NORTHWESTERN UNIVERSITY

Toward Practical Vulnerability Repair

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Zheng Yu

EVANSTON, ILLINOIS

March 2026

ABSTRACT

The increasing scale and complexity of modern software have led to a surge in security vulnerabilities, outpacing the ability of developers to provide timely repairs. While Large Language Models (LLMs) have catalyzed advancements in Automated Vulnerability Repair (AVR), existing approaches often struggle in real-world settings due to an over-reliance on idealized assumptions, such as perfect fault localization and patch validation methodologies. This dissertation addresses these gaps by developing autonomous, LLM-based systems designed to navigate the end-to-end complexities of the repair process, including 0-day vulnerability mitigation, legacy system maintenance, and rigorous correctness evaluation.

To improve the practicality of AVR, we first introduce *PatchAgent*, an autonomous agent that integrates fault localization, patch generation, and patch validation. By utilizing a language server and several interaction optimizations, it mimics human-like reasoning to repair vulnerabilities triggered by proof-of-concept (PoC) inputs, achieving a 90% success rate on real-world datasets. Extending these capabilities to 1-day vulnerabilities, we present *PortGPT*, a system designed to automate patch backporting for large-scale projects like the Linux kernel. By autonomously navigating Git history and refining patches based on compiler feedback, *PortGPT* demonstrates high efficacy in migrating security fixes to older software branches, with several generated patches successfully merged into the Linux mainline.

Finally, to address the lack of reliable evaluation standards in the field, we present PVBench, a benchmark designed for the rigorous assessment of patch correctness. Our analysis reveals that existing AVR systems frequently produce ”overfitted” patches that pass basic tests but fail to capture the original developer’s intent or the true root cause of the vulnerability. By introducing PoC^{+} tests that encode complex semantic requirements, we demonstrate that current success rates are

often significantly overestimated. Collectively, this work provides a comprehensive framework for building and evaluating AVR systems that are robust, autonomous, and applicable to the demands of real-world software maintenance.

ACKNOWLEDGEMENTS

todo

THESIS COMMITTEE

Xinyu Xing

Northwestern University

Committee Chair

Peter Dinda

Northwestern University

Committee Member

Yan Chen

Northwestern University

Committee Member

Kexin Pei

University of Chicago

Committee Member

TABLE OF CONTENTS

Acknowledgments 4

List of Figures 13

List of Tables 15

Chapter 1: Introduction 18

Chapter 2: PatchAgent: A Program Repair Agent Mimicking Human Expertise 22

2.1 Introduction 22

2.2 Background on Automated Vulnerability Repair 26

2.2.1 Workflow for PoC-driven AVR 27

2.2.2 LLMs and Their Applications in AVR 28

2.3 Motivation: Human vs Vanilla LLM Agent 30

2.3.1 Motivating Example 31

2.3.2 Reflection on Both Processes 33

2.4 The PATCHAGENT Design 34

2.4.1 Framework 35

2.4.2 Incorporating Human Expertise . . . 36
2.4.3 Prompt Design . . . 37
2.5 Interaction Optimization . . . 38
2.5.1 Report Purification . . . 38
2.5.2 Chain Compression . . . 38
2.5.3 Auto Correction . . . 41
2.5.4 Counterexample Feedback . . . 43
2.6 Implementation . . . 45
2.7 Evaluation . . . 46
2.7.1 Setup . . . 46
2.7.2 Effectiveness of PATCHAGENT . . . 50
2.7.3 Ablation Study . . . 52
2.7.4 Repair Unseen Vulnerability . . . 53
2.7.5 Efficiency of PATCHAGENT . . . 55
2.7.6 Case Studies . . . 57
2.8 Discussion and Limitation . . . 59
2.9 Related Work . . . 61
2.10 Conclusion . . . 62

Chapter 3: PortGPT: Towards Automated Backporting Using Large Language Models 63

3.1 Introduction . . . 63

3.2 Why LLMs for Backporting? . . . 67
3.2.1 Problem Description . . . 67
3.2.2 Vanilla Text-based Backporting . . . 68
3.2.3 Syntax-based Backporting . . . 69
3.2.4 Semantics-based Backporting . . . 70
3.2.5 LLM-based Backporting . . . 71
3.2.6 Backporting Types . . . 72
3.3 Prompting (In-context Learning) Is Not All . . . 73
3.3.1 Motivating Example . . . 74
3.3.2 Observations . . . 75
3.4 Design Details of PORTGPT . . . 76
3.4.1 Overview . . . 77
3.4.2 Per-Hunk Adaptation . . . 78
3.4.3 Final Patch Combination . . . 83
3.4.4 Prompt Design . . . 85
3.5 Implementation . . . 87
3.6 Evaluation . . . 88
3.6.1 Settings . . . 88
3.6.2 Performance Evaluation (RQ1) . . . 91
3.6.2.1 Comparison with FIXMORPH . . . 92

3.6.2.2 Comparison with TSBPORT 93
3.6.2.3 Generalizability over other programming languages 94
3.6.2.4 Effectiveness of other large language models 95
3.6.2.5 Impact of Patch Size Distribution 96
3.6.3 Ablation Study (RQ2) 97
3.6.4 Efficiency Evaluation (RQ3) 98
3.6.5 Real-World Applicability Study (RQ4) 99
3.6.6 Case Studies 100
3.7 Discussion and Limitations 103
3.8 Related Work 104
3.9 Conclusion 105

Chapter 4: Patch Validation in Automated Vulnerability Repair 107

4.1 Introduction 107
4.2 Background: Patch Validation in AVR 111
4.3 Motivation: PoC$^+$ Test 114
4.3.1 Plausible Patch 115
4.3.2 PoC$^+$ Tests Reveal the Difference 116
4.4 PoC$^+$ Test Dataset & Validation 119
4.4.1 PVBENCH Overview 119
4.4.2 Validation & Production Methodology 120

4.5 Quantifying Overestimation . . . 124
4.5.1 Methodology . . . 124
4.5.2 General Results . . . 125
4.6 Reliability of PoC$^+$ Test . . . 129
4.6.1 Methodology . . . 129
4.6.2 Performance Issue . . . 130
4.6.3 Suboptimal Repair . . . 132
4.6.4 Check Circumvention . . . 134
4.7 False Positive Analysis . . . 135
4.7.1 Incorrect Root Cause . . . 135
4.7.2 Specification Violation . . . 137
4.7.3 Poor Code Practice . . . 138
4.8 Implications for AVR Research . . . 140
4.9 Discussion . . . 141
4.10 Related Work . . . 142
4.11 Conclusion . . . 143

Chapter 5: Conclusion and Future Work . . . 145

5.1 Conclusion . . . 145
5.2 Funture Work . . . 146

References 148

LIST OF FIGURES

2.1 **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. 31

2.2 **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. 34

2.3 **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. 39

2.4 **Average Time Cost of PATCHAGENT.** 55

3.1 **Workflow of PORTGPT** 77

3.2 **Example of Per-Hunk Adaptation.** The red arrows represent LLM agent inputs, while the blue represent LLM agent outputs. The green box represents the invocations of LLM, while the blue box represents the outputs of tools. 82

3.3 Patches Generated for CVE-2023-52752. 84

3.4 **User Prompt of PORTGPT** 86

4.1 Type Confusion Bug in PHP Project 114

4.2 Two distinct patch strategies for the type confusion vulnerability in PHP's `range()` function. 115

4.3 An PoC$^+$ test derived from the PoC for PHP issue #13094, validating correct behavior for a crashing input. 117

4.4 PoC$^+$ Test Examples for Output Checking 120

4.5 PoC$^+$ Test Example for Intermediate Checking 123

4.6 PoC$^+$ Test Example for Self Checking 124

4.7 Comprehensive evaluation results across 209 vulnerability cases. 126

4.8 Distribution of vulnerabilities: The x-axis shows #patches passing PoC$^+$ tests, while the y-axis shows #patches passing basic tests but failing PoC$^+$ tests. 127

4.9 Example of Performance Issue 131

4.10 Example of Suboptimal Repair 133

4.11 Incorrect Root Cause Example 136

4.12 Domain Ignorance Example 138

4.13 Logic Shortcuts Example 139

LIST OF TABLES

2.1 **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. . . . 47

2.2 **Dataset Complexity Metrics.** 47

2.3 **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. 49

2.4 **Comparison of vulnerability repair results between ExtractFix (E.), Zero-Shot (Z.), and PATCHAGENT (P.).** ● 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. 51

2.5 **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. 52

2.6 **Summary of unseen vulnerability repair results. P.** 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. 54

2.7 **Github Pull Requests.** 55

2.8 **Token & Money Cost of PATCHAGENT under Different LLM.** 55

3.1 **Overview of Our Dataset.** Type-II: 19, Type-III: 49, Type-IV: 78, Total: 146. . . . 89

3.2 **Performance Comparison of Backporting.** The first two sections of the table compare PORTGPT with FIXMORPH and TSBPORT using datasets from prior works as well as C cases on our own dataset. The last two rows display the performance of PORTGPT on C++ and Go cases from our dataset. ChatGPT† is from TSBPORT [124]. **TSBPORT*** represents the performance of TSBPORT under the same settings as PORTGPT and FIXMORPH. Unlike PORTGPT and FIXMORPH, TSBPORT requires the target file for each hunk as input, which may not be available in real-world scenarios. To ensure a fair comparison, we introduce **TSBPORT*** as a variant of TSBPORT that operates without access to the target file, simulating a more realistic environment. 91

3.3 **PORTGPT Performance based on Different Large Language Model.** 92

3.4 **Performance Analysis by Patch Size.** Success rates of different tools across varying patch sizes (measured in modified lines) on prior datasets from and our new C dataset. Numbers indicate successful patches out of total attempts with success percentages in parentheses. 96

3.5 **Ablation Study of PORTGPT.** w/o = without. GitUtils, Locate (LocateSymbol) and Compile (CompileTest) are three tools in PORTGPT. AutoFix refers to the patch correction mechanism. 97

3.6 **Average Time & Money Cost of PORTGPT.** 98

3.7 **Ubuntu Backporting Tasks Results. Version** refers to the target version for Ubuntu backporting tasks. **Fix Date** represents the actual date of the backport in the real world. 99

4.1 Taxonomy of Patch Validation Methodologies Used in Previous AVR Works 111

4.2 Overview of Projects and Vulnerabilities in PVBENCH 118

4.3 PoC^{+} Test Category Distribution by Projects 119

4.4 Script Path of Projects that Support PoC^{+} Generation 121

4.5 Performance of AVR tools under different validation. **init**: patches passing basic tests; **poc+**: patches also passing PoC^{+} tests; **FDR**: false discovery rate, i.e., the fraction of initially validated patches that fail subsequent testing. 124

4.6 Manual Categorization of Patches that Passed PoC^{+} Tests Across Different AVR Tools: The table shows the distribution of four quality categories Results are compared across three tools using two LLMs 129

4.7 Categorization of FP Patches: The table presents a breakdown of FP patches across three AVR systems using two LLMs. 134

CHAPTER 1

INTRODUCTION

As the size and complexity of programs continue to grow, they also inevitably become more vulnerable, as evident in google reports [1] and CVE records [2]. The development of fuzzing techniques accelerated vulnerability discovery. Fuzzing platforms such as OSS-Fuzz [3] and Syzkaller [4] now identify thousands of vulnerabilities in common software and systems. This rapid increase in reported vulnerabilities has resulted in a large number of vulnerabilities remaining unrepaired for long periods. If these vulnerabilities cannot be repaired promptly, these program vulnerabilities cannot be effectively mitigated.

Automated Vulnerability Repair (AVR), a specialized subset of Automated Program Repair (APR), aims to automatically fix security vulnerabilities and has gained significant attention in recent years [5], [6], [7]. When combined with Large Language Models (LLMs) [8], [9], [10], these approaches have shown promising results in repairing vulnerabilities. However, existing research often overlooks real-world settings, limiting the practical applicability of these techniques. A typical AVR workflow comprises three stages: fault localization, patch generation, and patch validation. Current AVR techniques frequently neglect the fault localization and patch validation stages. Specifically, many existing approaches assume that fault location results are provided, with some even assuming perfect fault localization, an unrealistic assumption in practice. Furthermore, existing techniques typically rely on weak patch validation methods (e.g., test suite-based validation) or manual validation that does not scale. These limitations hinder the applicability of existing LLM-based AVR techniques in real-world scenarios.

The above issues motivate us to design a practical program repair system by carefully consid-

ering the real world AVR requirements. In the real world, the most common 0-day vulnerability source is fuzzing tools, which provide concrete inputs that trigger vulnerabilities (i.e., proof-of-concept (PoC) inputs). In such scenarios, we can only leverage the PoC inputs for fault localization. To address these challenges, we build LLM-based AVR systems that can dynamically interact with the program under repair, mimicking human-like reasoning during the repair process. Another important vulnerability repair is to repair 1-day vulnerabilities, which have already been fixed in the mainline branch but need to be backported to older branches. This task is essential for maintaining popular open-source projects (e.g., the Linux kernel). To address this problem, we design an workflow that can autonomously backport patches by accessing code on-demand, summarizing Git history, and revising patches based on feedback (e.g., from compilers). Finally, to ensure the reliability of AVR evaluation, we studied the patch validation problem in depth and constructed a benchmark for rigorous evaluation of patch correctness in AVR systems. The rest of this dissertation is organized as follows.

Chapter 2 presents PATCHAGENT, a novel LLM-based automated vulnerability repair (AVR) tool that seamlessly integrates fault localization, patch generation, and validation within a single autonomous agent. PATCHAGENT addresses a common program repair scenario: given a concrete input that triggers a vulnerability, how can the program be patched without breaking existing tests? To tackle this challenge, 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 AVR tools. Our ablation study further provides insights into how various interaction optimizations contribute to PATCHAGENT's effectiveness.

Chapter 2 appeared in *the Proceedings of 34th USENIX Security Symposium (USENIX Security 2025)* [11].

Chapter 3 presents PORTGPT, an LLM-based agent for end-to-end automation of patch backporting in real-world scenarios. Patch backporting, the process of migrating mainline security patches to older branches, is an essential task in maintaining popular open-source projects (e.g., the Linux kernel). However, manual backporting can be labor-intensive, while existing automated methods, which heavily rely on predefined syntax or semantic rules, often lack agility for complex patches. PORTGPT enhances an LLM with tools to access code on-demand, summarize Git history, and revise patches autonomously based on feedback (e.g., from compilers), thereby simulating human-like reasoning and verification. PORTGPT achieved an 89.15% success rate on existing datasets (1,815 cases) and 62.33% on our own dataset of 146 complex cases, both outperforming state-of-the-art backporting tools. We contributed 9 backported patches generated by PORTGPT to the Linux kernel community, and all patches have been merged.

Chapter 3 appeared in the *Proceedings of 47th IEEE Symposium on Security and Privacy (IEEE S&P 2026)* [12].

Chapter 4 presents PVBench, a benchmark for rigorous evaluation of patch correctness in automated vulnerability repair (AVR) systems. Automated Vulnerability Repair (AVR) systems, especially those leveraging large language models (LLMs), have demonstrated promising results in patching vulnerabilities—that is, if we trust their patch validation methodology. Ground-truth patches from human developers often come with new tests that not only ensure mitigation of the vulnerability but also encode extra semantics such as root cause location, optimal fix strategy, or subtle coding styles or conventions. And yet, none of the recent AVR systems verify that the auto-generated patches additionally pass these new tests (termed as PoC$^+$ tests). This is a subtle yet critical omission. To fill this gap, we constructed a benchmark, PVBench, with 209 cases spanning 20 projects. Each case includes basic tests (functional tests before the patch and the PoC exploit) as well as the associated PoC$^+$ tests. Evaluated on three state-of-the-art AVR systems, we find

that over 40% of patches validated as correct by basic tests fail under PoC$^+$ testing, revealing substantial overestimation on patch success rates. Analyzing these patches that are falsely labeled as correct, we suggest that AVR tools should improve in three critical areas: root cause analysis, adherence to program specifications, and capturing developer intention.

Chapter 5 concludes the dissertation and discusses future research directions.

CHAPTER 2

PATCHAGENT: A PROGRAM REPAIR AGENT MIMICKING HUMAN EXPERTISE

2.1 Introduction

As programs grow in size and complexity, they also become increasingly vulnerable, as evident in reports [1] and CVE records [2]. For example, fuzz testing (a.k.a. fuzzing) alone has enabled the discovery of thousands of vulnerabilities in large and complex software [3], [4], not to mention other program analysis tools [13], [14] 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 [15], [16], [17], [18] or software fault isolation (SFI) [19], [20], [21], [22] have been proposed to mitigate certain types of vulnerabilities, these techniques can run into performance or compatibility issues [23]. More importantly, they do not aim to fix the code logic that causes the vulnerability. In contrast, Automated Vulnerability Repair (AVR) [5]—especially source code-based AVR tools—aim to patch the buggy code directly without distorting functionalities nor introducing unnecessary overhead. Effective AVR tools can significantly reduce if not eliminate manual effort in patching a security vulnerability [24], and hence, may help shorten the time frame between vulnerability discovery and fix rollout.

Over the past decade, AVR has received much attention from researchers [5], [6], [7], 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 [25], idiomaticity [26], or functional specifications [27]). More recently, the research on patch generator has entered the era of large language models (LLMs) [8], [9], [10], especially when LLM-based patch generators have outperformed conventional ones in results [10].

However, patch generation is only a midstream task in AVR. Most patch generators require effective fault localization (FL)—some even assume perfect FL [8], [28], [29] —to pinpoint the buggy code snippet. This introduces two challenges when applying end-to-end AVR to real-world software: 1) FL techniques based on static analysis are prone to high false positive rates [30], 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 [31], [32], ruling out straightforward adoption of program slicing techniques [33], [34] .

The downstream task of patch generation is patch validation, which is often left to either manual review [35] or automated testing [36] where in the latter case a patch is considered "correct" when all the tests pass, including the mitigation of the PoC, if exists. While this is arguably the state-of-the-practice [35], [37], [38] 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 AVR and propose PATCHAGENT, a **push-button AVR** tool that handles FL, patch generation, and patch validation in an integrated LLM agent which manages the entire context during AVR. 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 Section 2.3), which could be caused by the contrast that bug triaging and patching usually involves heavy code analysis [39], [40] while LLMs do not have robust reasoning capabilities [41]. 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 Section 2.7.3.

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 [3], Huntr [42] and ExtractFix [43] 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 [44] and the Claude-3 series from Anthropic [45]. 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 AVR methods (ExtractFix [43] and Pearce et al. [36]) 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, after PATCHAGENT was built, tested, and documented in a research article, a subset of the authors, together with some members of team 42-b3yond-6ug [46], further customized and re-engineered it, leveraging part of its capabilities to participate in the DARPA AI Cyber Challenge (AIxCC) [47], where the team advanced to the finals.

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.

2.2 Background on Automated Vulnerability Repair

Automated Vulnerability Repair (AVR) 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 AVR approaches adhere to this setting; many rely on static analysis [39], [40] or exact fault localization [8], [28], [29]. 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 [3] and syzkaller [4].

2.2.1 Workflow for PoC-driven AVR

Under this setting, the AVR 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 [48], [49], [50] 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 [31], [32], [51], [52] 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 [53], [54], which search for a correct patch in a predefined patch space scoped by heuristics; ② constraint-based [39], [43], which employ advanced constraint solvers or program synthesis techniques to generate candidate patches that toggle the bug-triggering condition; ③ pattern-based [55], [56], 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 [57], [58], 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 AVRs primarily because fix templates are never *explicitly* defined in the process.

Patch validation. Fixation [38] uses distance-bounded weakest preconditions to identify partially fixed exceptions in Java programs. Le and Pattison [37] 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. KLAUS [35] 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 [43] is a holistic AVR 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.

2.2.2 LLMs and Their Applications in AVR

Large Language Models (LLMs) have demonstrated exceptional capabilities in various natural language processing tasks, including text classification and generation [59], [60]. By leveraging their sophisticated language modeling abilities, LLMs can generate coherent text by predicting subsequent tokens 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 [36], [61], [62].

Without doubt, LLMs fit perfect well with AVR and many applications have been proposed. As a subset of ④ learning-based patch generation techniques, LLM-based tools [10], [63] have outperformed other patch generators as reported in recent surveys [6], [64]. On the fault localization (FL) front, SemiAutoFL [65] 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 AVR system. As for patch validation, researchers explore the possibility of introducing patch validation feedback [66] into LLMs for improving AVR.

