement. To enhance the overall performance of PATCHAGENT, we recommend focusing on research in patch validation and semantic extraction. Current validation methods may yield false negatives, and fuzzing remains a primary method for vulnerability discovery. Therefore, we propose leveraging LLMs to analyze patches and synthesize specialized seeds for fuzzing. Recent work [19, 56, 101] has demonstrated that LLMs can provide more efficient mutations, enhancing vulnerability detection capabilities. A significant challenge in patch functional validation is the often poorly defined expectations of program functions, which hampers comprehensive functional testing. A potential solution is to incorporate LLMs into the development workflow [83], using them to generate functional tests that ensure every line of code is covered. With more functional test cases during the patch validation stage, the likelihood of detecting incorrect yet plausible patches increases. Additionally, if LLMs can accurately understand the semantic intent of the code, they are more likely to generate correct and elegant patches. Fine-tuning [14] is a promising approach. By training an LLM for a specific type of application, we can enhance its understanding of particular types of programs, thereby improving patch generation and overall system performance.

9 Related Work

Vulnerability Mitigation. In addition to addressing vulnerabilities in source code, numerous prior works have mitigated the impact of specific vulnerabilities through memory defense mechanisms, particularly in defending against temporal and spatial memory vulnerabilities. For temporal vulnerabilities, methods such as garbage collection based [7, 22] and pointer invalidation [47, 51] have been employed to clear dangling pointers, while one-time allocators [98, 109] have been used to prevent memory reuse. Regarding spatial memory vulnerabilities, prior works [21, 61, 112] typically involve storing additional metadata and instrumenting the program to insert boundary-checking instructions. However, these methods generally ensure only that the program is not exploitable, leaving the potential for attackers to still conduct DoS attacks.

LLM for Code. The use of LLMs in code-related tasks has gained significant attention in recent years. One important domain for code tasks is resolving functional issues. Devin [6] pioneered the use of LLMs in resolving functional issues by deconstructing user requirements and employs various tools to accomplish the task. Notably, it was the first to achieve over a 10% success rate on SWE-bench [37]. Following this, SWE-Agent [110] and AutoCodeRover [115] adopted a similar design, enabling effective interaction with the codebase to address issues. These agents are primarily focused on resolving functional issues in Python programs, while our work is centered on repairing the security vulnerabilities of programs written in low-level languages. Another important application is vulnerability detection [20, 90]. By analyzing the target function, LLMs can identify patterns of vulnerabilities. Code generation [42, 75] is also an important application of LLMs. This task is particularly valuable for modernizing legacy systems and integrating software components written in different languages.

10 Conclusion

In this work, we introduced PATCHAGENT, an APR tool designed to automate the end-to-end process of repairing vulnerabilities in real world programs. Using the capabilities of LLMs and enhancing them with specialized modules for fault localization, patch generation, and validation, PATCHAGENT is able to emulate the decision-making process of human experts. The interaction optimizations further bolster the agent's ability to generate accurate and diverse patches. Our extensive evaluation on a diverse dataset of real-world vulnerabilities demonstrated that PATCHAGENT achieves superior performance compared to existing APR tools, successfully repairing a significant majority of vulnerabilities with high accuracy.

Ethics Considerations

In developing PATCHAGENT, an LLM-based agent designed to repair software vulnerabilities, we took careful consideration of ethical guidelines and compliance standards. The system was developed and tested in a controlled environment, specifically on a private server, ensuring that no unauthorized data or external systems (other than the LLM service providers) were involved in the process. At the dataset level, we collect the cases from open-source projects, which ensures the transparency and accessibility of all test cases.

The primary function of PATCHAGENT is to enhance software security by automatically repairing vulnerabilities, which inherently aligns with ethical goals of improving cybersecurity and protecting users from potential harm. We ensured that the agent operates within a defined scope, with no capability to perform actions outside its intended purpose, thus minimizing any risk of unintended consequences. Throughout the development and experimentation phases, all activities adhered to established ethical standards and legal requirements, confirming that our work does not violate any regulations or ethical principles.

Open Science

To facilitate reproducibility and transparency, we will publish our dataset and source code alongside our paper at https://osf.io/8k2ac. Additionally, to enable others to reproduce our results more conveniently, we plan to release our detailed running logs. These logs will include all relevant data presented in Table 1, Table 2, Table 3, and Table 5. By providing comprehensive logs, we aim to ensure that researchers can validate our findings, build upon our work, and benchmark their approaches using our results.