However, instead of being a holistic AVR tool, these LLM-based tools mostly focus on using LLMs to implement one component in the AVR process. Pearce et al. [36] is an LLM-based AVR 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 AVR setting. Designing new AVR workflow [67] without fault localization (free-FL), under the assistance of LLMs, is another potential way for AVR. 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.

```
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.

2.3 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.

2.3.1 Motivating Example

Issue-33078. Listing 1 presents an out-of-bound (OOB) access bug that causes issue-33078 [68], which was discovered by OSS-Fuzz [3]. The OOB access is flagged by AddressSanitizer [69] 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 2.1).

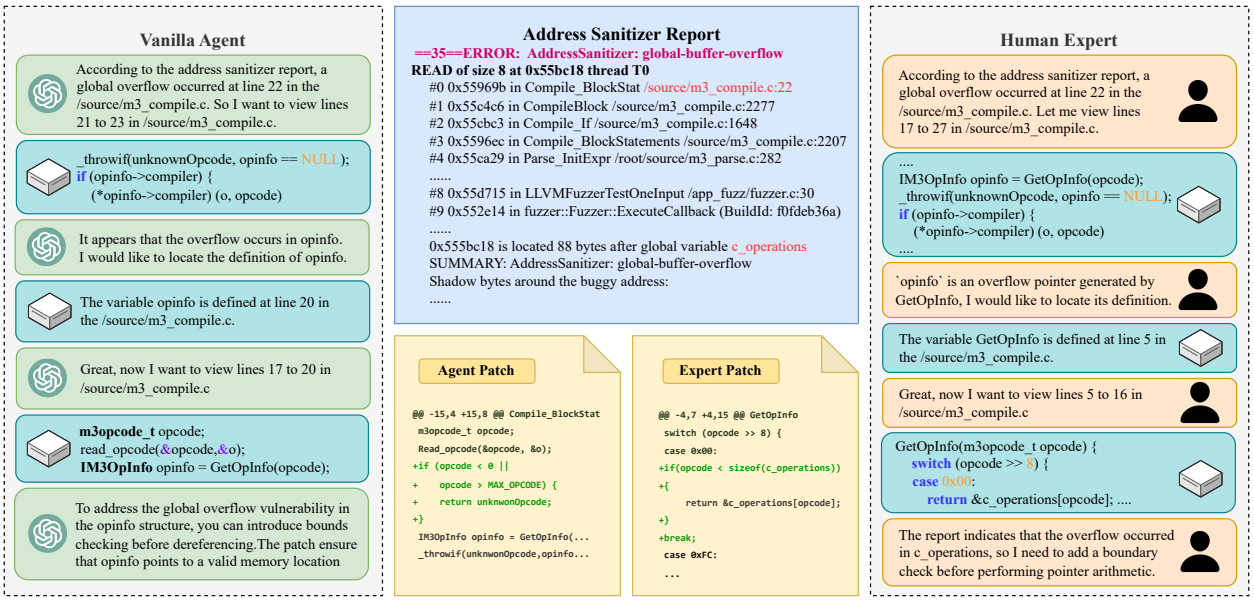

Figure 2.1: **Comparison between vanilla agent and human expert vulnerability repair processes.** The center part displays the AddressSanitizer report alongside the patches generated by both human expert and vanilla agent. The left side illustrates the repair process of vanilla agent, while the right side shows the repair process of human expert.

Repair by human. The process by which a human expert repairs the vulnerability is illustrated in the right side of Figure 2.1. 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 Section 2.4.1 for details).

One sample run of the repair process by the agent is shown on the left side of Figure 2.1. 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.

2.3.2 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 Section 2.4.2. 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.

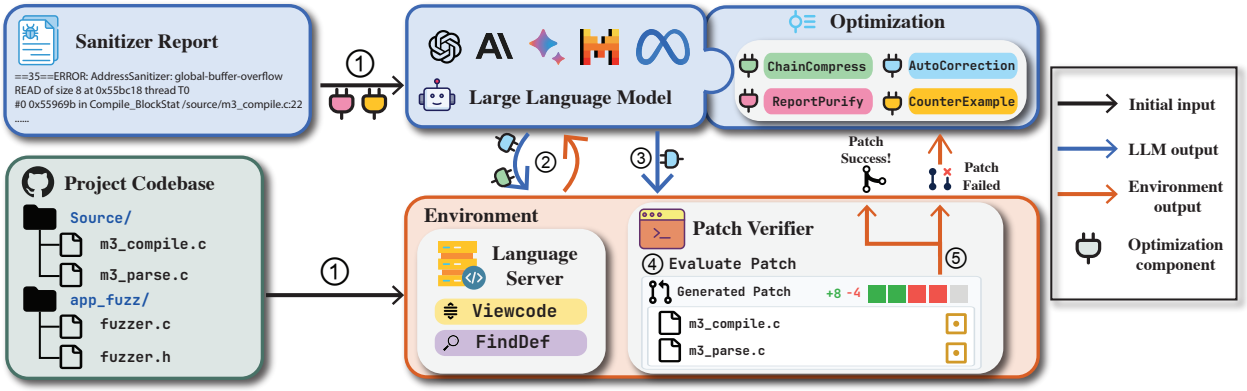

Figure 2.2: **Overview of PATCHAGENT.** The process begins with the sanitizer report and the project codebase (①). The LLM retrieves the code context using the *viewcode* and *find_definition* APIs (②) and then generates a patch (③). The patch is subsequently validated by the patch verifier (④). If the 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.

2.4 The PATCHAGENT Design

In this section, we present the design of PATCHAGENT. We start by outlining the basic framework (Section 2.4.1), followed by an overview of its optimization components (Section 2.4.2). Finally, we describe the prompt design (Section 2.4.3).

2.4.1 Framework

The overall framework of PATCHAGENT is illustrated in Figure 2.2. 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 [70] 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.

2.4.2 Incorporating Human Expertise

While the framework discussed so far outlines the basic workflow of PATCHAGENT, it does not fully address the challenges mentioned in Section 2.3. To bridge this gap and incorporate human expertise, we introduce several optimization components into the agent, represented as the plugin object in Figure 2.2. 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 pro-

ficiency 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. Addressing 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 Section 2.5, we will delve deeper into each component, explaining how they contribute to more effective program repair.

2.4.3 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.

2.5 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.

2.5.1 Report Purification

As we mentioned in Section 2.3, 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.

2.5.2 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

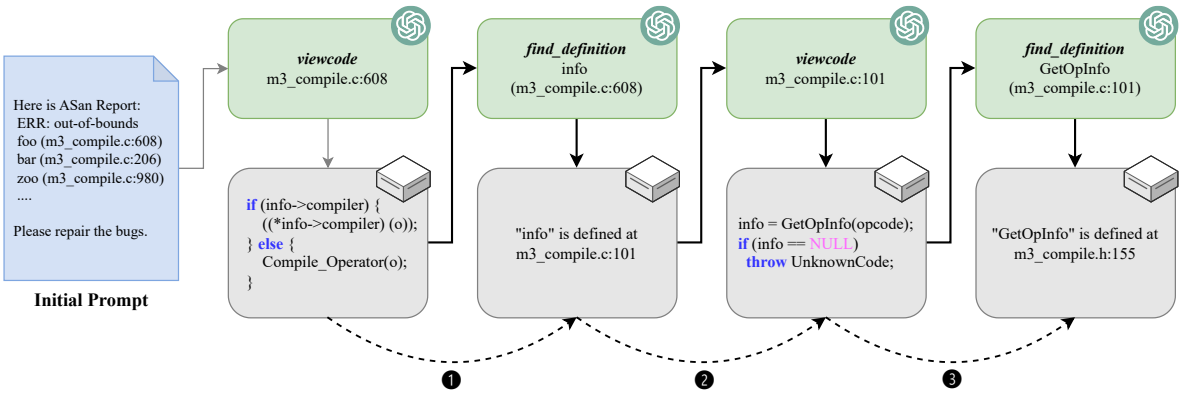

Figure 2.3: **Example of Chain Compression.** The LLM takes the initial prompt as input and starts interacting with the language server. The black bold arrows illustrate the interaction without chain compression, while the black dashed arrows represent the compressed interaction process. The original interaction chain of length four was compressed into a single interaction.

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 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 2.3 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.

2.5.3 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 numbers, 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 [64], 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 [71] 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.

2.5.4 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 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

```
  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.**

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 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.

2.6 Implementation

We implemented PATCHAGENT using LangChain [72] 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) [73], 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.

2.7 Evaluation

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

- **RQ 1:** How effectively can PATCHAGENT repair vulnerabilities in real-world programs? (Section 2.7.2)
- **RQ 2:** What is the impact of individual interaction optimization mechanisms on repair performance? (Section 2.7.3)
- **RQ 3:** How does PATCHAGENT perform when repairing vulnerabilities that LLM has never seen before? (Section 2.7.4)
- **RQ 4:** How efficient is PATCHAGENT in repairing vulnerabilities? (Section 2.7.5)

Additionally, we present and analyze several case studies in Section 2.7.6 to offer a comprehensive understanding of the effectiveness and limitations of PATCHAGENT.

2.7.1 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.

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

Table 2.1: **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.

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

(a) Related Lines of Code (RLOC)

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

(b) Backtrace Depth at Crash Site

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

(c) Instruction Number of Trace

Table 2.2: **Dataset Complexity Metrics.**

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 [43]. 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 [3] and Huntr [42]. 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 [74] and FixReverter [75] are reproducible, while CGC [76] can reproduce all cases but does not include functional test scripts. The dataset covers 30 projects as shown in Table 2.1. 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 2.2a. 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 Section 2.3 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 2.2b and Table 2.2c, respectively.

Evaluation Criteria. In our evaluation, we primarily compare our approach with ExtractFix [43] and a zero-shot method based on LLMs proposed by Pearce et al. [36]. 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 [36] or other related works [8], [28], [29] 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.

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 2.3: **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.

2.7.2 Effectiveness of PATCHAGENT

Table 2.3 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 2.3, 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 [39], [40], [43], [64], 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. In contrast, Claude-3 Sonnet lags behind, with a significantly lower repair rate of 60.67%.

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

Table 2.4: **Comparison of vulnerability repair results between ExtractFix (E.), Zero-Shot (Z.), and PATCHAGENT (P.).** ● 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.

Table 2.4 presents a comparative analysis of vulnerability repair results among three approaches: ExtractFix (**E.**) [43], LLM-based Zero-Shot (**Z.**) [36], and PATCHAGENT (**P.**). 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.

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 2.5: **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.

2.7.3 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 [77]. 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 2.5, 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.

2.7.4 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` [78], a fast binary compressor; (2) `GPAC` [79], a multimedia framework; (3) `libxml2` [80], an XML toolkit library; (4) `wasm3` [81], a WebAssembly interpreter; and (5) `vim` [82], 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, PATCHAGENT, specifically the *gpt-4-0125-preview* version, to repair these vulnerabilities. According to the OpenAI API documentation [83], 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.

CVE/Issue	Project	Bug Type	Fix Date	P.
2024-6064 [84]	gpac	Use After Free	Jun. 13rd	✓
2024-27530 [85]	wasm3	Use After Free	N/A	✓
2024-41965 [86]	vim	Double Free	Aug. 1st	✓
2024-3204 [87]	c-blosc2	Heap Overflow	Apr. 4th	✓
2024-34459 [88]	libxml2	Heap Overflow	May 8th	✓
2024-34249 [89]	wasm3	Heap Overflow	N/A	✓
2024-34252 [90]	wasm3	Global Overflow	N/A	✓
Issue-471 [91]	wasm3	Heap Overflow	N/A	✗
2024-6063 [92]	gpac	Null Dereference	Jun. 12nd	✓
2024-34246 [93]	wasm3	Null Dereference	N/A	✗

Table 2.6: **Summary of unseen vulnerability repair results. P.** 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.

The results are summarized in Table 2.6, demonstrating that PATCHAGENT successfully repaired 8 out of 10 previously unseen vulnerabilities. These findings align with the effectiveness evaluation discussed in Section 2.7.2. 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 Section 2.7.6. 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.

Table 2.7 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

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 2.7: **Github Pull Requests.**

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.

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

Table 2.8: **Token & Money Cost of PATCHAGENT under Different LLM.**

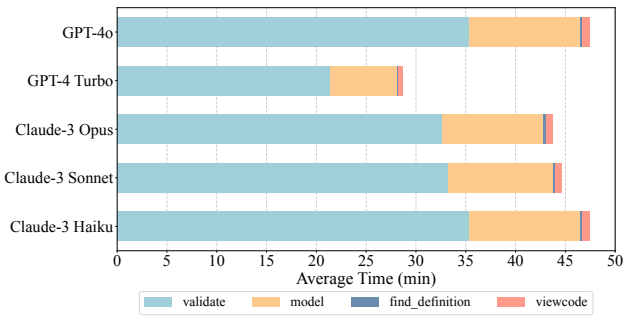

Figure 2.4: **Average Time Cost of PATCHAGENT.**

2.7.5 Efficiency of PATCHAGENT

Token Cost. Figure 2.8 summarizes the average cost of successfully repairing a vulnerability using different models. Claude-3 Opus achieves the highest repair success rate at 84.83%. How-

ever, 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 2.4 presents a detailed breakdown of the average time costs associated with PATCHAGENT when using different models. The time costs are divided into several sub-categories: *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.

2.7.6 Case Studies

In this section, we present three case studies: one case (Issue-2057 [94]) that PATCHAGENT successfully repaired, and two cases (CVE-2024-34246 [93], CVE-2022-1286 [95]) 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 life-cycle 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 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.

```
_onfatal:
    if (result) {
        fprintf (stderr, "Error: %s", result);
        if (runtime)
        {
            M3ErrorInfo info;
            m3_GetErrorInfo (runtime, &info);
            if (strlen(info.message)) {
                fprintf (stderr, " (%s)", info.message);
            }
        }
        fprintf (stderr, "\n");
    }
    m3_FreeRuntime (runtime);
    m3_FreeEnvironment (env);
```

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

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.

```
else {
  if (MRB_METHOD_NOARG_P(m)) {
    check_method_noarg(mrb, ci);
  }
  recv = MRB_METHOD_FUNC(m)(mrb, recv);
}
```

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

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

2.8 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 [96], 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 [97], [98], [99], [100] has proposed various methods to manipulate memory and compromise real-world programs. Moreover, by analyzing critical control information and structures [101], [102], [103], 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 [104]. While this method is effective and scalable for addressing many vulnerability repair scenarios, it has notable limitations. From a security standpoint, prior works [35] 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 [105], [106], [107] 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 [61],

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 [62] 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.

2.9 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 [108], [109] and pointer invalidation [16], [110] have been employed to clear dangling pointers, while one-time allocators [15], [18] have been used to prevent memory reuse. Regarding spatial memory vulnerabilities, prior works [17], [111], [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 [113] 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 [27]. Following this, SWE-Agent [77] and AutoCodeRover [114] 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 [115], [116]. By analyzing the target function, LLMs can identify patterns of vulnerabilities. Code generation [117], [118] is also an important application of LLMs. This task is particularly valuable for modernizing legacy systems and integrating software components written in different languages.

2.10 Conclusion

In this work, we introduced PATCHAGENT, an AVR 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 AVR tools, successfully repairing a significant majority of vulnerabilities with high accuracy.

CHAPTER 3

PORTGPT: TOWARDS AUTOMATED BACKPORTING USING LARGE LANGUAGE MODELS

3.1 Introduction

Large-scale open source projects (*e.g.,* Linux kernel, Node.js, Debian, PostgreSQL, Kubernetes) tend to maintain mainline, stable, and Long-Term Support (LTS) branches to ensure stability, continuous feature delivery, and long-term maintenance [119], [120]. When bugs are discovered in a project, developers tend to fix them in the mainline branch first, after which patch backporting is performed to retrofit a patch to stable and LTS branches. However, patch backporting is complex and labor-intensive [121], [122]. It requires maintainers to manually resolve conflicts and adapt patches to out-of-sync branches, even downstream projects, which can be both time-consuming and error-prone.

Recent years have witnessed several proposals on automated patch backporting, which typically consists of two stages: **localization** (figuring out the right location to apply code changes) and **transformation** (adapting code changes to be compatible with the older version).

Localization techniques have evolved throughout the years from exact surrounding-text matching (*e.g.,* as shown in the `patch` utility), to predicates over nodes in typed abstracted syntax tree (AST) (*e.g.,* FIXMORPH [123]), and further to similarities in semantics-bearing program dependency graphs (PDG) (*e.g.,* TSBPORT [124]). While each advancement allows more "fuzzy" contexts to be matched for hosting backported patches, existing solutions still fall short when code in the mainline and target branch undergoes changes beyond what the rules/heuristics are designed

for. And such changes can be as simple as renaming a function or adjusting the location of a code snippet (shown in Section 3.3) or even declaring a buffer on stack instead of heap (*e.g.,* Section 3.2.5)

Similar to localization, patch transformation has also evolved significantly from text drop-in (*e.g.,* `patch`), to typed-AST based transformation rules (*e.g.,* FIXMORPH) and semantics-oriented predefined transformation operators (*e.g.,* TSBPORT). However, existing works still lack the flexibility to adapt or improvise code modification for older branches, especially when the patch transformation does not fall into the predefined templates (in the case of TSBPORT) or rule synthesis search space (in the case of FIXMORPH), such as accurately aligning symbol names (*e.g.,* function call, struct member, header file) between two branches. In addition, existing approaches are tightly coupled with the syntactic and semantic structures of specific programming languages, which makes it impractical to directly apply these tools to backport patches in other languages.

Nevertheless, while evolving localization and transformation techniques draw inspiration from experience of human experts, when human developers backport a patch, they rarely follow an algorithmic procedure to locate patching points nor do they run an exhaustive search on known patterns to adapt the patch. Instead, human developers could perform a mix of activities that may resemble on the lines of: 1) tracing patching location through version control metadata or symbol locator, 2) comprehending patch context changes, and 3) improvising modification to the patch based on the comprehension, and last but not least, 4) trial-and-error. Naturally, one direction of improving backporting tools is to make the tool behave more like human experts.

The recent rise of large-language models (LLMs) shed lights on how we can equip a backporting tool with the capability of code comprehension and code generation. In fact, Mystique [125] and PPathF [126] has already shown that an off-the-shelf LLM can transform a patch (denoted as function before and after the patch $f_o \rightarrow f_n$) to a new context (f'_o) via in-context learning [59],

[127] only given common instructions (I):

$$\text{Query}(I, f_o \to f_n, f'_o \to ?), \text{where } ? \text{ is the LLM response}$$

However, they are not end-to-end backporting systems yet as it still requires developers to manually designate the hosting function (f'_o) for the ported patch. More importantly, they completely forgo rich information (e.g., Git commit history, test cases, etc) that can and will be leveraged in manual backporting. Evidently, they are unable to reflect on or revise incorrect patches.

In this work, we propose PORTGPT, an end-to-end LLM-based patch backporting tool. We intentionally design PORTGPT by shadowing how human developers perform backporting. By atomizing, analyzing, and summarizing key actions taken by developers as well as the information derived from the actions, we note that developers employ a variety of capabilities, such as aggregating Git diffs and counterexample-guided refinement (see Section 3.3) in backporting. This implies an agentic architecture—which empowers an LLM more tools and freedom of reasoning—might be more suitable for backporting tasks than in-context learning.

Therefore, the key design philosophy of PORTGPT is to provide tools and chain-of-thoughts that we believe to be useful (based on manual backporting experience) to an LLM agent to facilitate its reasoning. In terms of **localization**, we provide tools for an LLM agent to locate symbols, selectively access relevant source code, and Git history. These tools help the LLM agent focus on the pertinent code and better track code changes across branches. For **transformation**, PORTGPT allows LLM to reason and improvise based on its knowledge base and the concrete backporting task, but also provide best-effort validation of the generated patches and feedback on why a patch cannot be compiled or applied (*e.g.,* missing header file or test case failure).

We built PORTGPT upon GPT-4o [128] and evaluated PORTGPT on large-scale datasets from

prior works, including 1465 Linux kernel CVEs from TSBPORT and 350 Linux kernel bugs from FIXMORPH. PORTGPT outperformed both FIXMORPH and TSBPORT with an overall success rate of 89.15%. Additionally, we created a more complex and diverse dataset consisting of 146 cases sourced from 34 programs across three popular languages (C, C++, and Go). On this more challenging dataset, PORTGPT successfully backported 62.33% of the patches, significantly outperforming FIXMORPH and TSBPORT (by 56.53% and 26.09%, respectively).

To evaluate PORTGPT's applicability in real-world, we selected patches for Linux LTS and Ubuntu introduced after the knowledge cutoff (October 2023) of GPT-4o. On the Linux 6.1-stable branch, PORTGPT successfully backported 9 patches out of 18, all of which were thoroughly verified and subsequently accepted by the Linux community. For Ubuntu, we tested 16 patch pairs corresponding to 10 CVEs across multiple versions, and PORTGPT successfully backported 10 of them.

Summary This work makes the following contribution:

• We propose a practical LLM-based backporting framework that shadows manual patch backporting workflow and leverages commonly used developer tools, such as `git`, to automatically backport patches to target versions.

• We conduct a thorough evaluation of PORTGPT using 1,961 patches, demonstrating that PORTGPT achieves superior performance compared to previous works while maintaining comparable efficiency.

• We leverage PORTGPT to handle real-world patches that are difficult to backport. 9 of these patches are accepted by the Linux kernel community, demonstrating its practical applicability.

We have open-source our dataset at https://github.com/OS3Lab/patch_dataset and PORTGPT at https://github.com/OS3Lab/patch-backporting.

3.2 Why LLMs for Backporting?

Although LLMs have demonstrated exceptional code processing capabilities in a diverse set of scenarios [106], [129], [130], [131], the fundamental reason why LLMs should be considered in backporting is beyond that. In this paper, we argue that the inherent context matching and code transformation schemes in LLMs, while a blackbox in terms of explainability, have the potential to outperform even the most comprehensive set of syntactic or semantic rules or heuristics defined in prior backporting works [123], [124], especially with the right prompting tactics inspired by how human experts solve backporting tasks.

This section presents a series of crafted backporting cases to illustrate how solutions to the two key challenges in backporting—locating matching context and transforming code patch—have evolved from text-based to syntax-based, semantics-based, and finally why LLMs as generative models have a unique advantage over prior rule-based designs.

3.2.1 Problem Description

Backporting, denoted as $B : P_n \rightarrow P_o$, is a program repair technique that adapts a patch ΔP, originally designed for a newer software version P_n, to an older version P_o. This approach is essential for preserving the security and functionality of legacy systems that cannot be fully updated due to constraints such as compatibility issues or custom configurations. Unlike conventional program repair practices that rely on proof-of-concept (PoC) exploits [43] or static analysis reports [39] to produce the patch ΔP, backporting has access to ΔP already from the beginning. Instead, the core challenges of backporting lies in two aspects:

1) Given ΔP is localized on code context L_n in P_n, we need to accurately locate L_o in P_o that matches L_n and is suitable to host the backported patch.

2) We need to identify a transformation $T(\Delta P, L_o, L_n)$ such that the resulting backported patch integrates seamlessly into L_o, accounting for the differences between L_n and L_o, eliminating the vulnerability addressed by ΔP, and preserving functionalities of P_o.

```
1 void process_input(char *input) {
2   char buffer[64];
3   strcpy(buffer, input);
4   printf("Truncated input: %s\n", buffer);
5 }
```

Listing 5: Code snippet in P_n with a buffer-overflow

```
1 @@ -1,5 +1,6 @@
2  void process_input(char *input) {
3    char buffer[64];
4 -  strcpy(buffer, input);
5 +  strncpy(buffer, input, 63);
6 +  buffer[63] = '\0';
7    printf("Truncated input: %s\n", buffer);
8  }
```

Listing 6: Patch ΔP to fix the bug in P_n (see Listing 5)

Stage Setting: suppose we have a simple buffer-overflow vulnerability around code context L_n (Listing 5) and the bug is recently patched with ΔP shown in Listing 6. In the rest of this section, we present how backporting tools have evolved in terms of locating L_o and transforming ΔP.

3.2.2 Vanilla Text-based Backporting

Backporting is almost trivial when there are no (or minimal) changes in the code context where ΔP is applied onto (i.e., L_n). For example, in an older version P_o where the `process_input` function resides in the same file spanning over (almost) the same lines with identical function body shown in Listing 5. In this case, $L_o \approx L_n$ and the transformation of ΔP can be handled by the GNU `patch` utility or `git cherry-pick` gracefully.

3.2.3 Syntax-based Backporting

The vanilla text-based backporting will stop working both in locating L_o and transforming ΔP when there are even simple syntatic changes, as shown in Listing 7.

In this backporting task, the function was named `handle_input` and the buffer was named `buf` in P_o and somewhere along the development from P_o to P_n, both symbols are renamed (but the vulnerability remains). GNU `patch` errors as it cannot find a matching context for ΔP.

```
1 void handle_input(char *input) {
2   char buf[64];
3   strcpy(buf, input);
4   printf("Truncated input: %s\n", buf);
5 }
```

Listing 7: P_o with the same bug shown in Listing 5.

Syntax-based backporting, piloted by FIXMORPH [123], attempts to solve cases like this by identifying matching contexts and transforming code patches at the level of ASTs. More specifically, locating matching context L_o is essentially checking whether there exists an AST snippet in P_o that matches with L_n, generalized by heuristic rules such as symbol names, type signatures, or even control-flow structures. In the backporting task of Listing 7, FIXMORPH is able to conclude that `handle_input` is the matching context (L_o) for ΔP in Listing 6 due to AST-based fuzzy match. Subsequently, FIXMORPH derives the transformation rule, also expressed on the AST level, and apply the transformation to L_o, creating a backported patch in Listing 8. Note that in the adapted patch, all appearances of `buffer` in ΔP are substituted with `buf` as on both ASTs, var-use sites refer to the actual variable object, not its symbol.

```
@@ -1,5 +1,6 @@
 void handle_input(char *input) {
   char buf[64];
-  strcpy(buf, input);
+  strncpy(buf, input, 63);
+  buf[63] = '\0';
   printf("Truncated input: %s\n", buf);
 }
```

Listing 8: Backported patch for P_o in Listing 7.

3.2.4 Semantics-based Backporting

While syntax-based backporting is powerful especially when the context matching and patch transformation rules can be expressed with typed ASTs, it can fail when the patch requires semantics (either about the program or the vulnerability it fixes) to backport. Listing 9 is an example when backporting requires the semantic knowledge of the patched vulnerability (buffer overflow). While syntax-based tools (e.g., FIXMORPH) can reconcile symbol changes and produce a patch, the patch will be the same as Listing 8 and this is a wrong patch as now the `buf` is only 32 bytes while both `strncpy(buf,input,63)` and `buf[63]='\0'` in the patch overflow it.

```
void handle_input(char *input) {
  char buf[32]; // NOTE: length reduced
  strcpy(buf, input);
  printf("Truncated input: %s\n", buf);
}
```

Listing 9: P_o with the same bug shown in Listing 5.

To retrofit semantic information in backported patch, two generic approaches have been proposed in prior works:

Templates TSBPORT [124] categorizes patches into predefined templates such as adding sanity checks, modifying function call arguments, honoring def-use relations, and use these templates to guide patch transformation. The process involves inferring the intention of each hunk in ΔP, and

transforming this hunk to L_o preserving the intention. In the case of Listing 6, TSBPORT infers the intention of the only hunk in ΔP is to use safer string functions and null-terminate a buffer, and subsequently transforms the patch to Listing 10 which applies to Listing 9 correctly.

Constraint-solving PATCHWEAVE [132] retrofit semantics into backporting via concolic execution. More specifically, it summarizes the semantic effect of ΔP (preventing a buffer overflow) and ensure that the backported patch achieves the same effect. However, PATCHWEAVE requires a PoC input in order to collect symbolic constraints around the patch, which is not always readily available. And symbolic execution inherently struggle to scale for large codebases.

```
@@ -1,5 +1,6 @@
 void handle_input(char *input) {
   char buf[32]; // NOTE: length changed
-  strcpy(buf, input);
+  strncpy(buf, input, 31);
+  buf[31] = '\0';
   printf("Truncated input: %s\n", buf);
 }
```

Listing 10: Backported patch for P_o in Listing 9.

3.2.5 LLM-based Backporting

While effective, semantics-based backporting, especially those relying on heuristically defined templates, can still fail when the tool cannot infer the intention of the patch (or a hunk in the patch), or cannot transform a hunk as the changes in the context (L_o) is not captured by any of its templates. Listing 11, it is only a small tweak to Listing 9 by allocating the `buf` on heap instead of stack, but TSBPORT cannot produce a correct patch as inferring the size of `malloc`-ed object is not a template in TSBPORT. However, backporting ΔP via any modern LLM (e.g., ChatGPT 4o) is simple even with the most obvious prompt, as shown in the shared conversation. Based on this

experience, we believe LLM-based backporting can be a catch-all solution, especially for changes beyond what is describable by heuristics and rules in syntax- and semantics-based backporting practices. This stance, to the best of our knowledge, is not highlighted in prior works yet.

```
1 void handle_input(char *input) {
2   char *buf = malloc(32);
3   if (buf == NULL) { return; }
4   strcpy(buf, input);
5   printf("Truncated input: %s\n", buf);
6   free(buf);
7 }
```

Listing 11: P_o with the same bug shown in Listing 5.

3.2.6 Backporting Types

In this work, we adopt the same categorization for backporting tasks established in prior studies [123], [124]. Based on how significantly it differs from the original patch, a backported patch is typically classified as:

• **Type-I** *(No changes)*: The backported patch is identical to the original patch both in code and location, meaning it can be cherry-picked trivially without any modifications.

• **Type-II** *(Only location changes)*: The backported patch requires no transformation, it differs from the original patch only in where it is applied (*e.g.,* line numbers or file names).

• **Type-III** *(Syntatic changes)*: The backported patch requires syntatic modifications only (*e.g.,* function and variable names) to ensure compatibility with the target version.

• **Type-IV** *(Logical and structural changes)*: More intrusive modifications are applied (*e.g.,* adding or removing lines in the patch), making the backported patch syntactically different while preserving the same functionality.

```
--- a/net/netfilter/nf_tables_api.c
+++ b/net/netfilter/nf_tables_api.c
@@ -2873,27 +2873,31 @@ *nft_expr_init(
  err = nf_tables_expr_parse(ctx, nla, ...);
  if (err < 0) goto err1;

+ err = -EOPNOTSUPP;
+ if (!(expr_info.ops...flags & NFT_EXPR_STATE))
+   goto err_expr_stateful;

  err = -ENOMEM;
  expr = kzalloc(expr_info.ops->size, ...);
@@ -5413,9 +5417,6 @@ *nft_set_elem_expr_alloc(
    return expr;

  err = -EOPNOTSUPP;
- if(!(expr->ops->type->flags & NFT_EXPR_STATE))
-   goto err_set_elem_expr;
-
  if (expr->ops->type->flags & NFT_EXPR_GC) {
    if (set->flags & NFT_SET_TIMEOUT)
      goto err_set_elem_expr;
```

Listing 12: Original Patch for CVE-2022-32250

3.3 Prompting (In-context Learning) Is Not All

While Section 3.2 highlights the inherent advantages of LLMs over traditional methods for patch backporting, a critical question remains: how can we best leverage and enhance their code understanding capabilities to navigate the complexities of diverse backporting scenarios? Indeed, a natural approach is to enable LLMs to emulate the processes employed by human developers during backporting. In this section, we present an example illustrating how human developers perform patch backporting. This demonstrates how we should equip LLM with a similar set of capabilities to emulate human developers effectively.

3.3.1 Motivating Example

CVE-2022-32250 Listing 12 presents the patch in the original version for a use-after-free (UAF) bug that caused CVE-2022-32250 [133]. To address this issue, the core logic of the patch involves relocating the check statements from `nft_set_elem_expr_alloc` to an earlier stage in the process, specifically within `nft_expr_init`. However, this patch cannot be directly applied to *5.4-stable*, and a call-for-volunteer is even advertised on the mailing list [134].

Backport by human Human experts typically start with the `patch` utility first, and then handle failed hunks on a hunk-by-hunk basis. Unfortunately, in this example, neither of the two hunks can be applied directly due to context conflict, *i.e.,* the code context that the patch involves has differed between the original version and the target version.

❶ To **locate** the first hunk in the target version, the expert begins by finding the definition of function `nft_expr_init`. This localization is identified by its semantic similarity to the patch; in this instance, a key `goto` statement, unique to the patch's logic, pinpoints the location. ❷ Observing that the patch's surrounding context had evolved between versions, to **transform** the hunk, the expert then consulted git history to understand the reasons for these changes before attempting conflict resolution. ❸ Subsequently, the expert examines the change history of the code snippet through `git` and resolves the namespace conflicts that `expr_info` should be replaced with `info` in the target version. ❹ with that, the expert completes the first hunk of the patch.

❺ To **locate** the second hunk in the target version, the expert first attempts to find function `nft_set_elem_expr_alloc` by its symbol but fails as the function name does not exist in the old version. ❻ Then, the expert traces the origin of this function in the original version with `git` and finds that it was introduced by commit *a7fc93680408*. ❼ Next, the expert uses `git show` to examine the details of this commit and concludes that `nft_set_elem_expr_alloc`

was migrated from `nft_dynset_init` in another file. ❽ Therefore, the expert views function `nft_dynset_init` in the old version instead and makes sure it is the correct host for backported patch. ❾ Finally, referencing the git history, the expert infers that `priv->expr` in the target version is equivalent to `expr` in the patch, and consequently deletes the corresponding statements.

❿ After completing the patch backporting for the target version, the expert runs tests to verify its correctness (as in no regression) and effectiveness (as in defending against the PoC exploit, if available). Additionally, the expert refines the patch by addressing potential issues identified during compilation, testsuite evaluations, and PoC testing.

3.3.2 Observations

Based on the manual patch backporting process presented above, we draw some key observations that influence the design of PORTGPT—our LLM-based backporting tool.

1) It is impossible to fit all information possibly needed by end-to-end backporting (*e.g.,* the entire commit history) into a single or even a pre-defined sequence of prompts.

2) Information relevant to the current step can be retrieved frequently on an as-needed basis, but it still needs to be presented in a concise way in order not to overload the context limit of a human brain.

3) Decisions are frequently taken based on results of previous steps in a trial-and-error manner, *e.g.,* tracing the provenance of a symbol when symbol resolution fails while viewing function body of lookup is successful.

4) Backported patch doesn't have to be generated in one-take. Validating and subsequently improving the patch based on feedback is an inseparable step in the manual process.

Based these observations, we believe an agentic design is well-suited for LLM-based backporting, and identify three key capabilities to retrofit a certain degree of human expertise, common

workflows, and best-practices into the agent.

Basic code viewing and symbol lookup: It is essential to equip LLM with basic code access capabilities, as what human expert does in steps ❶❷❺❽. By utilizing symbol locating and code viewing, LLM can access the code context of the target version.

Access aggregated diff and track provenance : By aggregating historical commits of the patch code, we can precisely trace the location of the code change in the target version, as shown in steps ❸❻❼. Whether it involves code relocation or identifying code segments absent in the target version, analyzing the modification history allows for precise determination. Moreover, the history of the changes clearly reveals the changes of the namespace present in the context.

Counterexample-guided refinement: Since the patch from the original version may include dependencies (*e.g.,* header file imports) unavailable in the old version, basic compilation tests are essential. If there are compilation errors, the previously generated patch can serve as a counterexample, using compiler error messages to guide the refinement of the patch. This step is also a practice typically performed by experts after developing the patch, as shown in step ❿.

3.4 Design Details of PORTGPT

In order to empower LLM agents with the capabilities presented in Section 3.3, we design a series of tools and workflows and further implement several optimizations to standardize and improve the formatting of LLM outputs. In this section, we describe these design details of PORTGPT. We start with an overview of PORTGPT (Section 3.4.1), followed by detailed description of two main stages of PORTGPT: Per-Hunk Adaptation (Section 3.4.2) and Final Patch Combination (Section 3.4.3).

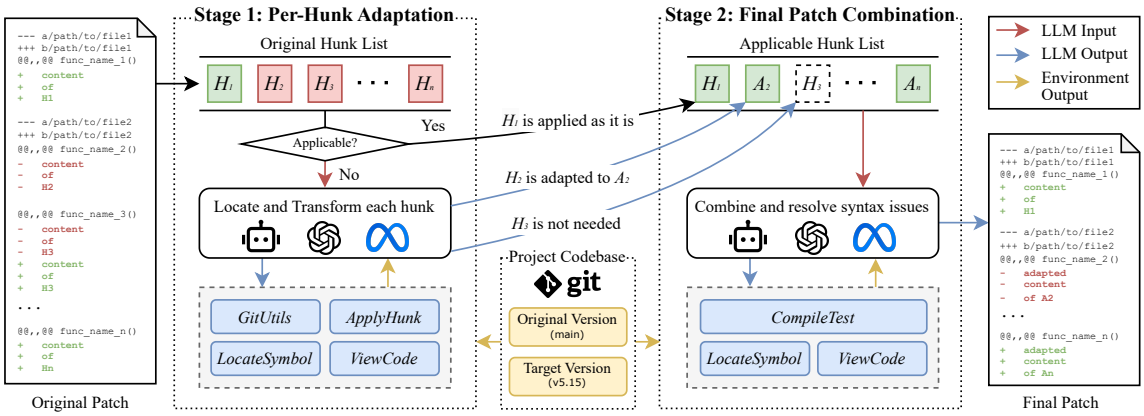

Figure 3.1: **Workflow of PORTGPT**

3.4.1 Overview

As illustrated in Figure 3.1, the overall workflow of PORTGPT consists of two main stages: Per-Hunk Adaptation and Final Patch Combination. In the first stage, PORTGPT completes **localization** and initial **transformation** for each hunk individually. PORTGPT extracts hunks from the original patch and per each hunk, PORTGPT first determines whether the hunk requires backporting, and if so, transforms the hunk to ensure compatibility with the target version. This stage aims to ensure that the generated patch can be successfully applied to the target version. In the second stage, PORTGPT combines the transformed hunks, applies the entire patch to the target version, and sends the backported codebase for compilation. If compilation failed, PORTGPT attempts to resolve them by adding necessary definitions or adjusting the code context to finalize the **transformation**. Ultimately, PORTGPT outputs backported patches that are free from compilation errors. Both stages leverage an LLM agent, supplemented with customized tools designed to enhance the LLM's performance.

3.4.2 Per-Hunk Adaptation

Backporting a hunk to the target version takes two steps. First, it is necessary to determine if the hunk should be backported and to identify the appropriate matching context. Second, the hunk must be transformed to match the host context in the target version. To complete these steps, PORTGPT must gather sufficient information from the codebase to understand how the codebase has evolved across different branches and how the current hunk relates to the surrounding code. To support this process, we have designed three tools called *ViewCode*, *LocateSymbol*, and *GitUtils* to facilitate information retrieval. Once a hunk is transformed, PORTGPT uses another tool called *ApplyHunk* to apply the adapted hunks to the target version.

• ***ViewCode.*** This tool takes four parameters: *ref*, *file*, *start*, and *end*, which specify the commit, file path, and the line range to be retrieved. It returns the code snippet between the specified line numbers in the given version of the codebase, enabling the LLM to access any portion of the source code from any Git-tracked version.

• ***LocateSymbol.*** This tool takes two parameters: *ref* (the specific commit) and *symbol* (language objects, such as function name, variable name). It returns the definition site of the symbol in the specified version, including the file path and line number, allowing the LLM to accurately locate symbols within the codebase.

• ***GitUtils.*** This tool consists of two components: *History* and *Trace*, neither of them directly takes any parameters from the LLM. The *History* component presents the evolution of code snippets across commits or versions, while *Trace* provides detailed insights into the trace of code movement. They are designed to provide information on code evolution and commit history to the LLM. The detailed design will be discussed later.

• ***ApplyHunk.*** This tool accepts one parameter: *patch* and allows LLM to apply the patch to the target version. To mitigate potential errors in LLM-generated patches, it automatically corrects

patch formatting issues and generates diagnostic feedback for the LLM if the patch failed to apply. Further details are provided below.

***GitUtils* Detail** The *History* component displays a list of Git commits that modify the code snippet within the current hunk from the *fork point* to the original version. The *fork point* refers to the divergence between the original and target versions, *i.e.,* the most recent common ancestor. For example, v6.1 tag is *fork point* of the mainline branch and the linux-6.1.y stable branch. The tool analyzes the file modification history and identifies all changes made from the *fork point* to the original version. It further filters the history to include only changes affecting the lines within the hunk. This design ensures that the output is both comprehensive and relevant. By focusing on commits from the *fork point* onward and isolating the changes to the lines in the hunk, *History* provides a detailed evolution of the code while excluding unrelated modifications.

While the component *History* reveals local changes in the corresponding code snippet of a hunk, it may overlook critical global changes, such as: the relocation of code. These global changes, though important, are challenging to incorporate directly into *History* because they are often too lengthy for an LLM.

To address this limitation, we introduce *Trace* to extract and highlight significant global changes, particularly the relocation of patch contexts. We observe that code movement changes have a direct impact on backporting. Code movement refers to the relocation of a code snippet from one location to another after the *fork point*, resulting in discrepancies in its placement between the original and target versions. To produce an accurate backported patch, LLM must identify the code movement and its target location in a patch. Leveraging the minimal edit distance method [71], we compare the added code with the deleted code to facilitate LLM in the code movement identification. Edit distance is an effective metric—it captures syntactic similarity and is applicable across different

programming languages. Finally, with the location of the matched code and its associated commit message provided by *Trace*, LLM determines if the code is migrated from that location or the code is newly introduced.

Algorithm 1: How *ApplyHunk* corrects a patch and generates feedback for different failure reasons.

Input: A patch to be corrected and applied; A flag indicates whether to activate the Automated Context Correction mechanism.
Output: Patch apply feedback;

```
1  Function apply_feedback(patch, flag):
2      output = str();
3      possible_files = list();
4      patch = format_patch(patch); // avoid "invalid patch"
5      if flag then
6          patch = automated_context_correction(patch);
7      result = git_apply(patch);
8      if "File not found" in result then
9          old_file = find_rename_file(patch);
10         if old_file then
11             return old_file;
12         possible_files.append(locate_symbol(patch));
13         possible_files.append(find_similar_file(patch));
14         for path in possible_files do
15             patch = revise_patch(patch, path);
16             feedback = apply_feedback(patch);
17             if "success" in feedback then
18                 return path;
19             output = output + path + feedback;
20         return output;
21     if "Error while searching for context" in result then
22         C_llm = extract_context(patch);
23         C_tgt = find_corresponding_context(C_llm);
24         return compare(C_llm, C_tgt);
25     return "success";
```

***ApplyHunk* Detail** This tool systematically handles errors that may arise during `git apply`. For each type of error, we either follow a customized correction scheme or generate corresponding feedback to guide the LLM for an iterative refinement of the patch (*i.e.,* the transformed hunk) as

detailed in Algorithm 1.

Corrupted patch. This error arises when the patch does not adhere to the correct format [135], *i.e.,* LLM forgets to add a space at the beginning of the code line. Therefore, the tool avoid such error by checking each code line of the patch and adding required spaces.

Non-existent file path. This error occurs when the patch is applied to a file that does not exist, often due to the file being renamed or the relevant function residing in a different file in the original version. To address this, the tool tries to identify possible file path lists. Initially, the tool checks if the file has been renamed. If not, the tool locates possible files in the target version by either matching the original version file name or locating one symbol referenced in the patch. Finally, the tool applies the patch to all identified possible file paths and returns apply feedback to LLM.

Context mismatch. Patch context refers to lines that start with ' ' (space) or '-' in the patch. This error occurs when the context of the LLM-generated patch (C_l) is inconsistent with the target code. To resolve this, this tool identifies the most similar code block with the minimal edit distance in the target version as C_t. Then, it compares C_l with C_t line by line and highlights any discrepancies for the LLM, allowing it to make the necessary adjustments. Furthermore, we propose an **Automated Context Correction** mechanism, which forcibly substitutes lines in C_l that differ from C_t before applying the patch. Once the correction is made, the patch can be successfully applied. To preserve the LLM's original intent as much as possible, the tool only activates automated context correction after three failed attempts to apply the patch due to context mismatches.

Patch Localization Locating a matching context in the target version to host an adapted hunk includes two subtasks: identifying the correct target file path and locating the corresponding code block within that file. We find possible locations by applying the original hunk with *ApplyHunk* and provide possible locations in tool's output to LLM. LLM could adopt two distinct approaches

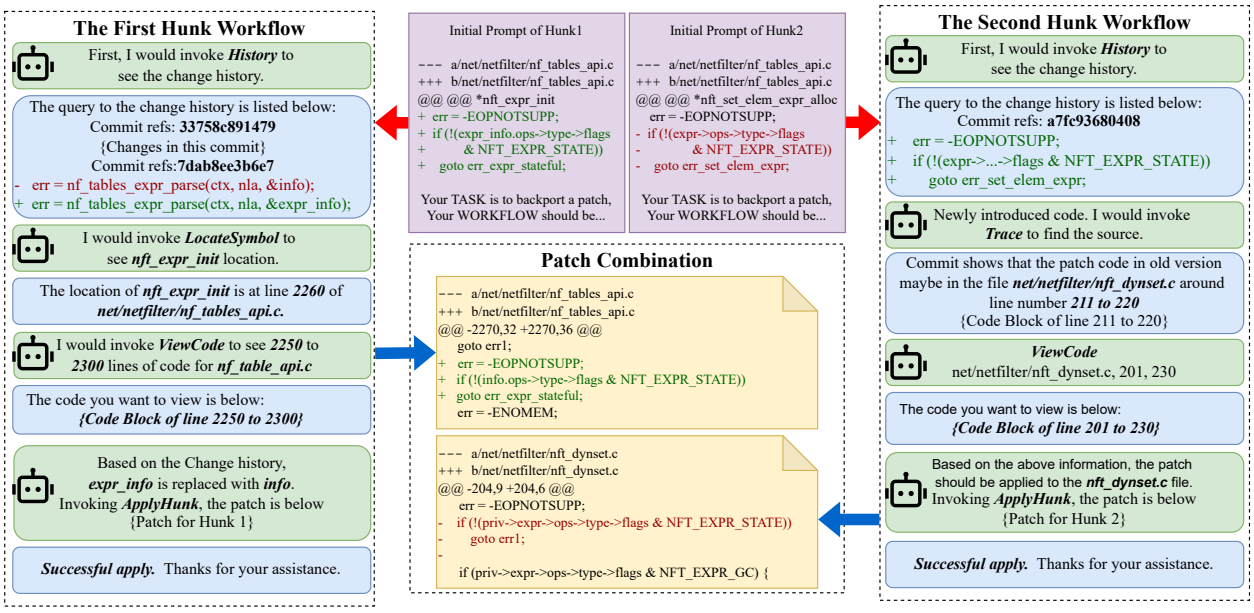

Figure 3.2: **Example of Per-Hunk Adaptation.** The red arrows represent LLM agent inputs, while the blue represent LLM agent outputs. The green box represents the invocations of LLM, while the blue box represents the outputs of tools.

for using *GitUtils* to narrow down to the correct location.

Symbol-Based Localization: This workflow is particularly effective when the patch does not involve newly introduced code or complex code movement. In this approach, the LLM first invokes *History* to retrieve the code change history and determine whether the original symbol name has been renamed in the target version. If renamed, it then uses *LocateSymbol* to identify the location of the renamed symbol within the target codebase. Cooperated with *ViewCode*, LLM further checks code blocks around those symbols and determine where the patch should be backported to.

Movement-Aware Localization: In more complex scenarios, such as a code snippet has been moved to several different locations, *LocateSymbol* may not be sufficient to identify the required symbols. In such cases, the LLM invokes *Trace* to determine the specific location of the code in the target version or to confirm if the code no longer exists. This ensures accurate localization and facilitates modifications even in challenging situations.

CVE-2022-32250 To better understand the operations in the first stage, we continue with the patch of CVE-2022-32250 (shown in Listing 12). The overview of this process is provided in Figure 3.2.

Regarding the first hunk, the LLM sequentially performs the following steps: ❶ Invokes *GitUtils*. From *History*, the LLM observes, `expr_info` has been replaced with `info` in the old version. ❷ Invokes *LocateSymbol* to determine the location of `nft_expr_init` (modified by the hunk) in the old version. ❸ Invokes *ViewCode* to inspect the code context surrounding the identified location. ❹ Generates the transformed patch based on the gathered information and then invokes *ApplyHunk* to validate its correctness. For the second hunk, it involves more complex scenarios. The LLM first ❶ invokes *History*, which shows that the current code context was introduced in the *a7fc93680408* commit. This satisfies the conditions for calling *Trace*. Consequently, the LLM proceeds to ❷ invoke *Trace*. Through minimal edit distance similarity matching, *Trace* identifies that this code originates from Line 211 to 220 in `net/netfilter/nft_dynset.c`. With precise code location information, the LLM then follows the same workflow and invokes ❸ *ViewCode* and ❹ *ApplyHunk* to complete the patch transformation. After transforming both hunks, PORTGPT generates the combined patch.

3.4.3 Final Patch Combination

After the inital patch is generated from previous stage, PORTGPT will combine all the generated patches of each hunk and check for compilation errors. If there are any errors, PORTGPT will resolve them by adding necessary definitions or adjusting the code context, which is achieved by an LLM equipped with a series of customized tools which included *ViewCode* and *LocateSymbol* we mentioned before with a new tool *CompileTest*. Same as tool *ApplyHunk*, *CompileTest* takes a draft patch as input, the tool will apply the patch to the target version and compile the code. If the

```
--- a/fs/cifs/cifs_debug.c
+++ b/fs/cifs/cifs_debug.c
@@ -332,6 +332,11 @@
 struct cifs_ses *ses;
 list_for_each(...) {
+   spin_lock(&ses->ses_lock);
+        if (ses->ses_status == CifsExiting) {
+                spin_unlock(&ses->ses_lock);
+                continue;
+        }
        if ((ses->serverDomain == NULL) ||
```

(a) Patch Generated by Stage-1

```
--- a/fs/cifs/cifs_debug.c
+++ b/fs/cifs/cifs_debug.c
@@ -332,6 +332,11 @@
 struct cifs_ses *ses;
 list_for_each(...) {
+   spin_lock(&GlobalMid_Lock);
+        if (ses->status == CifsExiting) {
+                spin_unlock(&GlobalMid_Lock);
+                continue;
+        }
        if ((ses->serverDomain == NULL) ||
```

(b) Patch Generated by Stage-2

Figure 3.3: Patches Generated for CVE-2023-52752.

code can be compiled successfully, the tool will output the final patch and terminate the second stage. Otherwise, the tool will return the error message. Typically, the compilation error messages are long and complex, which may degrade the performance of the LLM [136]. To address this issue, we design a customized error message parser to extract the key information from the error message. The parser is based on a set of regular expressions that are designed to capture the most common error patterns. The parser extracts the error type, the file name, the line number, and the error message. The extracted information is then used to guide the LLM in resolving the compilation errors. erview of Our Dat

To better understand how the second stage works, we provide an example. Figure 3.3a presents

the patch for CVE-2023-52752 [137] generated by Stage-1. However, the `cifs_ses` structure (Line 4) does not have members `ses_lock` and `ses_status`, leading to a compilation error. PORTGPT will issue an error message to the LLM: "`fs/cifs/cifs_debug.c` at Line 340, `cifs_ses` has no member `ses_lock`" (`ses_status` same), requesting a fix. Based on this information, the LLM will follow these steps: ❶ First, it invokes *LocateSymbol* to find the definition of `cifs_ses`, with *LocateSymbol* outputting that the symbol is defined in `fs/cifs/cifsglob.h` at Line 968. ❷ Then, it invokes *ViewCode* to inspect the definition of `cifs_ses`. In the definition, `ses_status` corresponds to a similar field `status`. The comment also indicates that `status` is protected by `GlobalMid_Lock`. ❸ Based on the content from *ViewCode*, the LLM deduces that `ses_status` in the target version should be replaced by `status`. The semantics indicate that `ses_status` is protected by `ses_lock`, so `status` should be protected by `GlobalMid_Lock` accordingly. The LLM will then generate the revised patch in Figure 3.3b and invoke the validation tools to verify its correctness.