Acknowledgement

We appreciate our reviewers and shepherd for their valuable feedback and comments. Besides, we would like to thank open source project maintainers during the communication and interaction with real-world patches, generated by PATCHAGENT. We also received insightful advice and active help from our 42-b3yond-6ug team mates, *Yueqi Chen, Qinrun Dai, Zheyun Feng, Dang Khac Minh Le, Youngjoo Lee, Wenxuan Shi, Xinqian Sun, Kefu Wu, Yuhang Wu, Dongpeng Xu, Jun Xu, Yunhang Zhang, Qingyang Zhou*. PATCHAGENT is honored to be sponsored by, DARPA AI Cyber Challenge (AIxCC). Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author(s) and do not necessarily reflect the views of the institutions above.

References

[1] 42-B3YOND-6UG: AI Cyber Security Reasoning System. https://b3yond.org/.

[2] A fast, compressed and persistent data store library for C. https://github.com/Blosc/c-blosc2.

[3] A fast WebAssembly interpreter and the most universal WASM runtime. https://github.com/wasm3/wasm3.

[4] A review of zero-day in-the-wild exploits in 2023. https://blog.google/technology/safety-security/a-review-of-zero-day-in-the-wild-exploits-in-2023/.

[5] AddressSanitizerLeakSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer.

[6] Cognition AI. Devin: The world's first fully autonomous ai software engineer. https://cognition.ai, 2024. Early Access.

[7] Sam Ainsworth and Timothy M. Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[8] AIxCC: AI Cyber Challenge. https://www.darpa.mil/research/programs/ai-cyber.

[9] An XML toolkit implemented in C. https://gitlab.gnome.org/GNOME/libxml2.

[10] Anthropic. https://www.anthropic.com/.

[11] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. {AURORA}: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 2020.

[13] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. Understanding automatically-generated patches through symbolic invariant differences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[15] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[16] Yue Chen, Mustakimur Khandaker, and Zhi Wang. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017.

[17] CodeQL. https://codeql.github.com/.

[18] DARPA Cyber Grand Challenge Final Event Archive. http://www.lungetech.com/cgc-corpus/.

[19] Yinlin Deng, Chun Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *International Conference on Software Engineering*, 2024.

[20] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[21] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[22] Márton Erdős, Sam Ainsworth, and Timothy M Jones. Minesweeper: a "clean sweep" for drop-in use-after-free prevention. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[23] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[24] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[25] GitHub Actions documentation. https://docs.github.com/en/actions.

[26] Google Protect Zero - 0day In the Wild. https://googleprojectzero.blogspot.com/p/0day.html.

[27] GPAC is an open-source multimedia framework focused on modularity and standards compliance. . https://github.com/gpac/gpac.

[28] Zhongxian Gu, Earl T Barr, David J Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[30] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. {FreeWill}: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[31] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: Scalable, precise, and safe memory-error repair. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

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

[33] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. A survey on automated program repair techniques. *arXiv preprint arXiv:2303.18184*, 2023.

[34] huntr. https://huntr.com/.

[35] Issue 33078: wasm3:fuzzer: global-buffer-overflow. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=33078.

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

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

[38] Ma Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[39] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[40] Jan Keller and Jan Nowakowski. Ai-powered patching: the future of automated vulnerability fixes. Technical report, Technical report, 2024.

[41] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. *ArXiv*, 2024.

[42] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[43] LangChain. https://www.langchain.com/.

[44] Wei Le and Shannon D Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[45] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, 2016.

[46] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 2021.

[47] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[48] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel. In *31st Annual Network and Distributed System Security Symposium, NDSS*, 2024.

[49] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[50] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[51] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. CAMP: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[52] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[53] Ziqin Liu, Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Yalong Zou, Dongliang Mu, and Xinyu Xing. Towards unveiling exploitation potential with multiple error behaviors for kernel bugs. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[54] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[55] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakcs. In *NDSS*, 2022.

[56] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[57] Metrics | CVE. https://www.cve.org/About/Metrics.

[58] Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018.

[59] Aniruddhan Murali, Noble Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[60] Sardar Bin Murtaza, Aidan Mccoy, Zhiyuan Ren, Aidan Murphy, and Wolfgang Banzhaf. Llm fault localisation within evolutionary computation based automated program repair. In *GECCO Companion*, 2024.

[61] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[62] NVD - CVE-2022-1286. https://nvd.nist.gov/vuln/detail/CVE-2022-1286.