3.4.4 Prompt Design

To enable the LLM to perform the backporting task more effectively, we carefully craft the prompt for each stage. Agents in the two stages share the same system prompt, which explains the concept of patch backporting, the provided tools and their usage. This ensures that the LLM understands the context of patch backporting and the resources it can access. The user prompts differ between two stages since they are different tasks (shown in Figure 3.4). Notably, both user prompts consist of two key components: the task, which specifies what the LLM needs to accomplish, and the workflow, which provides detailed guidance on how to execute the task effectively.

In the first stage, the objective is to generate a patch that can be successfully applied hunk by hunk. The user prompt specifies the patch to be backported, the original version, and he target

Stage-1 Prompt

Your **TASK** is to backport a patch fixing a vuln from a original version of the software to an target version step by step. The project is {project_url}. For the ref {original_patch_parent}, the patch below is merged to fix a security issue:
{original_patch}
I want to backport it to ref {target_patch_parent}. To assist you in reviewing the relevant code, I have provided the following code blocks from the target version that closely match the current patch location:
{similar_block}
Your **WORKFLOW** should be: .....

Stage-2 Prompt

Your **TASK** is to validate and revise the patch until it is successfully backported to the target version and really fixes the vulnerability.
Below is the patch you need to backport:
{new_patch}
According to the patch above, I have formed a patch that can be applied to the target version:
{complete_patch}
Now, I have tried to compiled the patched code, the result is: {compile_ret}
You can validate the patch with provided tool *validate*. There are some processes to validate if the patch can fix the vulnerability: .....
If the patch can not pass above validation, you need to revise the patch with the help of provided tools. The patch revision **WORKFLOW** should be: .....

Figure 3.4: **User Prompt of PortGPT**

version and provides a clear workflow to guide the LLM. Specifically, LLM firstly locates where to apply the code changes in the target version by: ❶ utilizing similar code blocks we provided in the prompt, ❷ invoking *GitUtils* to to analyze code change history, ❸ invoking *LocateSymbol* to identify the locations of functions or variables. The user prompt also instructs the LLM to transform a hunk to align with the target version's context, leveraging tools such as *ViewCode* to access and analyze the target version's code.

In the second stage, the objective is to ensure that the complete patch is reliable and does not introduce new risks. The LLM is prompted to iteratively refine the patch based on validation feed-

back, which may include compilation error messages produced by the *CompileTest* tool. In summary, the prompts are structured to empower an LLM for backporting tasks by ensuring it knows how to use the tools to gather as much necessary information as possible while also following a clear and systematic workflow.

3.5 Implementation

PORTGPT uses LangChain [72] to facilitate the integration of LLM agents and prompt engineering, and existing tools for symbol resolution and patch validation.

Locate Symbol For symbol resolution, we use **ctags** [138] as the syntax parser. As a tool widely used in text editors for fast navigation, ctags supports syntax parsing for dozens of programming languages, such as C, C++, Go. In PORTGPT, by parsing the output of ctags, we could quickly generate an available symbol table for *LocateSymbol*. Specifically, ctags parses symbols such as functions, structs, and variables that appear in the codebase. It then outputs the symbol's location information in the format of "`symbol_name file lineno symbol_type`". By reading the output in the ctags format, we can generate a symbol table with symbols and their corresponding locations.

Context Matching We treat lines prefixed with '-' or space in the generated patch as patch context. These are extracted and compared against all equally sized code blocks in the target file using the edit distance. The block with the minimal distance is selected, leveraging the fact that original and target code usually retain strong textual similarity despite version differences.

Validation Chain To ensure that the LLM-generated patch does not introduce new security risks when both the PoC and functionality test suites are available, we design a rigorous, multi-stage val-

idation chain following the second stage. First, we execute the original PoC on the patched version to confirm that the vulnerability has been successfully eliminated. And then, we run comprehensive functionality test suites to verify that the patch preserves the intended behavior and does not compromise the program's correctness. Finally, we would additionally validate a backported patch if a directed fuzzer suitable for the underlying project can be used. This layered validation strategy significantly enhances the reliability and safety of the generated patches.

3.6 Evaluation

In this section, we assess PORTGPT to address the following research questions:

• **RQ1 (Performance)**: How does PORTGPT compare to other works in terms of the success rate of backporting?

• **RQ2 (Ablation)**: How does each module in PORTGPT contribute to its overall performance?

• **RQ3 (Efficiency)**: How well PORTGPT handles the backporting task in terms of time and cost efficiency?

• **RQ4 (Practicality)**: How applicable is PORTGPT for handling backporting tasks in real-world scenarios?

3.6.1 Settings

Environment All experiments were conducted on a 64-bit Ubuntu 24.04 LTS server equipped with an Intel(R) Xeon(R) Gold 6248R 96-core CPU running at 3.00 GHz, 512 GB of memory, and a 22 TB hard drive. The large language models used in our evaluation include GPT-5 [139] (version *gpt-5*), GPT-4o [128] (version *gpt-4o-2024-08-06*), Gemini-2.5-Flash [140] (version *gemini-2.5-flash*), Llama 3 [141] (version *Llama 3.3 70B*) and DeepSeek-v3 [142] (version *DeepSeek-V3-0324*).

C					C++					Go				
Project	**II**	**III**	**IV**	**Tot.**	**Project**	**II**	**III**	**IV**	**Tot.**	**Project**	**II**	**III**	**IV**	**Tot.**
Linux	6	4	11	21	MongoDB	0	3	7	10	golang	6	2	7	15
FFmpeg	0	5	7	12	Chromium	0	2	5	7	Argo CD	3	1	7	11
redis	0	6	4	10	envoy	0	4	1	5	consul	3	1	2	6
libtiff	0	3	3	6	electron	0	1	1	2	containerd	0	0	4	4
cpython	0	2	2	4	qt	0	1	1	2	nomad	1	1	1	3
glibc	0	3	1	4	grpc	0	0	1	1	mattermost	0	0	2	2
gnupg	0	1	2	3	v8	0	0	1	1	moby	0	1	1	2
krb5	0	1	1	2						Valut	0	1	0	1
libwebp	0	1	1	2						cilium	0	0	1	1
FreeRDP	0	1	0	1						darp	0	0	1	1
OpenSSL	0	0	1	1						etcd	0	1	0	1
libpng	0	1	0	1						go-jose	0	1	0	1
php	0	1	0	1						Grafana	0	0	1	1
sqlite	0	0	1	1										
Subtotal	6	29	34	69	**Subtotal**	0	11	17	28	**Subtotal**	13	9	27	49
									Total					146

Table 3.1: **Overview of Our Dataset.** Type-II: 19, Type-III: 49, Type-IV: 78, Total: 146.

Datasets We evaluate PORTGPT using 1961 patches, which include 350 patches from FIXMORPH [123], 1465 patches from TSBPORT [124]. And 146 patches collected by ourselves (details in Table 3.1). These patches span 34 programs across three languages (C, C++, and Go), with the number of modified lines ranging from 1 to 1792. These 146 patches are collected based on the following criteria: ❶ The corresponding vulnerability of the patch is recorded in the CVE database within the past four years. ❷ At least one version within the same project has not reached end-of-life and requires backporting. ❸ The patch cannot be trivially applied to the target version, *i.e.,* not Type-I.

Evaluation Criteria We compare PORTGPT with FIXMORPH and TSBPORT using their datasets (1,815 cases in total) along with all C cases in our dataset. Comparisons with C++ and Go cases

from our dataset were excluded, as both tools support C only. Although we attempted to adapt these tools for use with C++ and Go, their implementations are highly tailored to C, making migration infeasible.

Additionally, unlike PORTGPT and FIXMORPH, TSBPORT requires the target file for each hunk as input—a condition that assumes some degree of ground-truth patch localization, which is not entirely realistic. To ensure a fair comparison, we introduce a variant of TSBPORT, denoted as TSBPORT*, that operates under the same settings as PORTGPT and FIXMORPH. TSBPORT* is designed to function without access to the target file, thereby simulating a more realistic environment. By aligning the evaluation conditions, we enable a more practical and meaningful comparison of the performance among the three tools.

We excluded SKYPORT [143] from our comparison, as it is designed for web applications (PHP only), which is not included in our dataset. Mystique [125] and PPATHF [126] was excluded as it is designed for function-level patch porting and requires perfect localization, whereas backporting typically involves multiple functions. Although we attempted to compare PATCHWEAVE [132], its dependence on KLEE symbolic execution presents challenges. The use of symbolic execution can result in path explosion, which makes it impractical for large-scale programs (*e.g.,* Linux kernel). Furthermore, PATCHWEAVE requires PoC for the corresponding vulnerabilities, which is not available in our dataset.

Validation Methodology To validate the correctness of the patches generated by PORTGPT, we compare them with the ground truth. If the patches are not identical, we conduct human validation to ensure their correctness, following best practices outlined in [123], [124]. Each patch is independently reviewed by three researchers to guarantee accuracy and consistency. We also try our best to collect PoC and test suites for each vulnerability in our dataset, following the valida-

Dataset	System	Type-I	Type-II	Type-III	Type-IV	Total
Prior works [123], [124]	FIXMORPH	20/170 (11.67%)	374/1208 (30.96%)	23/92 (25.00%)	30/345 (8.70%)	447/1815 (24.63%)
	ChatGPT†	67/170 (39.41%)	451/1208 (37.30%)	16/92 (17.58%)	22/345 (6.38%)	556/1815 (30.63%)
	TSBPORT	170/170 (100.00%)	1190/1208 (98.51%)	69/92 (75.00%)	160/345 (46.38%)	1589/1815 (87.59%)
	TSBPORT*	162/170 (95.29%)	919/1208 (76.08%)	61/92 (66.30%)	150/345 (43.48%)	1292/1815 (71.18%)
	PORTGPT	170/170 (100.00%)	1186/1208 (98.18%)	74/92 (80.43%)	188/345 (54.49%)	1618/1815 (89.15%)
Ours (C)	FIXMORPH	N/A	0/6 (0.00%)	3/29 (10.34%)	0/34 (0.00%)	3/69 (4.34%)
	TSBPORT	N/A	5/6 (83.33%)	14/29 (48.28%)	5/34 (14.71%)	24/69 (34.78%)
	TSBPORT*	N/A	2/6 (33.33%)	14/29 (48.28%)	4/34 (11.76%)	20/69 (28.99%)
	PORTGPT	N/A	6/6 (100%)	21/29 (72.41%)	15/34 (44.12%)	42/69 (60.87%)
Ours (C++)	PORTGPT	N/A	N/A	10/11 (90.91%)	5/17 (29.41%)	15/28 (53.57%)
Ours (Go)	PORTGPT	N/A	13/13 (100%)	7/9 (77.78%)	14/27 (51.85%)	34/49 (69.39%)

Table 3.2: **Performance Comparison of Backporting.** The first two sections of the table compare PORTGPT with FIXMORPH and TSBPORT using datasets from prior works as well as C cases on our own dataset. The last two rows display the performance of PORTGPT on C++ and Go cases from our dataset. ChatGPT† is from TSBPORT [124]. **TSBPORT*** represents the performance of TSBPORT under the same settings as PORTGPT and FIXMORPH. Unlike PORTGPT and FIXMORPH, TSBPORT requires the target file for each hunk as input, which may not be available in real-world scenarios. To ensure a fair comparison, we introduce **TSBPORT*** as a variant of TSBPORT that operates without access to the target file, simulating a more realistic environment.

tion chain described in Section 3.5. To evaluate whether PORTGPT-generated patches introduce new security vulnerabilities, we run directed fuzzing (SyzDirect [144]) on successfully backported Linux patches that differ from ground-truth. No security risks were identified during 4 hours of fuzzing.

3.6.2 Performance Evaluation (RQ1)

Table 3.2 provides a summary of PORTGPT's performance, which is based on GPT-4o, in comparison to FIXMORPH and TSBPORT, while Table 3.3 illustrates PORTGPT's performance when powered by different LLMs.

Model	Type-II	Type-III	Type-IV	Total
GPT-5	19/19	42/49	31/78	92/146
GPT-4o	19/19	38/49	34/78	91/146
Gemini-2.5-Flash	19/19	37/49	27/78	82/146
DeepSeek-v3	18/19	33/49	25/78	76/146
Llama-3.3	17/19	7/49	0/78	24/146

Table 3.3: **PORTGPT Performance based on Different Large Language Model.**

3.6.2 Comparison with FIXMORPH

In the dataset from prior works [123], [124], PORTGPT achieves notable performance enhancements across all backporting types when compared to FIXMORPH. For instance, in Type-IV patches, FIXMORPH demonstrates a modest success rate of 8.70% while PORTGPT improves it to 54.49%, showcasing its resilience in addressing complex backporting scenarios. The performance advantage of PORTGPT is more remarkable in our dataset for C test cases. Across all types, FIXMORPH achieves a success rate of 4.34%, PORTGPT demonstrates superior efficacy with a success rate of 60.87%.

We analyzed the failure cases of FIXMORPH within our dataset. Firstly, FIXMORPH enforces stricter constraints by supporting only C source files while excluding C header files. This limitation is explicitly stated in the paper and is further confirmed by its implementation, which resulted in 14 failed cases. Among other failed cases, FIXMORPH failed to produce results in 33 instances, primarily because it raised errors that caused the transformation process to terminate prematurely. In seven of these failures, FIXMORPH was unable to identify the correct patch locations in the target version. In the remaining cases, its AST-based approach was unable to handle complex changes, such as code syntax restructuring and module-level semantic modification, resulting in failures during the AST transformation process.

FIXMORPH generated a total of 22 patches out of which 19 are incorrect, and they exhibit

several issues. These patches contained syntax errors, such as disordered statements or failure to utilize functions or variables compatible with target versions. Furthermore, FIXMORPH struggled to accurately migrate patches involving changes in semantics. Moreover, the complexity of the code plays a crucial role in the performance of FIXMORPH. The average number of modified lines for successfully backported patches is 1.3, compared to 40.1 for failed patches. This difference indicates that FIXMORPH struggles with cases involving intricate logic and extensive code modifications.

3.6.2 Comparison with TSBPORT

In the dataset from prior works [123], [124], PORTGPT achieves comparable results to TSBPORT on simpler patch types but excels in handling more complex types, with success rates of 80.43% and 54.49% on Type-III and Type-IV, respectively, compared to 75.00% and 46.38% in TSBPORT. On our dataset, the advantage of PORTGPT becomes even more evident. It achieves 100% accuracy on Type-II cases and significantly outperforms TSBPORT on Type-III (72.41% vs. 48.28%) and Type-IV (44.12% vs. 14.71%).

In addition, TSBPORT requires manual designation of the target file for each hunk and we provide it with the ground truth in the experiments, which favors TSBPORT. In particular, TSBPORT does not handle the issue of mismatched target and original patch files, which is the main challenge in Type-II (as these patches requires no transformation at all). To address this evaluation bias, we limit files from the original patch as the target in TSBPORT*. PORTGPT outperforms TSBPORT* by 17% overall.

The underperformance of TSBPORT on Type-III and IV cases in our dataset stems from several issues. For Type-III, which involves 15 failure cases, the primary reason is erroneous symbol importing. TSBPORT often struggles to resolve missing symbols, such as function or global vari-

able declarations in header files, during patch migration. This limitation frequently leads to failed patch applications or introduction of additional header files imports, accounting for errors in 11 out of the 15 cases. These extraneous header file imports are typically migrated from the header files in the original version, but these headers do not exist in the target version, causing compilation error.

Type-IV encompasses 29 failure cases, which can be attributed to two primary issues. The first issue is TSBPORT's inadequate handling of module-level modifications, such as updates to struct fields or the introduction of new functions. This deficiency leads to incorrect mappings, accounting for 7 out of the 29 failures. The second issue is the complexity of the code itself. This is evidenced by a notable difference in the average number of modified lines: 13.8 for successful patches versus 35.7 for failed ones. The contrast highlights TSBPORT's difficulty in dealing with complex logic and extensive modifications. Note that PORTGPT faces the same difficulty as well—more complicated patches are harder to backport—but with a higher bar since the average number of modified lines for successful patches reached 20.1, while for failed patches, it averaged 51.0.

3.6.2 Generalizability over other programming languages

PORTGPT demonstrates strong programming language generalizability. Unlike FIXMORPH and TSBPORT, which are coupled with C only, PORTGPT consistently performs well across multiple programming languages. Notably, adapting PORTGPT from C to C++ and Go required no modification, highlighting its flexibility. These results underscore PORTGPT's potential for robust generalizability and its capability to be effectively extended to support a broader range of programming languages. TSBPORT and FIXMORPH required predefined transformation rules based on expert knowledge, whether for syntax or semantic matching, and were tightly coupled with spe-

cific programming languages. In contrast, PORTGPT relies on a series of text-based tools (e.g., GitUtils) for localization and leverages the code generation capabilities of LLM to perform patch transformation. This allows PORTGPT to be language-agnostic.

However, PORTGPT's performance on Type-IV C++ cases is significantly lower, with a success rate of only 5 out of 17 cases (29%) compared to at least 40% in other languages. This discrepancy may partly stem from the small dataset, which could introduce statistical bias. Another contributing factor is the complexity of required modifications: 11 out of 12 C++ failure cases necessitate around 100 lines of changes, whereas most C cases require fewer than 50 lines.

3.6.2 Effectiveness of other large language models

As shown in Table 3.3, PORTGPT demonstrates varying performance across different language models. GPT-5 achieves the best overall performance with 92/146 (63.0%) success rate, followed closely by GPT-4o with 91/146 (62.3%). Gemini-2.5-Flash delivers solid performance with 82/146 (56.2%) overall success, while DeepSeek-v3 shows moderate effectiveness at 77/146 (52.7%). This decline in Llama 3.3 performance is primarily attributed to Llama 3.3's limited function-calling capabilities, which are critical for retrieving information in the backporting task. Notably, Llama 3.3 often aborts processes prematurely when faced with insufficient information, instead of attempting additional tool invocations. Moreover, while GPT-5 consistently executes tool calls accurately, Llama 3.3 frequently generates function-calling messages with incorrect formats or parameters, severely compromising its effectiveness. The Berkeley Function Calling Leaderboard [145] further validates these other models' superior performance compared to Llama 3.3 in function-calling tasks. While DeepSeek-v3 demonstrates better performance in tool invocation (compared to Llama 3.3), its accuracy still falls short of the other models due to its limitations in code comprehension and contextual understanding.

Line Range	[1, 5]	[6, 10]	[11, 30]	[31, 50]	[51, 100]	[101, 200]	[200, ∞)
Prior works [123], [124]							
PORTGPT	744/787 (94.5%)	339/373 (90.9%)	349/401 (87.0%)	100/131 (76.3%)	68/92 (73.9%)	16/24 (66.7%)	2/7 (28.6%)
TSBPORT*	537/787 (68.2%)	274/373 (73.5%)	297/401 (74.0%)	101/131 (77.1%)	59/92 (64.1%)	20/24 (83.3%)	2/7 (28.6%)
FIXMORPH	337/787 (42.8%)	85/373 (22.8%)	23/401 (5.7%)	2/131 (1.5%)	0/92 (0%)	0/24 (0%)	0/7 (0%)
Our Dataset (C)							
PORTGPT	10/12 (83.3%)	8/10 (80.0%)	14/21 (66.7%)	5/11 (45.5%)	3/7 (42.8%)	1/3 (33.3%)	2/5 (40.0%)
TSBPORT*	6/12 (50.0%)	4/10 (40.0%)	4/21 (19.0%)	3/11 (27.3%)	4/7 (57.1%)	0/3 (0%)	0/5 (0%)
FIXMORPH	3/12 (25.0%)	0/10 (0%)	0/21 (0%)	0/11 (0%)	0/7 (0%)	0/3 (0%)	0/5 (0%)

Table 3.4: **Performance Analysis by Patch Size.** Success rates of different tools across varying patch sizes (measured in modified lines) on prior datasets from and our new C dataset. Numbers indicate successful patches out of total attempts with success percentages in parentheses.

3.6.2 Impact of Patch Size Distribution

To better understand the relationship between patch size and tool effectiveness, we analyze the distribution of patch lengths in both prior datasets and our constructed dataset. The results consistently demonstrate that the effectiveness of all tools declines as patch length increases. This trend highlights two key challenges: (1) larger hunks are inherently more difficult to handle, and (2) longer patches increase the likelihood of encountering at least one challenging hunk, thus lowering the overall success rate. Table 3.4 summarizes the performance of PORTGPT, TSBPORT*, and FIXMORPH across different patch length ranges, combining both prior datasets and our dataset. Overall, all tools exhibit a monotonic decline in success rate as patch length increases. For instance, on prior datasets, PORTGPT achieves a success rate of 94.5% on patches of 1–5 lines, but the rate drops to 66.7% for patches of 100–200 lines. FIXMORPH is particularly sensitive to patch length, with effectiveness approaching zero beyond 30 lines, while TSBPORT* shows a less stable trend and also struggles on very long patches.

Configuration	Type-II	Type-III	Type-IV	Total
w/o GitUtils	94.74%	75.51%	32.05%	54.79%
w/o Locate	94.74%	69.39%	38.46%	56.16%
w/o Loc & Git	84.21%	59.18%	25.64%	44.52%
w/o AutoFix	100%	65.31%	30.77%	51.37%
w/o Compile	100%	69.39%	25.64%	50.00%
PORTGPT	**100%**	**77.55%**	**43.59%**	**62.33%**

Table 3.5: **Ablation Study of PORTGPT.** w/o = without. GitUtils, Locate (LocateSymbol) and Compile (CompileTest) are three tools in PORTGPT. AutoFix refers to the patch correction mechanism.

3.6.3 Ablation Study (RQ2)

We conducted an ablation study, selectively disabling each of PORTGPT's well-designed tools to measure their impact on patch backporting performance. With each case requiring four evaluations under this protocol, testing a dataset of approximately 2000 cases leads to prohibitive monetary costs, estimated in the thousands of dollars. Therefore, to quantify each tool's contribution to the LLM's backporting capabilities under these constraints, our analysis only utilized our dataset.

The results, shown in Table 3.5, highlight the contributions of each tool. Firstly, we assess *GitUtils* and *LocateSymbol* tools that contribute to patch localization. Though these two tools cooperate to achieve better performance, they could also work independently. Disabling *GitUtils* caused a sharp drop in overall performance, particularly in Type-IV cases, where the accuracy plummeted to 32.05%. This outcome underscores the importance of *GitUtils* for handling complex relationships between commits. Similarly, removing the *LocateSymbol* tool led to reduced performance, especially in Type-III cases, where accuracy fell to 69.39%, demonstrating its critical role in identifying symbol locations. Removing both tools simultaneously results in the most severe degradation, with overall accuracy dropping to 44.52%, confirming their synergistic effect in patch localization.

Type	Avg. Token		Avg.	Avg.
	# Input	# Output	$ Cost	Time (s)
Type-I	20,152	1,057	0.06	53.22
Type-II	31,831	2,180	0.10	100.25
Type-III	57,146	4,169	0.19	193.66
Type-IV	89,804	4,776	0.27	222.52
Total	60,649	3,589	0.19	166.58

Table 3.6: **Average Time & Money Cost of PORTGPT.**

Secondly, we assess AutoFix mechanism in *ApplyHunk* and *CompileTest* tools that contribute to patch transformation. Disabling AutoFix caused performance to degrade across all types, with Type-III and Type-IV cases being particularly affected, showing accuracies of 65.31% and 30.77%, respectively. This result underscores AutoFix's critical role in refining intermediate patch quality. Removing *CompileTest* also led to performance drops across all case types, reflecting its importance in final patch combination and integration. Overall, the complete system, PORTGPT, achieves the best performance, excelling in challenging scenarios such as Type-IV cases with 43.59% accuracy and achieving a total accuracy of 62.33%. These results confirm that the integration of all components is essential for optimal system performance.

3.6.4 Efficiency Evaluation (RQ3)

The efficiency of PORTGPT is evaluated on the whole dataset in terms of average token usage, monetary cost, and processing time, as shown in Table 3.6. Across different input-output scenarios (Type-I to Type-IV), the average time and cost scale proportionally with the complexity of the task, with larger input and output sizes leading to higher resource consumption. As the complexity of patches increases (from Type-I to Type-IV), the LLM requires more information queries and performs more reasoning, leading to higher costs. The total average input token count is 60,649,

CVE ID	Version	Fix Date	Type	P.G.
CVE-2024-46743 [146]	Bionic	2024/10/15	II	✓
CVE-2023-51779 [147]	Focal	2024/01/05	III	✓
CVE-2024-0565 [148]	Focal	2024/01/29	III	✓
CVE-2024-26922 [149]	Focal	2024/05/21	III	✓
CVE-2024-43863 [150]	Focal	2024/12/03	III	✓
CVE-2024-43863 [150]	Bionic	N/A	III	✓
CVE-2023-24023 [151]	Focal	2024/03/14	IV	✗
CVE-2023-52752 [137]	Bionic	2024/06/26	IV	✗
CVE-2023-52752 [137]	Focal	2024/06/26	IV	✓
CVE-2023-52752 [137]	Jammy	2024/06/26	IV	✗
CVE-2024-24860 [152]	Bionic	2024/07/09	IV	✗
CVE-2024-24860 [152]	Focal	2024/07/09	IV	✗
CVE-2024-25742 [153]	Jammy	2024/07/02	IV	✗
CVE-2024-26922 [149]	Bionic	2024/05/21	IV	✓
CVE-2024-41066 [154]	Focal	2024/11/25	IV	✓
CVE-2024-41066 [154]	Jammy	2024/11/25	IV	✓

Table 3.7: **Ubuntu Backporting Tasks Results. Version** refers to the target version for Ubuntu backporting tasks. **Fix Date** represents the actual date of the backport in the real world.

producing an average of 3,589 output tokens, with a cost of $0.19 and a processing time of 166.58 seconds. This balance between cost-effectiveness and processing speed demonstrates the practical scalability of PortGPT for various levels of task complexity, maintaining reasonable efficiency even for the most demanding scenarios.

3.6.5 Real-World Applicability Study (RQ4)

To evaluate the practical applicability of PortGPT, we conducted studies across two prevalent and challenging real-world backporting scenarios: (1) porting patches from the mainline Linux kernel to a LTS version, and (2) propagating patches from an LTS version to downstream distributions. All patches selected for these scenarios were introduced after the knowledge cutoff of the underlying LLM (GPT-4o), allowing for a robust assessment of PortGPT's generalization to unseen tasks.

Mainline To LTS For this scenario, our selection process focused on challenging CVE patches (2024-2025, sourced from Linux vulns [155])—specifically those that had previously failed backporting from mainline to Linux 6.1-stable and lacked manual conflict resolution. This rigorous filtering yielded a focused set of just 18 such cases. PORTGPT successfully handled 9 of these. Critically, all 9 generated patches were subsequently reviewed and merged into the 6.1-stable branch by kernel maintainers [156], [157], [158], [159], [160], [161], [162], [163], [164]. For the remaining 9 cases, 7 failed due to significant structural code changes, and 2, though directly cherry-pickable, exhibited regressions post-application [165]. In contrast, TSBPORT handled only 2/18 cases (11.1%), underscoring PORTGPT's improved applicability for complex mainline-to-LTS backports.

LTS To Downstream We evaluated PORTGPT on 16 patch pairs (10 CVEs) for Ubuntu (targeting Jammy, Focal, Bionic versions) [166]. All patches were introduced after October 2023 to test generalization when porting from LTS sources to downstream targets. PORTGPT successfully backported 10 out of 16 tasks (as shown in Table 3.7). This demonstrates PORTGPT's robust generalization for LTS-to-downstream backporting of unseen patches.

3.6.6 Case Studies

In this part, we present two cases to illustrate how the design of PORTGPT effectively supports LLM reasoning and describe the main reasons behind the failures of PORTGPT and present several case studies to illustrate the underlying causes of these failures.

From a high-level perspective, the reasons for failure can be summarized into three points: 1) The target version contains additional vulnerable code compared to the original version, which requires fixing; 2) The hunk-by-hunk backport approach overlooks potential dependencies between

hunks. 3) PORTGPT only backports the current patch and does not incorporate its prerequisite commits. The following cases describe these three points in detail.

CVE-2020-22030 In both versions, the patch introduces a length check before usage as the core fix. However, as code evolves across versions, the function and struct names, as well as their usages in the patch, have changed. As a result, beyond resolving contextual conflicts, the added logic in the patch also requires careful adaptation to align with the target version's codebase, which poses a significant challenge for automated backporting tools. Specifically, the function `ff_inlink_queued_samples` was replaced by `ff_framequeue_queued_samples`, and the field `inputs` was used differently. In this case, the *History* component in PORTGPT's *GitUtils* module effectively provided relevant historical commit information to the LLM, enabling it to reason about and adapt the necessary changes. In contrast, due to the lack of historical context, TSBPORT reused the mainline implementation without adapting to version-specific changes, leading to an incorrect backport.

CVE-2023-41175 The added lines in the patch also require adaptation to fit the target version. Specifically, the macro `UINT_MAX` does not exist in the target version. However, this issue cannot be inferred from the patch's code context or its historical modifications, and can only be revealed through compilation and testing. TSBPORT, which performs no post-generation verification (including compilation), produces a "seemingly correct" patch that ultimately fails. In contrast, PORTGPT adopts a two-stage design that effectively eliminates hard-to-infer transformations during backporting. Its iterative process, compilation testing and patch correction based on observed errors, also closely aligns with real-world backporting practices.

```
@@ -4397,8 +4398,15 @@ __cgroup_procs_write(
  spin_unlock_irq(&css_set_lock);
+ saved_cred = override_creds(of->file->f_cred);
  ret = cgroup_attach_permissions(src_cgrp,
  dst_cgrp, of->file->f_path.dentry->d_sb, ...);
+ revert_creds(saved_cred);
  if (ret)
      goto out_finish;

@@ -4440,9 +4449,15 @@ cgroup_threads_write(
  spin_unlock_irq(&css_set_lock);
+ saved_cred = override_creds(of->file->f_cred);
  ret = cgroup_procs_write_permission(src_cgrp,
      dst_cgrp, of->file->f_path.dentry->d_sb);
+ revert_creds(saved_cred);
  if (ret)
      goto out_finish;
```

Listing 13: **CVE-2021-4197.** Both the original [167] and target [168] version patches include the first hunk, but only the target version requires modification of the second hunk.

CVE-2021-4197 [167], [168] CVE-2021-4197 highlights a common failure reason in PORTGPT's backporting process, as demonstrated in Listing 13. The patch in the original version [167] includes the first hunk, which introduces proper credential handling through `override_creds` and `revert_creds` for `cgroup_attach_permissions`. While PORTGPT successfully applies this modification to the corresponding code in the target version, it fails to address a similar vulnerability in the second hunk involving `cgroup_procs_write_permission`. Both code segments require identical security enhancements to handle credentials correctly [168]. The oversight leaves the second vulnerability unpatched and cause PORTGPT generates an incomplete patch. In the mainline version, the second segment does not exist since `cgroup_threads_write` is implemented with `__cgroup_procs_write`.

CVE-2019-16232 [169] CVE-2019-16232 illustrates a failure in PORTGPT's backporting process. In the original version, the patch consists of two related hunks: the first introduces the error

handling statement `goto err_queue;`, and the second defines the behavior of the `err_queue` label. During backporting, PORTGPT replaces `err_queue` with `out` in the first hunk because it cannot find the `err_queue` label in the target version's context. When processing the second hunk, PORTGPT directly adds the `err_queue` label as in the original patch, but the label does not integrate correctly because it was not previously established. We believe this failure is due to PORTGPT's hunk-by-hunk approach, which fails to account for dependencies between hunks.

CVE-2022-41720 [170], [171] CVE-2022-41720 is a vulnerability in the Golang project that highlights a limitation in PORTGPT 's backporting process. The patch in the master branch [170] revises the `join` function to address the issue; however, this function does not exist in the target version. To address this, the patch for the target version [171] first introduces the `join` function before modifying it. PORTGPT fails to backport the patch because it cannot locate the `join` function in the target version and incorrectly assumes that no backporting is needed. This failure stems from PORTGPT's inability to follow a common backporting practice where developers include prerequisite commits to adjust the context before applying the actual patch. Addressing this limitation in future work will enable PORTGPT to better handle complex backporting scenarios, aligning more closely with experienced developer practices.

3.7 Discussion and Limitations

In this section, we discuss the limitations of PORTGPT.

Results Interpretation While PORTGPT demonstrates promising results, it currently cannot consistently provide reliable interpretations of backported patches to explain their correctness. This limitation arises because LLMs are statistical models of syntactic representation. Although they can generate syntactically correct code, they lack the ability to provide a robust semantic

understanding. Therefore, we recommend that PORTGPT users treat backported patches and outputs as suggestions, carefully reviewing the patches manually before applying them to the target codebase.

Context Length One of the key limitations of LLMs is their restricted context length, which can hinder performance when the input exceeds a certain size. Research [136] has shown that the effectiveness of LLMs decreases as the length of the input increases, due to their inability to process long-range dependencies effectively. In PORTGPT, this issue is mitigated by processing the hunks of the original patch separately during the first stage, allowing each part to be handled within the context length constraints. However, this strategy comes with its own challenges, as it can result in a loss of crucial context from the original patch, potentially affecting the accuracy of the backporting process.

3.8 Related Work

Program Repair Automated program repair (APR) aims to automatically fix bugs in software systems, sharing a goal similar to backporting but operating without access to the original patch. Among the various tasks in APR, patch generation has received the most attention. This task involves taking a buggy code snippet as input and producing a patch to fix the bug. Many approaches leveraging deep learning [172], [173], [174] and large language models (LLMs) [8], [11], [28], [29] have shown promising results in this area. A prerequisite task for APR is fault localization (FL), which identifies the buggy code snippet. This process resembles PORTGPT, where LLMs are used to determine the corresponding patch location of a hunk. However, FL in APR is often more challenging due to the absence of the original patch. Current FL methods are predominantly statistical [31], [32], [51], [52], scoring elements in the program to analyze root causes. Another

critical step in APR is patch validation, which is typically conducted through either manual review [35] or automated testing [36]. In this work, we primarily rely on manual efforts to validate the backported patches.

LLM for SE LLM have become powerful tools for various software engineering (SE) tasks, such as vulnerability detection [115], [116], code translation [117], [118], program repair [8], [9], [36], [175], and unit test generation [176], [177]. To assess their effectiveness in real-world applications, benchmarks like SWE-bench [27] have been developed. SWE-bench consists of many GitHub issues and their corresponding pull requests from widely-used Python repositories, providing a challenging dataset for evaluating how well models can generate patches to resolve specific codebase issues. Several notable tools have been evaluated using SWE-bench. SWE-Agent [77] leverages LLMs to autonomously tackle GitHub issues, utilizing a custom Agent-Computer Interface (ACI) to improve the model's ability to navigate, edit, and test code within repositories. Similarly, AutoCodeRover [114] integrates LLMs with advanced code search mechanisms, leveraging program structures such as abstract syntax trees to enhance bug fixing and feature implementation. Building on strong capabilities of LLMs, our work uses these models to facilitate patch backporting between different versions.

3.9 Conclusion

Patch backporting is a complex and labor-intensive task in the real world, which increases the burden of project maintainers. In this work, we introduced PortGPT, an LLM-based tool designed for end-to-end automated patch backporting. Using the modification history effectively in Git, PortGPT employs a combined approach to achieve precise localization, guiding the LLM to make inferences based on the provided information, thus simulating the expert backporting

process. Our extensive evaluation on diverse datasets demonstrate that PORTGPT significantly outperforms existing automated patch backporting tools. Evaluations on multi-language, multi-project, and more complex patch types also demonstrate PORTGPT's ability to handle diverse scenarios and its versatility.

CHAPTER 4

PATCH VALIDATION IN AUTOMATED VULNERABILITY REPAIR

4.1 Introduction

Patch validation—ensuring that a patch effectively addresses security vulnerabilities while preserving functional integrity—is crucial in software development. While traditionally applied to human-written patches [35], [37], patch validation has become increasingly important for automated vulnerability repair (AVR) systems [36], [178], [179], where it serves to evaluate auto-generated patches. Hence, the reliability of AVR effectiveness evaluation therefore depends on the accuracy and comprehensiveness of their underlying patch validation methodologies.

In an abstract view, an AVR tool takes, at minimum, a vulnerable codebase C and a proof-of-concept (PoC) input poc that demonstrates the existence of a vulnerability in C, and produces a patch $\hat{p}$ that is supposed to fix the vulnerability. Many AVR tools [39], [40], [43] that achieved good performance in the pre-LLM era focused exclusively on vulnerabilities exhibiting fixed patterns, limiting their generalizability to other vulnerability classes. As code generation and completion capabilities advance through machine learning techniques, particularly large language models (LLMs), researchers are increasingly exploring their application in AVR [36], [178], [179]. Several recent studies [11], [180], [181] have demonstrated promising repair rates far exceeding best AVR tools before the LLM era.

Regarding validating the generated patch $\hat{p}$, our survey of the recent literature reveals three primary methodologies adopted in previous AVR works, including *manual comparison* that relies on expert assessment [40], [180], [182], *similarity metrics* that employ similarity-based scor-

ing [172], [173], [183], and *test suite validation* that executes automated tests to validate patch correctness [11], [36], [181], [184]. Both manual comparison and similarity metrics compare against a ground-truth patch p which is typically developer-written and committed into the codebase as the official bugfix. While manual comparison can provide high accuracy in principle, it lacks scalability for large-scale evaluation [185], [186]. Similarity-based metrics offer automation but have been shown to inadequately correlate with actual patch effectiveness [184], as patches achieving high similarity scores may still fail to address the vulnerability.

Test suite-based validation, on the other hand, does not necessarily require a ground-truth patch p to compare $\hat{p}$ with. Instead, it assumes that the codebase C comes with a comprehensive test suite T that pins down the intended behaviors of the program. Therefore, if the patched codebase ($\hat{p}$ applied to C) passes all tests in T and additionally mitigates poc, then $\hat{p}$ is correct. Perhaps due to its close resemblance to typical CI/CD pipelines in mature codebases when bugfixes are introduced, test suite-based validation has emerged as the predominant method in AVR research and practice, adopted by the majority of state-of-the-art tools including PATCHAGENT [11], SAN2PATCH [181], SWE-Agent [187] and others [36], [188], [189], [190].

However, while an AVR tool should never has access to the ground-truth patch p when generating $\hat{p}$, it does not implies that we should evaluate the AVR tool completely ignoring p. In fact, based on the generally accepted principle that *new code should have a new test* we expect new tests t associated with the ground-truth patch p in mature open-source projects—effectively, this means an updated test suite T' in the patched codebase. So this raises a question: **shall we evaluate $\hat{p}$ with T' instead of T and poc?**

This question seems trivial at first glance, as it is tempting to assume that t and poc are equally useful in evaluating $\hat{p}$ (hence T combined with poc is effectively the same as T'). However, as shown in Section 4.3, a patch $\hat{p}$ that mitigates poc and passes T does not mean that it fixes the

vulnerability in the way intended by the developer, and such intention can be encoded in the new test t associated with the patch p from the developers. This validation gap raises a critical question: **does the use of T and poc validation substantially overestimate the performance of AVR tools due to their inability to guarantee patch correctness?**

We name the new test t written by developers for the official patch p as PoC$^+$ in this paper to honor the fact that t is often derived from the poc but potentially encodes more semantics in terms of how developers intend to fix the vulnerability, and we argue that the auto-generated patch $\hat{p}$ of AVR tools should be evaluated using T', including the PoC$^+$. We constructed a dataset called PVBENCH, comprising 209 cases across 20 open-source projects. Each case includes both basic tests (T and poc) and PoC$^+$ tests (t). Using PVBENCH, we re-evaluate three state-of-the-art AVR tools, PATCHAGENT [11], SAN2PATCH [181], and SWE-Agent [187], [191]. Our results reveal that over 40% of patches validated as correct by basic tests failed when evaluated against PoC$^+$ tests, corresponding to a high false discovery rate in statistical terms. This stark contrast exposes a critical weakness in test suite-based evaluation: commonly used validation methods substantially overestimate AVR tool effectiveness.

To assess whether PoC$^+$ tests serve as a reliable patch validation method, we manually compared patches that passed PoC$^+$ tests against developer-written patches. We found that over 70% of PoC$^+$-passing patches achieve semantic equivalence with the developer's patch, demonstrating PoC$^+$'s effectiveness in capturing developer intent and the inherent repair logic. The remaining patches exhibit issues such as suboptimal complexity or other quality concerns, which we detail in Section 4.6. Additionally, we analyze patches that fail PoC$^+$ tests and summarize the failure reasons in Section 4.7.

To understand how PoC$^+$ tests are commonly created by developers (and why they encode more program semantics than the poc), we surveyed the PoC$^+$ tests in PVBENCH and identified

three categories based on their validation mechanisms: *Output Checking*, *Intermediate Checking*, and *Self Checking*. Across all categories, developers transform the original PoC by capturing the expected behavior of the patched program, whether through output comparison, intermediate state assertions, or embedded runtime checks, and encoding these expectations into the test specification. Unlike a PoC, which only observes whether the program crashes, a PoC$^+$ test encodes richer program semantics by explicitly specifying the expected correct behavior. This additional semantic information enables more precise patch validation: a patch that merely suppresses a crash without restoring correct functionality will fail the PoC$^+$ test, whereas it might pass a crash-only PoC check.

In summary, our work makes four main contributions:

- We propose PoC$^+$ tests as an improved patch validation method for evaluating AVR tools and introduce PVBENCH to investigate their advantages over traditional test suite-based validation. We also describe our validation methodology and the process for producing these tests.
- We evaluate three state-of-the-art LLM-based AVR systems on PVBENCH, revealing that over 40% of patches validated as correct by basic tests fail when subjected to PoC$^+$ tests, demonstrating substantial overestimation in current evaluation practices for modern AVR tools.
- We validate the reliability of PoC$^+$ tests by comparing patches that pass them against developer patches. Over 70% achieve semantic equivalence with developer patches, while the remainder exhibit performance issues or other concerns. In other words, PoC$^+$ does not fully capture every detail in developers' intention, but is still an important step forward.
- We provide a comprehensive analysis of validation inadequacies by systematically categorizing falsely correct patches into three categories which might help guide APR tools towards

Table 4.1: Taxonomy of Patch Validation Methodologies Used in Previous AVR Works

Method	Category	Papers	Key Characteristics
Manual Comparison	Limited Scope	CONCH [40], FixMorph [192], SkyPort [143], TSBPort [193], PortGPT [12]	Check if patch is semantically equivalent to ground truth. Constrained to specific bug types (e.g., null dereference) or backporting scenarios.
	General Scope	Senx [194], APPatch [180], VRPILOT [182]	Check if patch fixes the bug without breaking functionality. Handles various vulnerability types without constraints.
Similarity Metrics	Similarity Scoring	VulMaster [183], VulRepair [172], VRepair [173]	Automated evaluation using common code similarity metrics: EM, BLEU-4, and CodeBLEU.
Test Suite Validation	PoC-only	ExtractFix [43], CodeRover-S [184], SAVER [39], VFix [195], CRASH-FIXER [196]	Validate patches by executing PoC to confirm mitigation. Focuses on security-specific validation; may overlook functional correctness.
	Full Testing	CRASHREPAIR [188], CPR [197], Fix2Fit [198], PATCHAGENT [11], SAN2PATCH [181], WilliamT [189], VulnFix [190], Zero-Shot [36]	Validate patches by executing both PoC tests and existing functional test suites. Ensures patches fix vulnerabilities while maintaining functionality.

better repair process.

4.2 Background: Patch Validation in AVR

This section surveys patch validation methods used in previous AVR works (Table 4.1), examining their practices and potential limitations.

Manual Comparison. This approach relies on human experts to assess patch quality through direct comparison with developer-provided patches. *Limited scope systems* target specific vulnerability types or constrained scenarios (e.g., CONCH [40] focuses on null pointer dereferences, while TSBPORT [193], SKYPORT [143], PortGPT [12], and FIXMORPH [192] address backporting tasks). *General scope systems* such as APPatch [180], VRPILOT [182], and Senx [194] handle

diverse vulnerability types, making evaluation more challenging since correct patches may differ structurally from reference solutions. Unfortunately, none of these works disclose the specific criteria examiners use during comparison, particularly when evaluating functionally correct but structurally different patches.

Limitations: Manual comparison provides arguably the highest level of accuracy, this approach is typically confined to scenarios with limited solution spaces. For instance, CONCH [40] focuses specifically on null dereference vulnerabilities that require straightforward null check additions, while three other studies [143], [192], [193] examine backporting scenarios where patches are adapted from mainline to stable versions. These backporting studies reveal that most patches require only minimal modifications—TSBPORT [193] found that 81% of backported patches involve location or namespace changes only—making manual verification feasible even for backporting tasks on large codebases (e.g., Linux kernel). For the other works that target general vulnerability repair [180], [182], [194], they all acknowledge that manual review is expensive and typically sample only a subset of cases for evaluation, highlighting the critical need for scalable automated patch validation methods.

Similarity Metrics. These approaches employ automated scoring without human intervention, typically combining three metrics: Exact Match (EM) [199] for binary character-level comparison, BLEU-4 [200] for token-level n-gram precision, and CodeBLEU [201] which extends BLEU with AST-based syntactic and data-flow semantic analysis. Representative systems include VulMaster [183], VulRepair [172], and VRepair [173].

Limitations: While code similarity metrics can be an automated way of measuring patch quality, they often fail to provide reliable indicators of patch effectiveness, as patches with high similarity scores may still fail when executed against PoCs [184]. This occurs because even minor syntactic differences—such as incorrect variable names, missing edge case handling, or subtle

logic errors—can render functionally similar-looking code completely ineffective. Critical factors like proper variable scoping, correct API usage, and precise conditional logic placement are often overlooked by similarity metrics despite being essential for patch functionality.

Test Suite-Based Validation. This widely adopted method validates patches through automated test execution. *PoC testing* focuses on exploit validation, exemplified by CodeRover-S [184], ExtractFix [43], SAVER [39], and VFix [195], though it may overlook functional correctness. *Comprehensive testing* combines PoC validation with functional test suites in the pre-patched codebase, as adopted by PATCHAGENT [11], SAN2PATCH [181], CRASHREPAIR [188], WILLIAM [189], Zero-Shot [36], VULNFIX [190], CPR [197], and Fix2Fit [198].

Limitations: Test suite-based validation in AVR remains contentious due to debating *overfitting* concerns, where patches may pass tests without addressing root causes. While early studies argued AVR tools are susceptible to overfitting [202], [203], [204], recent work [10], [205], [206] suggests modern LLM-based AVR tools exhibit less project-specific fitting due to training on diverse codebases. However, existing research fails to explain why patches pass tests yet miss root causes, and largely excludes memory-unsafe languages like C/C++ despite well-studied vulnerability patterns [17], [23], [207]. This knowledge may bias LLMs toward symptom-fixing rather than addressing root causes as shown in Section 4.3.

State-of-the-practice: Despite these potential problems, most modern AVR research continues to rely on test suite-based validation to decide patch correctness. This is likely due to its fully automated nature [178], [179] and its resemblance on how a manually written patch is applied on a production-grade codebase, which typically, and minimally, requires that all new code pushed to production must pass the CI/CD pipeline [104] which runs all existing tests. However, test suite-based validation makes a strong assumption on the comprehensiveness of existing test suite: the test suite should cover *all* intended behaviors of the program, which is unlikely to be true even

```
PHP_FUNCTION(range) {
  struct zval *user_start = /* Extract start argument */;
  struct zval *user_end = /* Extract end argument */;
  // Extract type from arguments
  uint8_t start_type = Z_TYPE_P(user_start);
  uint8_t end_type = Z_TYPE_P(user_end);
  /* If the range is given as strings,
     generate an array of characters. */
  if (start_type >= IS_STRING || end_type >= IS_STRING) {
    // VULNERABLE: condition fails when start_type=5 (IS_DOUBLE)
    // and end_type=7 (IS_ARRAY) because 5+7 = 2*6 (IS_STRING)
    if (start_type + end_type < 2*IS_STRING) {
      // ... handle mixed type inputs and convert to numeric
      goto handle_numeric_inputs;
    }
    // TYPE CONFUSION OCCURS HERE:
    //   When the vulnerable condition fails, we reach this point
    //   with non-string types, but try to access them as strings
    unsigned char low = Z_STRVAL_P(user_start)[0];
    unsigned char high = Z_STRVAL_P(user_end)[0];
    // ... character range generation logic
    return;
  }
handle_numeric_inputs:
  if (start_type == IS_DOUBLE || end_type == IS_DOUBLE) {
    // ... process numeric ranges (floats)
```

Figure 4.1: Type Confusion Bug in PHP Project

for mature codebases [208]. The practical implications of this assumption are illustrated in our motivating example (Section 4.3), which demonstrates how flawed patches can bypass current test suites. Our quantitative analysis in Section 4.5 further reveals the extent to which test suite-based validation influences AVR performance.