[63] NVD - CVE-2024-3204. https://nvd.nist.gov/vuln/detail/CVE-2024-3204.

[64] NVD - CVE-2024-34246. https://nvd.nist.gov/vuln/detail/CVE-2024-34246.

[65] NVD - CVE-2024-34249. https://nvd.nist.gov/vuln/detail/CVE-2024-34249.

[66] NVD - CVE-2024-34252. https://nvd.nist.gov/vuln/detail/CVE-2024-34252.

[67] NVD - CVE-2024-34459. https://nvd.nist.gov/vuln/detail/CVE-2024-34459.

[68] NVD - CVE-2024-41965. https://nvd.nist.gov/vuln/detail/CVE-2024-41965.

[69] NVD - CVE-2024-6063. https://nvd.nist.gov/vuln/detail/CVE-2024-6063.

[70] NVD - CVE-2024-6064. https://nvd.nist.gov/vuln/detail/CVE-2024-6064.

[71] Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/.

[72] OpenAI. https://openai.com/.

[73] OpenAI API. https://platform.openai.com/docs/models.

[74] Out-of-Bound Memory Write on "op_CopySlot_64" Function. https://github.com/wasm3/wasm3/issues/471.

[75] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. *arXiv preprint arXiv:2308.03109*, 2023.

[76] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[77] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. Benzene: A practical root cause analysis system with an under-constrained state mutation. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2023.

[78] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[79] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[80] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th international conference on software engineering*, 2014.

[81] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[82] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[83] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[84] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Conference on Annual Technical Conference*, 2012.

[85] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[86] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[87] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems*, 32, 2019.

[88] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 1981.

[89] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[90] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[91] syzkaller. https://syzkaller.appspot.com/upstream.

[92] Tool to Detect Bugs in Java and C/C++/Objective-C Code before it Ships . https://fbinfer.com/.

[93] Use After Free. https://github.com/gpac/gpac/issues/2057.

[94] Use-After-Free in ForEachModule. https://github.com/wasm3/wasm3/issues/458.

[95] Vim is a greatly improved version of the good old UNIX editor Vi. Many new features have been added: multi-level undo, syntax highlighting, command line history, on-line help, spell checking, filename completion, block operations, script language, etc. https://github.com/vim/vim.

[96] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[97] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013.

[98] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free attacks with fast forward allocation. In *Proceedings of 30th USENIX Security Symposium*, 2021.

[99] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[100] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating security risks in linux with {KLAUS}: A method for evaluating patch correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[101] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.

[102] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

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

[104] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[105] Yunlong Xing, Shu Wang, Shiyu Sun, Xu He, Kun Sun, and Qi Li. What {IF} is not enough? fixing null pointer dereference with contextual check. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[106] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the negative force: Efficient vulnerability {Root-Cause} analysis through reinforcement learning on counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[107] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. Aligning llms for fl-free program repair. *ArXiv*, 2024.

[108] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[109] Carter Yagemann, Simon Pak Ho Chung, Brendan Saltaformaggio, and Wenke Lee. Pumm: Preventing use-after-free using execution unit partitioning. In *Proceedings of 32nd USENIX Security Symposium*, 2023.

[110] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[111] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[112] Zheng Yu, Ganxiang Yang, and Xinyu Xing. ShadowBound: Efficient memory protection through advanced metadata management and customized compiler optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[113] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Retspill: Igniting user-controlled data to burn away linux kernel protections. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[114] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 2023.

[115] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.

[116] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. {FIXREVERTER}: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

```
1  _onfatal:
2      if (result) {
3          fprintf (stderr, "Error: %s", result);
4          if (runtime)
5          {
6              M3ErrorInfo info;
7              m3_GetErrorInfo (runtime, &info);
8              if (strlen(info.message)) {
9                  fprintf (stderr, " (%s)", info.message);
10             }
11         }
12         fprintf (stderr, "\n");
13     }
14     m3_FreeRuntime (runtime);
15     m3_FreeEnvironment (env);
```

Listing 3: Code Snippet of CVE-2024-34246.

```
1 else {
2   if (MRB_METHOD_NOARG_P(m)) {
3     check_method_noarg(mrb, ci);
4   }
5   recv = MRB_METHOD_FUNC(m)(mrb, recv);
6 }
```

Listing 4: Code Snippet of CVE-2022-1286.