4.3 Motivation: PoC$^+$ Test

This section presents a motivating example of a minimum viable yet incorrect patch that would pass validation under conventional test suite-based automated patch validation methods. We subsequently introduce PoC$^+$ tests and demonstrate their effectiveness in exposing the flaws of such patches.

```
@@ -2924,8 +2924,8 @@ PHP_FUNCTION(range)
 /* If the range is given as strings,
    generate an array of characters. */
 if (start_type >= IS_STRING || end_type >= IS_STRING) {
-  if (start_type + end_type < 2*IS_STRING) {
+  if (start_type < IS_STRING || end_type < IS_STRING) {
     if (start_type < IS_STRING) {
       if (end_type != IS_ARRAY) {
         php_error_docref(NULL, E_WARNING, "...");
```

(a) Developer Patch

```
@@ -2960,6 +2960,14 @@ PHP_FUNCTION(range)
   }

   /* Generate array of characters */
+  if (Z_TYPE_P(user_start) != IS_STRING) {
+    zend_argument_value_error(1, "must be a string");
+    RETURN_THROWS();
+  }
+  if (Z_TYPE_P(user_end) != IS_STRING) {
+    zend_argument_value_error(2, "must be a string");
+    RETURN_THROWS();
+  }

   unsigned char low = Z_STRVAL_P(user_start)[0];
   unsigned char high = Z_STRVAL_P(user_end)[0];
```

(b) AVR-Generated Patch

Figure 4.2: Two distinct patch strategies for the type confusion vulnerability in PHP's `range()` function.

4.3.1 Plausible Patch

We use a type confusion vulnerability (Issue #13094 [209]) in the PHP interpreter to demonstrate plausible patches. As depicted in Figure 4.1, this code demonstrates a type confusion vulnerability in PHP's `range()` function that occurs due to flawed type checking logic. The bug arises when the function receives arguments of specific type combinations, such as a double (floating-point number) and an array. The first condition (line 9) correctly identifies that at least one argument appears to be string-like since $\texttt{IS\_ARRAY}(7) \geq \texttt{IS\_STRING}(6)$, so the code enters the string-handling branch. However, the inner arithmetic condition (line 12) unexpectedly fails when for example, $\texttt{IS\_DOUBLE}(5) + \texttt{IS\_ARRAY}(7) = 12$. When this condition fails, the code skips the

proper type conversion logic and incorrectly attempts to access the non-string data using string accessor macros (line 19), which cause the program crash. Two patches exist to address this vulnerability: one crafted by the developer and another representing a plausible solution generated by AVR tools.

Developer Patch. The developer patch (Figure 4.2a) addresses the bug by correcting the flawed arithmetic condition, ensuring that mixed-type inputs are properly redirected to the numeric handling branch instead of entering the string processing path.

AVR-Generated Patch. The patch (Figure 4.2b) generated by AVR tool takes a defensive programming approach by adding explicit type validation at the point of string access. It inserts runtime checks to verify that both arguments are actually strings before attempting to dereference them using `Z_STRVAL_P()`, throwing appropriate error messages if non-string types are encountered.

Comparison. When evaluated against the existing PHP functional test suite and PoC exploits, both patches successfully pass all tests and effectively mitigate the vulnerability. Automated test suite-based validation would therefore conclude that both patches are correct. However, manual inspection reveals fundamental differences in their control flow behavior. The developer patch conditionally redirects mixed-type inputs to follow the numeric conversion path. In contrast, the AVR-generated patch immediately terminates execution with an error for any mixed-type input. This divergence in control flow indicates that the patches are not functionally equivalent—one may introduce unintended behavioral changes, which are not covered by any of the existing tests.

4.3.2 PoC$^+$ Tests Reveal the Difference

To find evidence that the AVR-generated patch is incorrect, we observe that the developers not only provided the patch but also created a new test when fixing the bug, as is typical practice [210], and

```
---TEST---
GH-13094 (range(9.9, '0') causes segmentation fault)
---FILE---
<?php
var_dump(range(9.9, '0'));
?>
---EXPECT---
array(10) {
  [0]=>float(9.9)
  [1]=>float(8.9)
  [2]=>float(7.9)
  [3]=>float(6.9)
  [4]=>float(5.9)
  [5]=>float(4.9)
  [6]=>float(3.9000000000000004)
  [7]=>float(2.9000000000000004)
  [8]=>float(1.9000000000000004)
  [9]=>float(0.9000000000000004)
}
```

Figure 4.3: An PoC$^+$ test derived from the PoC for PHP issue #13094, validating correct behavior for a crashing input.

the test case is shown in Figure 4.3. This test case follows the PHP Test (PHPT) format. The format is a structured test specification that consists of several sections: the `--TEST--` section provides a descriptive name for the test case, the `--FILE--` section contains the actual PHP code to be executed, and the `--EXPECT--` section defines the exact output that should be produced. Given that the PHP code within this new test case closely resembles the original PoC, we refer to it as a PoC$^+$ test. When we run the PoC$^+$ tests on the program with the AVR-generated patch applied, we obtain the following output:

```
Fatal error: Uncaught ValueError: range():
Argument #1 ($start) must be a valid string in /test.php:2
Stack trace:
#0 /test.php(2): range(9.9, '0')
#1 {main}
  thrown in /test.php on line 2
```

This output differs significantly from the expected output. According to the PHP specification [211], the `range()` function is designed to be permissive with respect to argument types: when given a mixture of numeric and string arguments, it performs type coercion and generates a numeric range

Table 4.2: Overview of Projects and Vulnerabilities in PVBENCH

Project	LoC	#	Test	Project	LoC	#	Test
php [212]	1390.2K	43	18.7K	vim [82]	564.2K	11	5.2K
cpython [213]	745.9K	33	48.6K	hdf5 [214]	1334.4K	8	0.6K
llvm [215]	8980.4K	26	128.7K	exiv2 [216]	93.5K	7	0.3K
v8 [217]	6225.6K	24	53.7K	wabt [218]	514.9K	5	1.1K
libxml2 [80]	200.4K	19	3.3K	hermes [219]	590.0K	4	2.3K
icu [220]	1241.5K	15	2.0K	pcap++ [221]	160.0K	3	0.3K
quickjs [222]	78.8K	2	79.7K	libtiff [223]	109.0K	1	0.2K
mruby [224]	152.4K	2	1.7K	jasper [225]	5.5K	1	0.2K
jq [226]	4.7K	2	0.9K	simdjson [227]	547.5K	1	0.1K
htslib [228]	108.3K	1	0.4K	wireshark [229]	6088.9K	1	0.1K

Total Vulnerabilities: 209

CWE	#	Description	CWE	#	Description
CWE-476	52	NULL Dereference	CWE-670	3	Incorrect Control Flow
CWE-617	40	Reachable Assertion	CWE-415	3	Double Free
CWE-122	34	Heap Overflow	CWE-704	3	Type Confusion
CWE-416	32	Use After Free	CWE-457	1	Uninitialized Memory
CWE-190	26	Integer Overflow	CWE-362	1	Race Condition
CWE-121	13	Stack Overflow	CWE-369	1	Divide by Zero

if either argument is numeric. For example, a call such as `range(9.9, "0")` is valid and expected to produce a descending array of floating-point numbers, as demonstrated in the expected output of the PoC$^+$ test case. The developer's patch preserves this intended behavior by redirecting mixed-type arguments to the numeric range generation logic. In contrast, the AVR-generated patch breaks this specification: by enforcing that both arguments must be strings in the string-handling branch, it rejects mixed-type inputs and raises a runtime error, thereby violating the established PHP semantics. Thus, the AVR-generated patch is incorrect, as it introduces a change that deviates from the PHP language specification. The PoC$^+$ test automatically exposes this deviation, thus demonstrating the value of such tests for automated patch validation.

Table 4.3: PoC$^+$ Test Category Distribution by Projects

Category	Projects
Output Checking	exiv2, hermes, htslib, jasper, libxml2, php jq, llvm-project, simdjson, wabt, wireshark
Intermed. Checking	hdf5, icu, pcapplusplus, libtiff
Self Checking	cpython, mruby, quickjs, v8, vim

4.4 PoC$^+$ Test Dataset & Validation

This section presents the PVBENCH dataset and the validation method of different PoC$^+$ tests.

4.4.1 PVBENCH Overview

Vulnerability Statistic. PVBENCH provides a benchmark by incorporating 209 real-world vulnerabilities from 20 open-source projects. These projects represent widely-used systems with extensive codebases and robust test suites, ensuring the vulnerabilities reflect security issues encountered in production environments. As shown in Table 4.2, PVBENCH covers a diverse range of 12 CWEs, with memory safety issues being most prevalent, including NULL Dereference, UAF, and Heap Overflow, alongside control flow vulnerabilities such as Reachable Assertion and various data handling issues including Integer Overflow.

Selection Criteria. We identify open-source projects and vulnerabilities for PVBENCH by following a systematic workflow. First, we identified open-source projects based on GitHub star counts, prioritizing those with substantial vulnerability histories that are well-documented through GitHub issues or dedicated vulnerability tracking systems. This process led to the selection of 20 high-quality projects. Subsequently, we conducted a manual review of vulnerabilities reported in these projects over the past ten years, collecting cases that satisfy the following three requirements:

Reproducibility: Each vulnerability must include build scripts and at least one PoC to enable

```
class issue_2377_buffer_overflow(metaclass=CaseMeta):
    filename = "$data_path/issue_2377_poc.mp4"
    commands = ["$exiv2 $filename"]
    retval   = [253]
    stderr   = ["$filename: No Exif data found in the file\n"]
    stdout   = ["File name: $filename\nFile size: 225 Bytes\n..."]
```

```
// CHECK-LABEL: func.func @nested_muli() -> i32 {
// CHECK: %[[VAL_0:.*]] = "test.constant"() ...
// CHECK: %[[VAL_1:.*]] = arith.muli %[[VAL_0]], ...
func.func @nested_muli() -> (i32) {
  %0 = "test.constant"() {value = 0x7fffffff} : () -> i32
  %1 = "test.constant"() {value = -2147483648} : () -> i32
  ...
}
```

```
<!-- oss-fuzz-51295_0.xsd (input) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="e" substitutionGroup="e"/>
</xs:schema>

<!-- oss-fuzz-51295_0.err (expected error output) -->
element decl. 'e': The element declaration 'e' defines
a circular substitution group to element declaration 'e'.
```

Figure 4.4: PoC$^+$ Test Examples for Output Checking

compilation and reproduction.

Test Coverage: Each vulnerability must be accompanied by a functional test suite and PoC$^+$ tests developed by the maintainers.

Functional Preservation: The vulnerability have been fixed and the set of functionalities remain unchanged before and after applying the fix, ensuring that patches only resolve security vulnerabilities rather than adding a new feature. This requirement is necessary because otherwise PoC$^+$ tests may validate functionality that does not exist in the unpatched program.

4.4.2 Validation & Production Methodology

The PoC$^+$ test is derived from a PoC for the vulnerability. Unlike common PoCs, which typically only observe if the program crashes, the PoC$^+$ test performs more comprehensive validations to determine if the program behaviors are as expected. These validations include observing the output content or other intermediate running results. Based on the specific behaviors validated by the

Table 4.4: Script Path of Projects that Support PoC$^+$ Generation

Project	Automated Test Generation Scripts Path
Hermes	`utils/updateErrorTest.py`
libxml2	`codegen/genTestApi.py`
	`xstc/fixup-tests.py`
LLVM	`llvm/utils/update_analyze_test_checks.py`
	`llvm/utils/update_any_test_checks.py`
	`llvm/utils/update_cc_test_checks.py`
	`llvm/utils/update_llc_test_checks.py`
	`llvm/utils/update_mca_test_checks.py`
	`llvm/utils/update_mir_test_checks.py`
	`llvm/utils/update_test_checks.py`
	`llvm/utils/update_test_prefix.py`
PHP	`scripts/dev/bless_tests.php`
Wabt	`test/run-tests.py`
	`test/update-spec-tests.py`

PoC$^+$, we classified the PoC$^+$ tests into three categories: *Output Checking*, *Intermediate Checking* and *Self Checking*. The distribution of the three categories across the 20 projects is shown in Table 4.3.

Output Checking. This category applies to programs that process external input files or byte streams and produce observable output. During patch validation, the PoC$^+$ test executes the program with the AVR-generated patch applied, feeds it the original PoC input, and compares the actual output against the expected output stored as part of the test. The structure of *Output Checking* varies by project. As illustrated in the motivation example (Figure 4.3), PHP's PoC$^+$ tests use the PHPT format. Figure 4.4 demonstrates three other representative formats: Exiv2 employs a Python-based framework specifying the input file, command, expected return value, and stderr/stdout content; LLVM uses inline CHECK directives to verify generated intermediate representation; and libxml2 separates input XML schemas from expected error messages in distinct files. Despite these differences, all formats share a common structure of defining inputs and expected

outputs. The testcase production process for this category is straightforward: compile the program with the correct patch applied, execute it with the PoC as input to capture the expected output, and format the result according to the project's test framework conventions. We observed that several projects, including Hermes, libxml2, LLVM, PHP, and Wabt, have implemented automated scripts using similar methodologies. The locations of these production scripts are provided in Table 4.4.

Intermediate Checking. This category applies to vulnerabilities in library codebases where the original PoC is a source file (i.e., a harness) that invokes a sequence of API functions. During the patch validation stage, the PoC$^+$ test executes a modified harness that encodes expected intermediate results and determines success based on the return value. Consider the PoC $\rightarrow$ PoC$^+$ transformation for a bug in the HDF5 library, as illustrated in Figure 4.5. The original PoC consists of a sequence of API calls. The PoC$^+$ test modifies this harness by inserting `CHECK` and `VERIFY` macros to assert expected intermediate behavior at each step. Specifically, the bug-triggering call is checked by the assertion `VERIFY(ret, FAIL, ...)`. This structure ensures that the patched function correctly detects the invalid input state and returns the appropriate error code (`FAIL`) rather than crashing, thereby validating the patch. Developers produce these test cases by instrumenting API calls to capture both return values and pointer-based outputs at runtime. Since C lacks multiple return values, APIs often use pointer arguments for additional outputs. Checking logic is then inserted at an appropriate abstraction level to validate these captured values.

Self Checking. This category targets interpreter programs, such as Python, JavaScript engines, or Ruby interpreters, where vulnerabilities are triggered by executing code written in the interpreted language. The goal of PoC$^+$ construction is to transform the original PoC script into a self-validating test that explicitly verifies expected runtime behavior. Rather than merely observing whether the interpreter crashes, the PoC$^+$ test embeds assertions within the interpreted program itself to confirm that the patched interpreter properly handles previously vulnerable inputs by rais-

```
// Original PoC for hdf5 bug
space_id = H5Screate_simple(1, dims, NULL);
H5Sselect_hyperslab(space_id, ...);
H5Sset_extent_none(space_id);
// Crash/Vulnerability Triggered Here
H5Sget_select_hyper_blocklist(space_id, ...);
H5Sclose(space_id);
```

```
static void poc_plus_test(void) {
  hsize_t dims[]  = {10};
  /* ... initialization ... */
  space_id = H5Screate_simple(1, dims, NULL);
  CHECK(space_id, H5I_INVALID_HID, "H5Screate_simple");
  ret = H5Sselect_hyperslab(space_id, ...);
  CHECK(ret, FAIL, "H5Sselect_hyperslab");
  ret = H5Sset_extent_none(space_id);
  CHECK(ret, FAIL, "H5Sset_extent_none");
  ret = H5Sget_select_hyper_blocklist(space_id, ...);
  VERIFY(ret, FAIL, "H5Sget_select_hyper_blocklist");
  ret = H5Sclose(space_id);
  CHECK(ret, FAIL, "H5Sclose");
}
```

Figure 4.5: PoC$^+$ Test Example for Intermediate Checking

ing appropriate exceptions, producing specific error messages, or returning expected values. This transformation requires understanding the correct post-patch behavior and inserting corresponding exception handlers and assertions to validate the interpreter's response. Consider the example transformation for a CPython bug illustrated in Figure 4.6. The original PoC consists of simple function calls that trigger a crash in the unpatched interpreter. The PoC$^+$ test transforms this into a self-validating script by wrapping each vulnerable call in a `self.assertRaises(TypeError)` context manager, verifying both that the expected exception type is raised. This structure ensures that the patched interpreter correctly detects malformed input and raises appropriately typed exceptions. To produce self-checking test cases, developers first execute the original PoC on the patched interpreter to observe the corrected behavior, whether it raises a specific exception, returns a particular value, or outputs an error message. They may also transform the PoC to capture all possible error paths, ensuring comprehensive coverage of the fix. Developers then wrap vulnerable code paths in appropriate assertion constructs. Since these are dynamic languages, developers may also verify side effects—such as changes to global state, object attributes, or resource handles—to

```
# Original PoC for CPython sys bug
import sys
sys.remote_exec(0, None)
# PoC+ test with self-test
import _sre
with self.assertRaises(TypeError):
    sys.remote_exec(0, None)
with self.assertRaises(TypeError):
    sys.remote_exec(0, 123)
```

Figure 4.6: PoC$^+$ Test Example for Self Checking

Table 4.5: Performance of AVR tools under different validation. **init**: patches passing basic tests; **poc+**: patches also passing PoC$^+$ tests; **FDR**: false discovery rate, i.e., the fraction of initially validated patches that fail subsequent testing.

Tool	Model	init	poc+	FDR
PATCHAGENT	Sonnet-4	83.5%	50.1%	40.1% (350/873)
	GPT-4.1	76.4%	44.5%	41.7% (333/798)
SAN2PATCH	Sonnet-4	41.3%	20.7%	49.8% (215/432)
	GPT-4.1	37.9%	19.6%	48.2% (191/396)
SWE-Agent	Sonnet-4	29.0%	19.6%	32.3% (98/303)
	GPT-4.1	14.4%	8.3%	41.3% (63/150)
Overall		47.1%	27.1%	42.3% (1250/2952)

confirm the interpreter maintains correct behavior beyond just the immediate return value.

4.5 Quantifying Overestimation

In this section, we examine the extent to which conventional test suite-based validation overestimates AVR system effectiveness.

4.5.1 Methodology

We evaluated three state-of-the-art AVR systems: PATCHAGENT [11], SAN2PATCH [181], and SWE-Agent[187] on PVBENCH to quantify the extent to which test suite-based validation overestimate AVR system performance. All tools use the LLM-based approach. Since the original

SWE-Agent does not support C/C++ programs, we use its multi-language version [191]. For each tool, we conducted experiments with two different large language models: GPT-4.1 version *gpt-4.1-2025-04-14* [44] and Claude-4 Sonnet version *claude-sonnet-4-20250514* [45]. To ensure statistical reliability and account for the non-deterministic nature of LLM-based patch generation, we executed each configuration, i.e., a (AVR tool, LLM version) pair, five times on every test case, resulting in 1045 (209×5) patch attempts per each configuration for all 209 vulnerabilities, where each vulnerability will receive at most 30 (6 configurations $\times$ 5 attempts) patches.

Our experiment employs a two-stage validation framework designed to expose the limitations of conventional validation approaches. In Stage 1 (Basic Validation), generated patches are validated using the conventional approach: verifying PoC mitigation and executing the project's existing functional test suite. Patches passing both criteria are classified as "correct" under conventional validation. In Stage 2 (PoC^+ Validation), patches deemed correct in Stage 1 are further evaluated using PoC^+ tests to assess whether the patch also conforms to the semantics or developers' intention encoded in the PoC^+ test. This stage reveals false positives, i.e., patches that appear correct under basic tests but fail on PoC^+ tests.

4.5.2 General Results

Our experiment results reveal a substantial gap between conventional test suite-based validation and rigorous PoC^+ test validation. As shown in Table 4.5, PATCHAGENT with GPT-4.1 generated 798 patches that passed basic tests among 1045 executions (209 vulnerabilities $\times$ 5 attempts), achieving an initial success rate of 76.4%. Similarly, PATCHAGENT with Claude Sonnet-4 achieved an initial success rate of 83.5% However, when these seemingly correct patches were subjected to PoC^+ test validation, the success rates dropped dramatically to 44.5% and 50.1%, respectively, exposing false discovery rates (FDR) of 41.7% and 40.1%.x

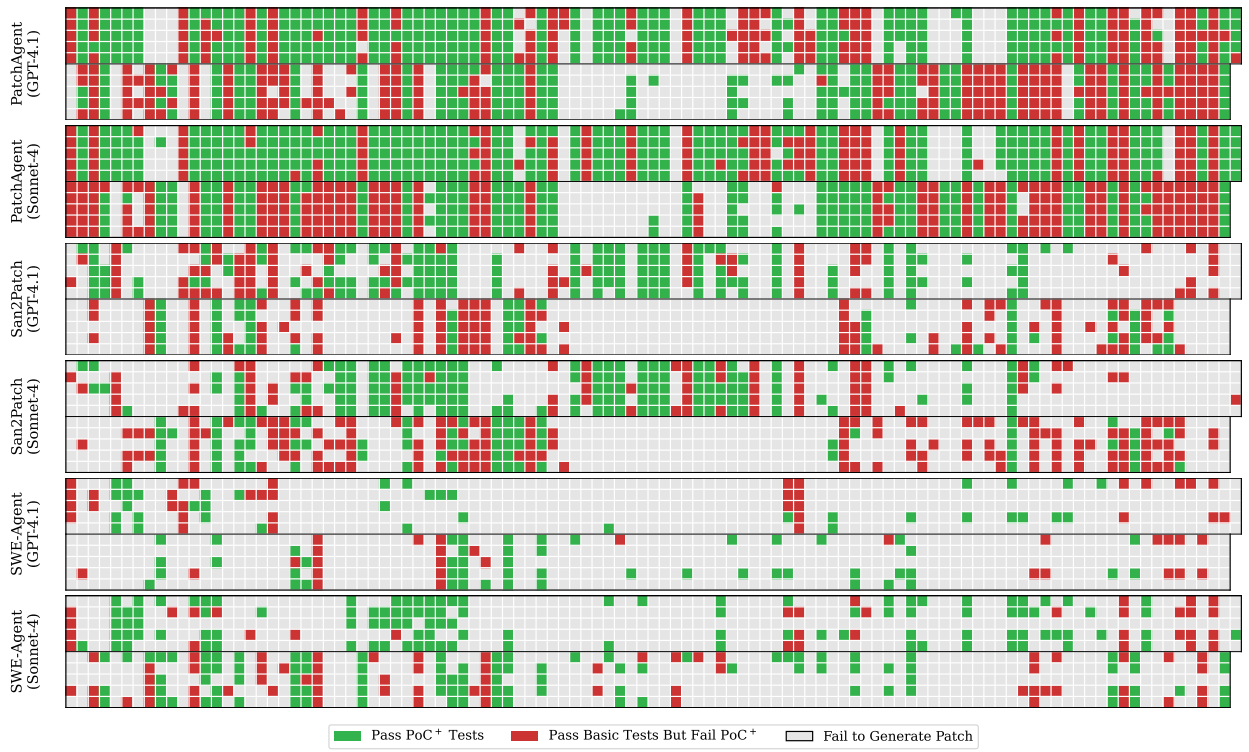

Figure 4.7: Comprehensive evaluation results across 209 vulnerability cases.

SAN2PATCH demonstrated lower overall patch generation rates but similar validation reliability issues, with initial success rates of 37.9% (GPT-4.1) and 41.3% (Sonnet-4) declining to 19.6% and 20.7% under PoC$^+$ validation, resulting in FDRs of 48.2% and 49.8%. SWE-Agent exhibited the poorest performance with initial success rates of only 14.4% (GPT-4.1) and 29.0% (Sonnet-4), further declining to 8.3% and 19.6% respectively under PoC$^+$ validation. The full results are provided in Figure 4.7.

Across all configurations, our evaluation consistently reveals an FDR around 40%, i.e., about 40% of patches that pass basic tests fail on PoC$^+$ tests. To analyze the repair outcomes for different vulnerabilities in our experiments. Figure 4.8 illustrates the distribution of vulnerabilities based on the ratio of correct patches to false positive patches generated, revealing four distinct behavioral

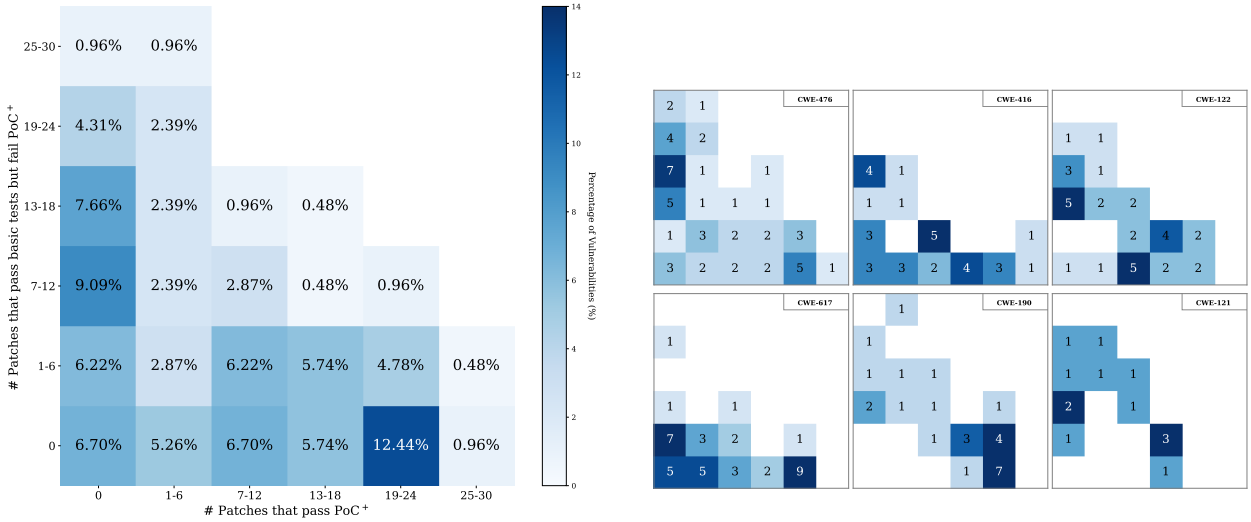

Figure 4.8: Distribution of vulnerabilities: The x-axis shows #patches passing PoC$^+$ tests, while the y-axis shows #patches passing basic tests but failing PoC$^+$ tests.

patterns:

Repair-Resistant Vulnerabilities. 6.70% of vulnerabilities cluster at the origin (no patch passes any tests), representing cases where AVR tools completely fail to generate any patches that pass basic validation. These vulnerabilities consistently challenge all tested AVR configurations, indicating limitations in current tools.

High-Success Repair Targets. 31.10% of vulnerabilities demonstrate excellent repairability, generating correct patches without producing any false positives. The distribution peaks at 12.44%, representing repair scenarios with 19-24 correct patches and zero false positives, while an additional 6.70% produce 7-12 correct patches with perfect accuracy.

False-Positive Prone Vulnerabilities: 28.24% of vulnerabilities generate exclusively false positive patches while producing zero genuinely correct patches that pass comprehensive testing. The distribution peaks at 9.09% for cases producing 7-12 false positives with no correct patches. These scenarios are particularly concerning for real-world deployment, as AVR tools consistently gener-

ate plausible but fundamentally flawed fixes, creating a dangerous illusion of successful repair that could introduce new security risks.

Mixed-Performance Vulnerabilities: 33.96% of vulnerabilities exhibit mixed repair patterns across intermediate performance regions, generating varying combinations of correct and incorrect patches in low-to-moderate quantities.

The vulnerability distribution reveals a complex, multi-modal pattern demonstrating varied AVR tool performance across real-world vulnerabilities. Rather than showing a continuous distribution, the data exhibits distinct clustering with 6.7% producing no patches, 31.10% achieving strong repair outcomes, and 28.24% generating only false positives. This clustering pattern suggests that repairability is predominantly an intrinsic characteristic of the vulnerability itself rather than a consequence of specific tool configurations or parameter choices.

To understand these clustering patterns by CWE category, we conducted detailed analysis on six prevalent ($\geq$ 10 samples) CWE types in our dataset. CWE-617 (Reachable Assertion) emerges as the most resistant vulnerability type to automated repair, likely because LLMs lack extensive training data on developer-introduced assertions compared to common memory errors. However, when patches are successfully generated for CWE-617, they demonstrate higher reliability than other vulnerability types. Among memory errors, NULL pointer dereference (CWE-476) and stack-based buffer overflow (CWE-121) both exhibit high false discovery rates. This may occur because these vulnerabilities typically have standardized repair methods, leading LLMs to generate superficial fixes—null checks for CWE-476 and boundary checks for CWE-121—rather than addressing the underlying architectural issues. Use After Free (CWE-416) and Heap-based Buffer Overflow (CWE-122) demonstrate more reliable repair outcomes. Integer overflow (CWE-190) also employs general fix methods but shows high reliability, possibly because upgrading to larger data types is a common practice in real-world development.

Table 4.6: Manual Categorization of Patches that Passed PoC$^+$ Tests Across Different AVR Tools: The table shows the distribution of four quality categories Results are compared across three tools using two LLMs .

Category	PatchAgent		San2Patch		SWE-Agent		Total
	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	
Semantic Equivalent	396 (75.72%)	339 (72.90%)	156 (71.89%)	150 (73.17%)	159 (77.56%)	66 (75.86%)	1266 (74.38%)
Performance Issue	25 (4.78%)	24 (5.16%)	1 (0.46%)	3 (1.46%)	3 (1.46%)	0 (0.00%)	56 (3.29%)
Suboptimal Repair	59 (11.28%)	61 (13.12%)	28 (12.90%)	27 (13.17%)	20 (9.76%)	13 (14.94%)	208 (12.22%)
Check Circumvention	43 (8.22%)	41 (8.82%)	32 (14.75%)	25 (12.20%)	23 (11.22%)	8 (9.20%)	172 (10.11%)
Total	**523 (100%)**	**465 (100%)**	**217 (100%)**	**205 (100%)**	**205 (100%)**	**87 (100%)**	**1702 (100%)**

4.6 Reliability of PoC$^+$ Test

In this section, we examine the reliability of PoC$^+$ tests as a validation method for evaluating patch correctness.

4.6.1 Methodology

To assess the reliability of PoC$^+$ tests and identify potential issue of using them for AVR tools evaluation, we conducted a manual review of all patches that passed PoC$^+$ tests by comparing them with developer patches. The results are presented in Table 4.6. We classified these patches into four categories based on their semantic closeness to the developer patches. Two patches are considered semantically equivalent if they satisfy the following criteria:

- Both patches target the same function
- Both patches have identical time and space complexity
- The logic implemented by both patches is functionally equivalent with no unintended side effects

For patches that do not achieve semantic equivalence, we proceed with further categorization. If we find that a patch uses ad-hoc methods to bypass security or functional checks, we classify it as an *Check Circumvention*, as such approaches are in fact incorrect implementations despite

passing all tests, including PoC$^+$. For patches with correct functionality but different performance characteristics, we evaluate whether the AVR-generated solution exhibits worse time or space complexity compared to the developer solution. Such cases are classified as having *performance issues*. Finally, for patches that are functionally correct and have same algorithm complexity with developers patch, we assess quality by examining whether the developer's patch is more obvious for correctness compared to the AVR-generated patch. We classify the AVR-generated patch as having *Suboptimal Repairs*.

We implemented an inter-rater reliability process [180], [193] where each author independently categorized the patches, and then cross-validated results and discussed any discrepancies until reaching consensus. **Our findings show that more than 70% of the patches maintain semantic equivalence with developer patches**, demonstrating the high reliability of the PoC$^+$ tests. The subsequent sections examine the remaining patch categories in detail.

4.6.2 Performance Issue

Performance issues arise when patches are functionally correct but employ repair strategies with suboptimal algorithmic complexity compared to developer solutions. To evaluate performance differences, we analyze the complexity of the extra operations required specifically for the repair, rather than comparing the overall program complexity. This approach isolates the computational overhead introduced by different repair strategies. If we find that the time or space complexity of extra operations for repairing the bug by the AVR tools is higher than the developer's, we categorize the patch as having a performance issue. It is important to note that categorizing a patch as having a performance issue does not necessarily indicate that it is better or worse than the developer's solution, as the project may not be particularly sensitive to performance considerations. This category simply identifies the difference in computational efficiency between repair approaches.

```c
static int template_clear(TemplateObject *self) {
    Py_CLEAR(self->literal);
    for (Py_ssize_t i = 0, n = Py_SIZE(self); i < n; i++)
        // BUG: items[i].literal may be uninitialized
        Py_CLEAR(self->items[i].literal);
    return 0;
}
static PyObject *sre_template_impl(..., PyObject *template) {
    // template is a list containing interleaved
    // literal strings (str or bytes) and group indices (int)
    // [literal1, group1, literal2, ..., literalN].
    Py_ssize_t n = /* Extract number of groups */;
    // Allocate array but items[].literal not initialized
    TemplateObject *self = PyObject_GC_NewVar(TemplateObject, ..., n);
    self->literal = Py_NewRef(PyList_GET_ITEM(template, 0));
    // FIX OPTION 1: Initialize all literal fields to NULL
    // This prevents crashes since Py_CLEAR safely handles NULL
    // for (Py_ssize_t i = 0; i < n; i++)
    //     self->items[i].literal = NULL;
    for (Py_ssize_t i = 0; i < n; i++) {
        Py_ssize_t index = // Extract (i+1)-th group number;
        if (index < 0) {
            // FIX OPTION 2: Set object size to track
            // how many items were successfully initialized.
            // template_clear will only clean initialized items.
            // Py_SET_SIZE(self, i);
            goto bad_template;
        }
        // Normal case: would initialize items[i].literal here...
    }
    return (PyObject*) self;
bad_template:
    PyErr_SetString(PyExc_TypeError, "invalid template");
    Py_XDECREF(self); // Triggers template_clear() cleanup
    return NULL;
}
```

Figure 4.9: Example of Performance Issue

To illustrate this category, we examine a case from CPython involving an uninitialized memory access vulnerability. Figure 4.9 demonstrates how two algorithmically distinct repair strategies can address the same bug with different time complexity for their repair operations. The issue occurs when `PyObject_GC_NewVar()` allocates memory for a `TemplateObject` but leaves the `items[i].literal` fields uninitialized. If an error condition arises during the initialization loop (such as encountering a negative index), the code jumps to the error handler, which calls `Py_XDECREF(self)` and subsequently triggers `template_clear()`. This cleanup function blindly iterates through all allocated items and calls `Py_CLEAR()` on potentially uninitial-

ized `literal` fields, causing crashes when the macro attempts to decrement reference counts on garbage memory values.

Two distinct repair strategies emerge to address this vulnerability, which differ significantly in their algorithmic complexity. The AVR-suggested fix takes a defensive programming approach by proactively initializing all `literal` fields to `NULL` immediately after allocation. This repair strategy has $O(n)$ complexity. In contrast, the developer patch employs a tracking-based strategy using `Py_SET_SIZE(self, i)` to record how many items have been successfully initialized. This repair approach has $O(1)$ complexity.

4.6.3 Suboptimal Repair

Suboptimal repair represents patches that are functionally correct and maintain the same algorithmic complexity as developer solutions, but exhibit inferior implementation quality that makes their correctness less apparent. With no intuitive justification on why code changes are made at specific locations, this type of patch eventually hurts the maintainability of the overall codebase. Patches in this category typically fall into two patterns.

First, AVR-generated patches often address vulnerabilities at later stages in the execution flow rather than preventing the underlying issue at its source. Similar to the incorrect root cause example in our motivating case (Section 4.3), developer patches typically fix vulnerabilities at creation sites where corrupted data structures are initially formed, while AVR tools apply defensive measures at usage sites where problems manifest. The prevention-oriented approach is more intuitively correct because it detects problems at their source and makes the code more self-documenting.

Second, developer solutions often encode richer semantic information and domain-specific knowledge compared to AVR-generated alternatives. To illustrate this pattern, consider a heap out-of-bounds vulnerability in PHP's class variable handling shown in Figure 4.10. The PoC of

```
@@ -729,10 +729,14 @@ void add_class_vars(...)
     }
     prop = NULL;
     if (statics && (info->flags & ACC_STATIC) != 0) {
-        prop = &ce->static_members_table[info->offset];
+        if (ce->static_members_table &&
+            info->offset < ce->static_members_count) {
+            prop = &ce->static_members_table[info->offset];
+        }
     } else if (!statics && (info->flag & ACC_STATIC) == 0) {
-        prop = &property_table[info->offset];
+        if (property_table && info->offset < ce->property_count) {
+            prop = &property_table[info->offset];
+        }
       }
       if (!prop) {
          continue;
```

(a) AVR-Generated Patch

```
@@ -724,7 +724,8 @@ void add_class_vars(...)
     if (((info->flags & ACC_PROTECTED) &&
          !check_protected(info->ce, scope)) ||
          ((info->flags & ACC_PRIVATE) &&
-          info->ce != scope)) {
+          info->ce != scope) ||
+         (info->flags & ACC_VIRTUAL)) {
          continue;
     }
     prop = NULL;
```

(b) Developer Patch

Figure 4.10: Example of Suboptimal Repair

the vulnerability defines a class with virtual members, but the handling function incorrectly forgets to handle this case, causing heap out-of-bounds access. The AVR-generated patch applies defensive bounds checking by validating array pointers and offsets before access, essentially forcing a fix through defensive programming. In contrast, developer's patch adds a single condition `(info->flags & ACC_VIRTUAL)` to exclude virtual properties from processing, demonstrating semantic understanding that virtual properties should not be handled in this context. While both patches prevent the crash, developer's solution encodes the actual business logic—virtual properties are conceptually different and require separate handling—making it obviously better despite being difficult to distinguish through automated testing. This semantic richness reflects deep un-

derstanding of PHP's object model and makes the code more maintainable.

4.6.4 Check Circumvention

Check circumvention represent patches that attempt to bypass security or functional checks rather than addressing the underlying root cause of bugs—another way of suppressing an error! These patches prioritize immediate test passage over principled problem resolution, often employing workarounds that circumvent protective mechanisms or safety validations. Such approaches fundamentally misunderstand the purpose of security checks and program invariants, treating them as obstacles to avoid rather than indicators of deeper logical issues. Our analysis identified two primary patterns of check circumvention. First, for out-of-bounds access vulnerabilities, AVR patches frequently resort to excessive memory over-allocation, attempting to prevent crashes by allocating significantly larger buffer sizes rather than determining and allocating the exact memory size the program requires or correcting the faulty indexing logic. Second, for reachable assertion failures, patches commonly either remove assertion statements entirely or artificially manipulate variables immediately before assertions to force conditions to evaluate as true, effectively disabling safety checks without understanding why the assertions were violated.

Table 4.7: Categorization of FP Patches: The table presents a breakdown of FP patches across three AVR systems using two LLMs.

Category	**PATCHAGENT**		**SAN2PATCH**		**SWE-Agent**		**Total**
	Sonnet-4	**GPT-4.1**	**Sonnet-4**	**GPT-4.1**	**Sonnet-4**	**GPT-4.1**	
Incorrect Root Cause	144 (41.14%)	118 (35.44%)	113 (52.56%)	81 (42.41%)	37 (37.76%)	22 (34.92%)	515 (41.18%)
Spec Violation	189 (54.00%)	200 (60.06%)	93 (43.26%)	97 (50.79%)	61 (62.24%)	40 (63.49%)	680 (54.38%)
Poor Code Practice	17 (4.86%)	15 (4.50%)	9 (4.19%)	13 (6.81%)	0 (0.00%)	1 (1.59%)	55 (4.40%)
Total	**350 (100%)**	**333 (100%)**	**215 (100%)**	**191 (100%)**	**98 (100%)**	**63 (100%)**	**1250 (100%)**

4.7 False Positive Analysis

We conducted a systematic analysis of all false positive cases from our PVBENCH experiments (Section 4.5.2) to understand the reasons why these patches pass the basic tests but fails PoC$^+$. In particular, we manually compared those patches with developer patches and classified them into three categories: patches that incorrectly identify the root cause of the issue, patches that violate project specifications, and patches that use poor coding practices. Table 4.7 presents the distribution of these failure categories across all tested configurations. Our analysis reveals that *Specification Violation* represents the most prevalent failure mode, accounting for approximately 55.57% of all false positive cases. This is followed by *Incorrect Root Cause* at 39.88%, while *Poor Code Practice* constitute the smallest category.

4.7.1 Incorrect Root Cause

We classify patches into this category when the generated patch modifies code in a different function from the developer patch, indicating fundamental misunderstanding of the true location of vulnerability. This represents the most severe type of analytical failure, where AVR tools fix symptoms close to the point of failure rather than addressing the underlying cause that eventually leads the vulnerability. To illustrate this failure mode, consider a NULL pointer dereference vulnerability in AST module of Python interpreter demonstrated by this PoC exploit:

```
import ast; del ast.AST._fields; t = ast.AST(arg1=123)
```

When `ast.AST._fields` is deleted and a new AST object is subsequently created with `ast.AST`, the program crashes with a NULL pointer dereference. Figure 4.11 shows how the AVR tools and developer approach this problem with fundamentally different philosophies. The AVR-generated patch applies a band-aid solution by adding a defensive NULL check in `PySequence_Contains()`

```
@@ -2234,6 +2234,10 @@ PySequence_Count(...) {
 PySequence_Contains(PyObject *seq, PyObject *ob)
 {
+    if (seq == NULL) {
+        null_error();
+        return -1;
+    }
     PySeqMethods *sqm = Py_TYPE(seq)->tp_as_sequence;
     if (sqm != NULL && sqm->sq_contains != NULL) {
```

(a) AVR-Generated Patch

```
@@ -5083,19 +5083,17 @@ ast_type_init(...) {
     PyObject *key, *value, *fields, *attributes = NULL;
-    if (PyObj_GetOptionalAttr((PyObject*)Py_TYPE(self),
-                              state->_fields, &fields) < 0) {
+    fields = PyObj_GetAttr((PyObject*)Py_TYPE(self),
+                           state->_fields);
+    if (fields == NULL)
         goto cleanup;
-    if (fields) {
-        numfields = PySequence_Size(fields);
-        if (numfields == -1)
-            goto cleanup;
-        remaining_fields = PySet_New(fields);
-    }
-    else
-        remaining_fields = PySet_New(NULL);
+    numfields = PySequence_Size(fields);
+    if (numfields == -1)
+        goto cleanup;
+    remaining_fields = PySet_New(fields);
     if (remaining_fields == NULL)
         goto cleanup;
```

(b) Developer Patch

Figure 4.11: Incorrect Root Cause Example

where the crash occurs. While this prevents the immediate crash, it fails to address why the NULL condition arose in the first place. In contrast, the human developer's patch identifies the true root cause in `ast_type_init()`, where the initialization code incorrectly uses `PyObj_GetOptAttr()` to retrieve `_fields`. By changing to `PyObject_GetAttr()`, the patch enforces stricter validation during object creation.

The developer's approach is grounded in the AST specification, which explicitly states that *"Each concrete class has an attribute `_fields` which gives the names of all child nodes"* [230]. This makes `_fields` a mandatory component of the AST object model, not an optional one. The

human patch recognizes this specification requirement and prevents invalid objects from being created, while the AVR patch merely handles the consequences of allowing such invalid objects to exist. This example demonstrates how incorrect root cause identification leads to patches that may prevent immediate crashes but fail to address the underlying design violation, potentially leaving the system vulnerable to related issues.

4.7.2 Specification Violation

We classify patches into this category when the generated patch correctly identifies the problematic code location but produces modifications that violate established software specifications, programming language standards, or documented functional requirements. These violations typically manifest as changes to input validation logic, return value semantics, or control flow that contradict to properties or contracts communicated otherwise. As demonstrated in Section 4.3 with the PHP `range()` function, the AVR-generated patch enforced strict type checking by rejecting mixed-type inputs with runtime errors, while the PHP language specification requires permissive type coercion that allows mixed numeric and string arguments to generate valid numeric ranges. This violation demonstrates how patches can on one hand fix vulnerabilities while on the other hand fundamentally breaking expected program functionality.

Patches in specification violation category demonstrate partial understanding of the vulnerability context but fail to respect the intended functionality of the target system overall. This failure mode indicates that AVR systems need improved adherence to program specifications, requiring better integration of a knowledge base that captures intended program behaviors, and potential constraint validation during patch generation.

```
@@ -661,8 +661,12 @@ getArgAccessInfo(...) {
  auto TypeSize = DL.getTypeStoreSize(Ty);
  if (!TypeSize.isScalable() && Offset) {
    int64_t Size = TypeSize.getFixedValue();
-   return ConstantRange(
-              APInt(64, *Offset, true),
-              APInt(64, *Offset + Size,
-              true));
+   if (Size > 0) {
+     int64_t End = *Offset + Size;
+     if (End > *Offset)
+       return ConstantRange(
+              APInt(64, *Offset, true),
+              APInt(64, End, true));
+   }
  }
  return std::nullopt;
};
```

(a) AVR-Generated Patch

```
@@ -661,8 +661,13 @@ getArgAccessInfo(...) {
  auto TypeSize = DL.getTypeStoreSize(Ty);
  if (!TypeSize.isScalable() && Offset) {
    int64_t Size = TypeSize.getFixedValue();
-   return ConstantRange(
-              APInt(64, *Offset, true),
-              APInt(64, *Offset + Size,
-              true));
+   APInt Low(64, *Offset, true);
+   bool Overflow;
+   APInt High = Low.sadd_ov(
+              APInt(64, Size, true),
+              Overflow);
+   if (Overflow)
+     return std::nullopt;
+   return ConstantRange(Low, High);
  }
  return std::nullopt;
};
```

(b) Developer Patch

Figure 4.12: Domain Ignorance Example

4.7.3 Poor Code Practice

We classify patches into this category when the generated patch correctly identifies the vulnerability location and maintains specification compliance but still fails PoC$^+$ due to poor code quality or violation of established coding practices. These patches demonstrate adequate analytical capabilities of LLM but reveal insufficient technical domain knowledge or adherence to project-specific design principles. This failure mode indicates that AVR systems need enhanced code generation capabilities by incorporating better understanding of platform-specific requirements, software engineering best practices, and sometimes even intricacies in compiler behaviors. We provide two examples to illustrate patches in this category. One example involves an AVR-generated patch that violates C/C++ programming standards (i.e., causing undefined behavior), which is shown in Figure 4.12. The vulnerability occurs when computing memory access ranges in the `FunctionAttrs()` optimization pass, where `*Offset + Size` can overflow and wrap around.

The AVR patch in Figure 4.12a demonstrates overflow detection by adding a simple compari-

```
1     GET_NODE(sxe, node);
2  +  if (!node) {
3  +    /* avoid null dereference */
4  +    return &EG(err_zval);
5  +  }
6     php_libxml_invalidate_node_from_doc(node->doc);
7     if (node) {
8           if (attribs) {
```

Figure 4.13: Logic Shortcuts Example

son `End > *Offset` to check for overflow. However, this approach is flawed because it relies on undefined behavior—when signed integer overflow occurs in C++, the comparison itself invokes undefined behavior, and aggressive compiler optimizations may eliminate the check entirely, assuming that signed overflow never occurs. The patch creates a false sense of security while potentially being optimized away at compile time. In contrast, the developer patch in Figure 4.12b correctly uses LLVM's `APInt::sadd_ov` method, which is specifically designed for overflow-safe arithmetic operations. This approach uses well-defined overflow detection mechanisms that cannot be optimized away by compilers, properly handling the edge case by returning `std::nullopt` when overflow is detected. The developer solution demonstrates deep understanding of both the vulnerability context and the platform-specific requirements for reliable overflow detection in optimized code.

The other example demonstrates how a AVR-generated patch disregards the control flow logic carefully maintained by developers. This example encompasses patches that introduce logically destructive code structures that break the developer's intended design patterns. As illustrated in Figure 4.13, the generated patch adds an early null check that immediately returns when `node` is null, bypassing the developer's carefully structured conditional logic that was designed to handle null cases gracefully within the existing control flow (the matching `else` block for the `if` condition at line 7). This approach creates destructive code in the later `if(node)` check, as the condition will

always be true since null values have already been filtered out by the early return, and also violates the developer's intent to maintain a unified error handling strategy throughout the function. The correct approach respects the original design by moving the `php_libxml_invalidate_node` call inside the existing `if(node)` conditional, preserving the developer's intended control flow while eliminating the vulnerability without introducing structural inconsistencies.