A Appendix

A.1 Case Studies

In this section, we present three case studies: one case (Issue-2057 [93]) that PATCHAGENT successfully repaired, and two cases (CVE-2024-34246 [64], CVE-2022-1286 [62]) that PATCHAGENT was unable to address.

Issue-2057. Issue-2057 is a use-after-free bug in the gpac project. This bug arises from the lifecycle management of objects in the struct list `codec->QPs`, which involves multiple references. The victim object within `codec->QPs` can be accessed through various references. When the status of the victim object changes, such as transitioning from allocated to de-allocated, it is critical to synchronize its status across all references to prevent inconsistent behavior. In this case, when the victim object is freed via reference A, subsequent operations continue to use reference B to interact with the victim object. PATCHAGENT successfully identifies the potential relationships between these different references and inserts multiple checks across three different functions to ensure the validity of those references. If a reference fails validation, PATCHAGENT returns null or error codes to notify upper-level functions of the latent error.

CVE-2024-34246. The code snippet relevant to CVE-2024-34246 is presented in Listing 3. The sanitizer flags a null dereference at line 8, where `info.message` is a null pointer, leading to an error when `strlen` is invoked. A straightforward approach to repair this vulnerability might involve adding a

Project	Issue ID	Bug Type	Status
assimp	5763	Heap Overflow	✓ Merged
assimp	5764	Stack Overflow	✓ Merged
assimp	5765	Null Dereference	✓ Merged
hdf5	5201	Heap Overflow	✓ Merged
hdf5	5202	Heap Overflow	◆ Open
hdf5	5209	Heap Overflow	◆ Open
hdf5	5210	Heap Overflow	✓ Merged
libredwg	1061	Use After Free	✓ Merged
Pcap++	1678	Heap Overflow	◆ Open
Pcap++	1680	Heap Overflow	✓ Merged

Table 6: **Github Pull Requests.**

Range	Prop (%)	Range	Prop (%)
< 400	22.46%	1600–3200	20.79%
$400 - 800$	30.90%	3200–6400	7.30%
$800 - 1600$	17.98%	> 6400	0.56%

(a) Related Lines of Code (RLOC)

Range	Prop (%)	Range	Prop (%)
< 16	29.78%	26–30	12.36%
16–20	15.73%	31–35	8.99%
21–25	11.24%	> 35	21.92%

(b) Backtrace Depth at Crash Site

Range	Prop (%)	Range	Prop (%)
$< 10^5$	0.57%	$10^7 - 10^8$	67.05%
$10^5 - 10^6$	1.14%	$10^8 - 10^9$	27.27%
$10^6 - 10^7$	0.57%	$> 10^9$	3.41%

(c) Instruction Number of Trace

Table 7: **Dataset Complexity Metrics.**

Project	Lang	Source	LoC	#Vulns	#Test
assimp	C++	OSS-Fuzz	347.0K	3	474
c-blosc	C	OSS-Fuzz	88.8K	2	1643
c-blosc2	C++	OSS-Fuzz	117.1K	7	1284
h3	C	OSS-Fuzz	17.2K	1	124
hoextdown	C	OSS-Fuzz	7.1K	1	83
hostap	C	OSS-Fuzz	438.0K	4	19
htslib	C	OSS-Fuzz	66.5K	1	159
hunspell	C++	OSS-Fuzz	83.9K	11	128
irssi	C	OSS-Fuzz	64.4K	3	5
krb5	C	OSS-Fuzz	301.6K	1	125
libplist	C	OSS-Fuzz	12.1K	3	34
libsndfile	C	OSS-Fuzz	56.4K	5	141
libtpms	C	OSS-Fuzz	115.0K	1	6
libxml2	C	OSS-Fuzz	200.4K	10	3272
lz4	C	OSS-Fuzz	18.6K	2	22
md4c	C	OSS-Fuzz	8.0K	4	24
openexr	C++	OSS-Fuzz	227.8K	3	111
sleuthkit	C	OSS-Fuzz	196.2K	5	2
wasm3	C	OSS-Fuzz	22.8K	8	35062
zstd	C	OSS-Fuzz	93.4K	5	28
gpac	C	Huntr	743.7K	32	711
libmobi	C	Huntr	19.1K	5	12
mruby	C	Huntr	62.2K	10	61
radare2	C	Huntr	841.3K	20	858
yasm	C	Huntr	132.1K	3	44
binutils	C	ExtractFix	666.8K	2	20
coreutils	C	ExtractFix	86.1K	4	573
jasper	C	ExtractFix	44.7K	2	16
libjpeg	C	ExtractFix	46.9K	4	530
libtiff	C	ExtractFix	85.9K	11	74
libxml2	C	ExtractFix	200.4K	5	3272

Table 8: **Evaluation Dataset.** The **LoC** column shows the lines of code for each project, the **#Vuln** column displays the number of vulnerabilities for the corresponding project, and the **#Test** column indicates the number of functional tests for each project.

null check before the dereference. In fact, we observed that PATCHAGENT attempted a similar solution. However, after inserting the check for `info.message`, the sanitizer raises a different error, indicating that this repair strategy is not feasible. Through manual analysis of the vulnerability, we found that the bug was caused by an incorrect error handling method. When the program encounters malicious input, an error may be raised during the initialization process. The same error handling logic is then used to manage it (line 1), intending to report the error and release resources. However, different internal exceptions indicate different resource states. Therefore, to properly repair the error, it is necessary to examine all sites where the error was raised. Given the complexity of this task, it is understandable that PATCHAGENT was unable to patch this vulnerability at this stage.

CVE-2022-1286. This bug is a heap overflow vulnerability discovered in the mruby project, which PATCHAGENT was unable to repair using any of its models. The primary challenge in addressing this case is the project's extensive use of indirect calls. If the target of an indirect call does not appear in the stack trace, it becomes difficult for LLMs to determine the correct target without additional runtime information. Listing 4 illustrates such an example, specifically in Line 5. This indirect invocation requires dynamic knowledge of the program's current state rather than relying solely on static information. Unfortunately, the root cause of this vulnerability does not appear in the stack trace and is the target of an indirect call, while the language server of PATCHAGENT can only provide static information. Although PATCHAGENT successfully analyzed the code at the call site of the root cause function, it could not proceed with further analysis due to the lack of runtime information. Consequently, we believe that

System Prompt

Your task is to patch the bug in the program as identified by the sanitizer report. Access the buggy codebase and the corresponding sanitizer report highlighting various issues.
.....
Begin by reviewing the sanitizer report to identify specific problems, such as null pointer dereferences, buffer overflows, or use-after-free. Then, delve into the codebase to locate the exact code sections where these issues occur.
.....
After identifying solutions, modify the code accordingly, ensuring adherence to code best practices. Test your patches thoroughly to confirm resolution of issues without introducing new ones.
.....

User Prompt

I will send you the sanitizer report for our program. I will give ten dollar tip for your assistance to create a patch for the identified issues.
.....
Now I want to patch the {program}, the tag is {tag}, here is the asan report {report}. The report provides the stack trace of the program. You can use the stack trace to identify a fix point for the bug.
.....
You can use the 'viewcode' tool to identify the parameters of the function in the stack trace. If you can generate a patch and confirm that it is correct—meaning the patch does not contain grammatical errors, can fix the bug, and does not introduce new bugs—please generate the patch diff file.
.....
{counterexamples}

Figure 5: **System and User Prompt of PATCHAGENT.**

dynamic information is crucial for vulnerability repair and will consider it in future work.

A.2 Github Pull Requests

Table 6 summarizes the 10 pull requests (PRs) we submitted across 4 projects to address real-world vulnerabilities, including 7 cases of heap overflows, 1 case of stack overflow, 1 case of use after free, and 1 case of null dereference. Among these, 7 PRs were successfully merged, while the remaining 3 are still under review. The table provides details such as the project name, PR ID, the type of bug targeted, and the current status, demonstrating the practical impact of our tool in identifying and resolving critical vulnerabilities in widely used software.

A.3 Dataset

The dataset covers 30 projects as shown in Table 8. Each case is accompanied by a reproduce script and a functional test script. The table also shows the lines of code (LoC) and the number of functional tests for each project. To better understand the complexity of vulnerabilities in the dataset, we collected statistics on the lines of code within the functions present in the crash stack traces for each case, referring to these as the related lines of code (RLOC). The distribution of RLOC is show in Table 7a. It is important to note that the vulnerability is not necessarily confined to these specific lines; it may also involve functions not appearing in the stack trace. Our motivating example in §3 illustrates such a scenario. Also, we collect the distribution of backtrace depth at the crash site and instruction number of trace, which are shown in Table 7b and Table 7c, respectively.

A.4 Prompt

The system prompt and user prompt of PATCHAGENT are shown in Figure 5.