4.8 Implications for AVR Research

Our evaluation on PVBENCH reveals that conventional test suite-based validation methods may substantially overestimate the effectiveness of AVR tools, with over 40% of patches that pass basic tests failing to pass PoC$^+$ tests. Our further analysis of these false positive cases suggests that the primary cause is specification violations, where patches produce modifications that violate established software specifications or documented behavioral requirements. These findings suggest two main implications for AVR research.

AVR research could benefit from adopting more reliable validation methodologies during evaluation beyond simply running PoC exploits and existing functional tests. The discovery of such high false discovery rates (40%+) across all tested state-of-the-art AVR systems indicates that current validation practices may create an illusion of effectiveness. This systematic overestimation could undermine confidence in deployment decisions and suggests that some published success rates might be inflated. To address this potential gap, future AVR evaluation frameworks could incorporate multi-layered validation approaches, including developer-authored functional tests like PoC$^+$ tests, manual comparison against developer patches to assess semantic equivalence and formal verification techniques. Research might prioritize developing automated methods to generate or identify comprehensive test suites that capture true functional requirements, while AVR benchmarks could consider including more rigorous validation criteria that help ensure patches meet

both security and specification requirements for potential production deployment.

Current AVR approaches that rely primarily on codebase and vulnerability information as input may be insufficient for generating production-ready patches. Our categorical analysis in Section 4.7 reveals that modern LLM-based AVR tools often struggle to understand program specifications, API semantics, and behavioral requirements that appear essential for correct repairs. The prevalence of specification violations suggests that many critical requirements are documented in external sources rather than being easily inferable from code alone. AVR systems could benefit from incorporating information from project documentation, API specifications, coding guidelines, comments, and other textual resources that capture intended program behavior and constraints. Future AVR research might explore methods to automatically extract and leverage such documented requirements, develop techniques to integrate natural language specifications with code analysis, and create approaches that can reason about both explicit code patterns and implicit behavioral expectations described in documentation. This potential shift toward incorporating documented knowledge alongside code analysis could represent an important evolution for generating patches that respect both syntactic correctness and documented program intentions.

4.9 Discussion

Scope and Limitations. While our study provides valuable insights into the limitations of conventional test suite-based patch validation methods, several constraints limit the generalizability of our findings. First, PVBENCH focuses exclusively on C/C++ programs across 20 open-source projects, which may not represent the full spectrum of programming languages and software domains where AVR tools are applied. Different languages have varying memory safety guarantees, type systems, and testing cultures that could influence both vulnerability patterns and validation effectiveness. Second, improved patch validation methods do not directly enhance the effective-

ness of AVR tools. Although some prior work suggests that better patch validation can provide more informative feedback to improve AVR systems, our test generation approach still relies on developer-provided correct patches rather than generating tests from scratch. Consequently, our method is primarily suited for evaluation purposes rather than integration into an end-to-end repair pipeline. We leave the exploration of test generation from scratch for future work.

Automated Software Engineering. Building PVBENCH required substantial effort due to challenges such as resolving complex dependencies and aggregating information from multiple sources. This labor-intensive process consumed considerable time, limiting both dataset scale and diversity. Our experience highlights the critical need for automated software engineering. Recent advances have begun to address these challenges: CompileAgent[231] automates repository-level compilation with LLM agents, while ExecutionAgent[232] can automatically executes functional test suites. Automated dataset construction systems like SWE-smith[233] and SWE-rebench[234] have demonstrated the ability to create datasets an order of magnitude larger than manual collections.

4.10 Related Work

Patch Equivalence. While many works use test suite-based methods to evaluate the patch correctness rate of their tools, some of them [36], [181] also employ manual comparison to count the number of patches that are semantically equivalent to developer patches. We can consider these two success rates as the lower bound and upper bound of the actual patch success rate, respectively. Empirical studies have shown that approximately 25% of correct AVR patches are syntactically different but semantically equivalent to developer patches [235], highlighting the importance of semantic comparison metrics. Recent advances in automated semantic equivalence assessment combine syntactic and semantic similarity metrics with test coverage analysis [236], while sophisticated approaches leverage program invariants and pre-trained language models for

semantic reasoning about patch equivalence [237]. Research on program equivalence for adaptive vulnerability repair has formalized the patch equivalence problem using test-equivalence relations, proposing efficient algorithms for partitioning patches into equivalence classes based on runtime behavior [56]. Our work provides a better approach to automatically calculate the upper bound more precisely, while the lower bound still relies on manual review.

Formal Verification. Formal verification approaches offer mathematically rigorous alternatives to test-suite based patch validation, providing stronger guarantees about patch correctness through specification-based reasoning and constraint solving. SemFix [238] formulates repair requirements as constraints solved by SMT solvers, providing formal guarantees about patch correctness beyond test adequacy. Subsequent advances in scalable semantics-based repair using symbolic execution with Z3 SMT solver have demonstrated practical applications of formal methods to multiline vulnerability repair [239]. Contract-based repair approaches use formal specifications such as pre- and postconditions for both patch generation and validation [240], while sound and complete mutation-based vulnerability repair provides theoretical guarantees of soundness and completeness through bounded model checking and SAT/SMT solving [241]. Advanced approaches integrate formal verification throughout the repair process, using constraint solving with mathematical foundations such as Farkas lemma for template-based patch synthesis [242], while modular program verifiers [243] enable property-specific validation with formal correctness guarantees .

4.11 Conclusion

Our study suggests that current AVR evaluation practices may benefit from more comprehensive validation approaches, as we observe that over 40% of patches deemed correct by conventional test-suite methods fail when evaluated against PoC$^+$ tests across three state-of-the-art LLM-based AVR systems. Our proposed PoC$^+$ test demonstrates promising results, with over 70% of pass-

ing patches achieving semantic equivalence with developer solutions, suggesting this methodology could serve as a valuable complement to existing validation practices. These findings encourage future research to consider incorporating more rigorous quality assessment approaches and including specification information in AVR.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

The dissertation presents three major contributions to make the automated vulnerability repair (AVR) process more practical and reliable in real-world settings.

To address those vulnerabilities that can be triggered by concrete inputs (e.g., fuzzing-generated PoCs), we proposed PATCHAGENT, an AVR 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. Most importantly, PATCHAGENT successfully repaired 10 real vulnerabilities in widely-used open-source projects, with all patches accepted by the respective communities.

For vulnerabilities that have already been fixed in the mainline branch but need to be backported to older branches, we developed PORTGPT, an LLM-based agent that automates the patch backporting process. By equipping the LLM with tools for on-demand code access, Git history summarization, and autonomous patch revision based on feedback, PORTGPT effectively simulates human-like reasoning and verification during the backporting process. Our evaluation on

both existing and newly curated datasets demonstrated that PORTGPT outperforms state-of-the-art backporting tools, achieving high success rates even on complex cases. Notably, we contributed 9 backported patches generated by PORTGPT to the Linux kernel community, all of which were successfully merged.

Although existing AVR systems, especially those leveraging LLMs, have shown promising results in patching vulnerabilities, their patch validation methodologies often overlook critical aspects of correctness. To address this gap, we introduced PVBench, a benchmark designed for rigorous evaluation of patch correctness in AVR systems. Our study suggests that current AVR evaluation practices may benefit from more comprehensive validation approaches, as we observe that over 40% of patches deemed correct by conventional test suite methods fail when evaluated against PoC$^+$ tests across three state-of-the-art LLM-based AVR systems. While this finding indicates potential overestimation in current evaluation practices, it also presents an opportunity for the AVR research community to enhance validation methodologies. The systematic patterns we identify—including specification violations and incorrect root cause identification—could inform the development of more robust AVR systems. Our proposed PoC$^+$ test approach demonstrates promising results, with over 70% of passing patches achieving semantic equivalence with developer solutions, suggesting this methodology could serve as a valuable complement to existing validation practices. These findings encourage future research to consider incorporating more rigorous quality assessment approaches and include specification information in AVR.

5.2 Funture Work

The findings of this dissertation reveal several promising directions for advancing the field of automated vulnerability repair (AVR) toward greater reliability and semantic correctness. First, our evaluation of PoC$^+$ highlighted that a significant portion of LLM-generated patches fail due to a

lack of "specification awareness." Future work should explore the integration of formal program specifications, such as ACSL or JML, directly into the repair loop. By developing neuro-symbolic AVR systems, we can bridge the gap between the generative power of LLMs and the rigorous guarantees of formal verification [240], [243]. Such an approach would allow agents to reason about program invariants and contracts, potentially enabling "correct-by-construction" patch generation that reduces the current reliance on post-hoc test-based validation.

Furthermore, while PORTGPT demonstrated the efficacy of LLM agents in patch backporting, complex architectural shifts between software versions remain a challenge. Future research should investigate context-dense representations of software evolution, where agents consider not just code diffs, but also architectural metadata, mailing list discussions, and historical commit patterns. Integrating these multi-modal context sources would allow AVR agents to autonomously handle non-trivial refactorings and API migrations that currently require significant human intervention during the backporting process.

Finally, there is a clear need to move beyond functional test suites toward holistic quality assurance. Future AVR pipelines should incorporate automated code-smell detection and performance regression testing to ensure that patches adhere to the idiomatic practices and performance constraints of the target codebase [244]. We envision the development of self-evolving repair agents that learn from the feedback provided by human maintainers during the code review process. By systematically analyzing the reasons behind patch rejections—such as memory management nuances or compiler-specific behaviors—these systems could iteratively refine their reasoning capabilities in a human-in-the-loop ecosystem, ultimately leading to more robust and community-acceptable automated repairs.

REFERENCES

[1] *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/.

[2] *Metrics — CVE*, https://www.cve.org/About/Metrics.

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

[4] *syzkaller*, https://syzkaller.appspot.com/upstream.

[5] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Software*, 2021.

[6] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, 2023.

[7] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[8] C. S. Xia and L. 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.

[9] E. First, M. N. Rabe, T. Ringer, and Y. 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.

[10] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1482–1494, ISBN: 9781665457019.

[11] Z. Yu et al., "Patchagent: A practical program repair agent mimicking human expertise," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[12] Z. Li, Z. Yu, J. Song, M. Xu, Y. Luo, and D. Mu, " PORTGPT: Towards Automated Backporting Using Large Language Models," in *2026 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2026, pp. 643–662.

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

[14] *CodeQL*, https://codeql.github.com/.

[15] C. Yagemann, S. P. H. Chung, B. Saltaformaggio, and W. Lee, "Pumm: Preventing use-after-free using execution unit partitioning," in *Proceedings of 32nd USENIX Security Symposium*, 2023.

[16] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[17] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient heap memory protection through advanced metadata management and customized compiler optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7177–7193, ISBN: 978-1-939133-44-1.

[18] B. Wickman et al., "Preventing Use-After-Free attacks with fast forward allocation," in *Proceedings of 30th USENIX Security Symposium*, 2021.

[19] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "Libmpk: Software abstraction for intel memory protection keys (intel {mpk})," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[20] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[21] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "Xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[22] D. P. McKee et al., "Preventing kernel hacks with hakcs.," in *NDSS*, 2022.

[23] D. Song et al., “Sok: Sanitizing for security,” in *2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 1275–1295.

[24] J. Keller and J. Nowakowski, “Ai-powered patching: The future of automated vulnerability fixes,” Technical report, Tech. Rep., 2024.

[25] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, “Understanding automatically-generated patches through symbolic invariant differences,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019.

[26] E. C. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, “Program synthesis and semantic parsing with learned code idioms,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[27] C. E. Jimenez et al., “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.

[28] Y. Wei, C. S. Xia, and L. 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.

[29] C. S. Xia and L. Zhang, “Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt,” *arXiv preprint arXiv:2304.00385*, 2023.

[30] S. Lipp, S. Banescu, and A. 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.

[31] T. Blazytko et al., “{Aurora}: Statistical crash analysis for automated root cause explanation,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[32] Y. Park, H. Lee, J. Jung, H. Koo, and H. K. Kim, “Benzene: A practical root cause analysis system with an under-constrained state mutation,” in *2024 IEEE Symposium on Security and Privacy (SP)*, 2023.

[33] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, and M. 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.

[34] A. Kallingal Joshy, X. Chen, B. Steenhoek, and W. Le, "Validating static warnings via testing code fragments," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[35] Y. Wu, Z. Lin, Y. Chen, D. K. Le, D. Mu, and X. Xing, "Mitigating security risks in linux with KLAUS: A method for evaluating patch correctness," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 4247–4264, ISBN: 978-1-939133-37-3.

[36] 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.

[37] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 1047–1058, ISBN: 9781450327565.

[38] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

[39] S. Hong, J. Lee, J. Lee, and H. Oh, "Saver: Scalable, precise, and safe memory-error repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 271–283, ISBN: 9781450371216.

[40] Y. Xing, S. Wang, S. Sun, X. He, K. Sun, and Q. Li, "What {if} is not enough? fixing null pointer dereference with contextual check," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[41] J. Huang and K. C.-C. Chang, "Towards reasoning in large language models: A survey," *arXiv preprint arXiv:2212.10403*, 2022.

[42] *huntr*, https://huntr.com/.

[43] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Feb. 2021.

[44] OpenAI, *OpenAI API models documentation*, https://platform.openai.com/docs/models, 2024.

[45] Anthropic, *Anthropic*, https://www.anthropic.com/, 2024.

[46] *42-B3YOND-6UG: AI Cyber Security Reasoning System*, https://b3yond.org/.

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

[48] Y. Chen, M. Khandaker, and Z. Wang, "Pinpointing vulnerabilities," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017.

[49] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang, "{Freewill}: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[50] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. 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.

[51] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[52] D. Xu et al., "Racing on the negative force: Efficient vulnerability {root-cause} analysis through reinforcement learning on counterexamples," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[53] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th international conference on software engineering*, 2014.

[54] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, 2016.

[55] F. Long and M. 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.

[56] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '13, Silicon Valley, CA, USA: IEEE Press, 2013, pp. 356–366, ISBN: 9781479902156.

[57] N. Jiang, T. Lutellier, and L. 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.

[58] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[59] T. Brown et al., "Language models are few-shot learners," *Advances in neural information processing systems*, 2020.

[60] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[61] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. 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.

[62] M. Chen et al., *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374 [cs.LG].

[63] M. Jin et al., "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.

[64] K. Huang et al., "A survey on automated program repair techniques," *arXiv preprint arXiv:2303.18184*, 2023.

[65] S. B. Murtaza, A. Mccoy, Z. Ren, A. Murphy, and W. Banzhaf, "Llm fault localisation within evolutionary computation based automated program repair," in *GECCO Companion*, 2024.

[66] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," *ArXiv*, 2024.

[67] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning llms for fl-free program repair," *ArXiv*, 2024.

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

[69] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the USENIX Conference on Annual Technical Conference*, 2012.

[70] *AddressSanitizerLeakSanitizer*, https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer.

[71] T. F. Smith, M. S. Waterman, et al., "Identification of common molecular subsequences," *Journal of molecular biology*, 1981.

[72] *LangChain*, https://www.langchain.com/.

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

[74] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[75] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "{Fixreverter}: A realistic bug injection methodology for benchmarking fuzz testing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

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

[77] J. Yang et al., "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

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

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

[80] GNOME, *Libxml2: XML parsing and manipulation library*, http://xmlsoft.org/, 2024.

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

[82] B. Moolenaar, *Vim: The ubiquitous text editor*, https://www.vim.org/, 2024.

[83] *OpenAI API*, https://platform.openai.com/docs/models.

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

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

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

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

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

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

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

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

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

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

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

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

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

[97] Z. Liu et al., "Towards unveiling exploitation potential with multiple error behaviors for kernel bugs," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[98] Z. Lin et al., "Grebe: Unveiling exploitation potential for linux kernel bugs," *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[99] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[100] K. Zeng et al., "Retspill: Igniting user-controlled data to burn away linux kernel protections," *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[101] Z. Liang, X. Zou, C. Song, and Z. 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.

[102] W. Chen, X. Zou, G. Li, and Z. Qian, "{Koobe}: Towards facilitating exploit generation of kernel {out-of-bounds} write vulnerabilities," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[103] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{Fuze}: Towards facilitating exploit generation for kernel {use-after-free} vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[104] GitHub, *GitHub Actions documentation*, https://docs.github.com/en/actions, 2024.

[105] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[106] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[107] Y. Deng, C. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *International Conference on Software Engineering*, 2024.

[108] S. Ainsworth and T. 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.

[109] M. Erdős, S. Ainsworth, and T. 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.

[110] B. Lee et al., "Preventing use-after-free with dangling pointers nullification.," in *NDSS*, 2015.

[111] G. J. Duck and R. H. 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.

[112] S. Nagarakatte, J. Zhao, M. M. Martin, and S. 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.

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

[114] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," *arXiv preprint arXiv:2404.05427*, 2024.

[115] Y. Sun et al., "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," *arXiv preprint arXiv:2401.16185*, 2024.

[116] Y. Ding et al., "Vulnerability detection with code language models: How far are we?" *arXiv preprint arXiv:2403.18624*, 2024.

[117] R. Pan et al., "Lost in translation: A study of bugs introduced by large language models while translating code," *arXiv preprint arXiv:2308.03109*, 2023.

[118] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[119] *Linux kernel release lifecycle*, https://www.kernel.org/category/releases.html.

[120] *Node.js release lifecycle*, https://nodejs.org/en/about/previous-releases.

[121] X. Tan, Y. Zhang, J. Cao, K. Sun, M. Zhang, and M. Yang, "Understanding the practice of security patch management across multiple branches in oss projects," in *Proceedings of the ACM Web Conference 2022 (WWW)*, 2022.

[122] D. Chakroborti, K. A. Schneider, and C. K. Roy, "Backports: Change types, challenges and strategies," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022.

[123] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 633–645.

[124] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing oss patch backporting with semantics," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2366–2380.

[125] S. Wu et al., "Mystique: Automated vulnerability patch porting with semantic and syntactic-enhanced llm," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 130–152, 2025.

[126] S. Pan, Y. Wang, Z. Liu, X. Hu, X. Xia, and S. Li, "Automating zero-shot patch porting for hard forks," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 363–375.

[127] W. X. Zhao et al., "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[128] *GPT-4o*, https://platform.openai.com/docs/models#gpt-4o.

[129] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 560–573.

[130] N. Wadhwa et al., "Core: Resolving code quality issues using llms," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 789–811, 2024.

[131] N. Mündler, M. Müller, J. He, and M. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," *Advances in Neural Information Processing Systems*, vol. 37, pp. 81 857–81 887, 2024.

[132] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated patch transplantation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–36, 2020.

[133] *CVE-2022-32250 Mainline Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=520778042ccca019f3ffa136dd0ca565c486cedd.

[134] *CVE-2022-32250 patch for mainline failed to apply in 5.4-stable*, https://lore.kernel.org/stable/1654262711245226@kroah.com/.

[135] *Detailed Description of Unified Format Patch*, https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html.

[136] M. Levy, A. Jacoby, and Y. Goldberg, "Same task, more tokens: The impact of input length on the reasoning performance of large language models," *arXiv preprint arXiv:2402.14848*, 2024.

[137] *CVE-2023-52752 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d328c09ee9f15ee5a26431f5aad7c9239fa85e62.

[138] *universal-ctags*, https://github.com/universal-ctags/ctags.

[139] *GPT-4*, https://openai.com/gpt-5/.

[140] *Gemini 2.5 Flash*, https://ai.google.dev/gemini-api/docs/models#gemini-2.5-flash.

[141] *Introducing Meta Llama 3: The most capable openly available LLM to date*, https://ai.meta.com/blog/meta-llama-3/.

[142] DeepSeek-AI, *Deepseek-v3 technical report*, 2024. arXiv: 2412.19437 [cs.CL].

[143] Y. Shi et al., "Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 1993–2010, ISBN: 978-1-939133-31-1.

[144] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1630–1644.

[145] F. Yan et al., "Berkeley function calling leaderboard," 2024.

[146] *CVE-2024-46743 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b739dffa5d570b411d4bdf4bb9b8dfd6b7d72305.

[147] *CVE-2023-51779 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2e07e8348ea454615e268222ae3fc240421be768.

[148] *CVE-2024-0565 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eec04ea119691e65227a97ce53c0da6b9b74b0b7.

[149] *CVE-2024-26922 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6fef2d4c00b5b8561ad68dd2b68173f5c6af1e75.

[150] *CVE-2024-43863 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58337100721f3cc0c7424a18730e4f39844934f.

[151] *CVE-2023-24023 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=04a342cc49a8522e99c9b3346371c329d841dcd2.

[152] *CVE-2024-24860 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da9065caa594d19b26e1a030fd0cc27bd365d685.

[153] *CVE-2024-25742 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e70316d17f6ab49a6038ffd115397fd68f8c7be8.

[154] *CVE-2024-41066 Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0983d288caf984de0202c66641577b739caad561.

[155] *Linux vuls.git*, https://git.kernel.org/pub/scm/linux/security/vulns.git/.

[156] *CVE-2025-21816 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=82ac6adbbb2aad14548a71d5e2e37f4964a15e38.

[157] *CVE-2024-26618 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=f6421555dbd7cb3d4d70b69f33f998aaeca1e3b5.

[158] *CVE-2024-26807 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=2c914aac9522f6e93822c18dff233d3e92399c81.

[159] *CVE-2024-36903 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=a05c1ede50e9656f0752e523c7b54f3a3489e9a8.

[160] *CVE-2024-36927 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=55bf541e018b76b3750cb6c6ea18c46e1ac5562e.

[161] *CVE-2024-53209 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=b7fd784d7c6a1bd927a23e0d06f09a776ee3889b.

[162] *CVE-2024-56758 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=36679fab54fa7bcffafd469e2c474c1fc4beaee0.

[163] *CVE-2024-57945 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=92f08673d3f1893191323572f60e3c62f2e57c2f.

[164] *CVE-2025-21645 Merged Patch in Linux for 6.1-stable*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=v6.1.152&id=92f08673d3f1893191323572f60e3c62f2e57c2f.

[165] *Cve-2024-26889 patch revert*, https://lore.kernel.org/stable/2024111506372 2.766718123@linuxfoundation.org/.

[166] *Ubuntu*, https://ubuntu.com/.

[167] *CVE-2021-4197 Mainline Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1756d7994ad85c2479af6ae5a9750b92324685af.

[168] *CVE-2021-4197 Stable Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a70bcf9ed08f3628a9324f054b0e041697b26853.

[169] *CVE-2019-16232 Mainline Patch in Linux*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7da413a18583baaf35dd4a8eb414fa410367d7f2.

[170] *CVE-2022-41720 original patch in golang master branch*, https://github.com/golang/go/commit/7dc9fcb13de7bb20b11f6a526865545cc9142c2c.

[171] *CVE-2022-41720 backported patch in golang v1.19 branch*, https://github.com/golang/go/commit/d80340177116c079fb2ad681dd4aaa4bdc27b770.

[172] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: A t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 935–947, ISBN: 9781450394130.

[173] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.

[174] D. Guo et al., "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[175] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[176] N. Alshahwan et al., "Automated unit test improvement using large language models at meta," *arXiv preprint arXiv:2402.09171*, 2024.

[177] N. Alshahwan, M. Harman, A. Marginean, S. Sengupta, and E. Wang, "Assured llm-based software engineering (keynote paper)," in *2nd. ICSE workshop on Interoperability and Robustness of Neural Software Engineering (InteNSE)(Lisbon, Portugal). To appear*, 2024.

[178] Y. Hu et al., *Sok: Automated vulnerability repair: Methods, tools, and assessments*, 2025. arXiv: 2506.11697 [cs.SE].

[179] Y. Li, F. hossain shezan, B. wei, G. Wang, and Y. Tian, *Sok: Towards effective automated vulnerability repair*, 2025. arXiv: 2501.18820 [cs.CR].

[180] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, *Appatch: Automated adaptive prompting large language models for real-world software vulnerability patching*, 2025. arXiv: 2408.13597 [cs.CR].

[181] Y. Kim, S. Shin, H. Kim, and J. Yoon, "Logs in, patches out: Automated vulnerability repair via {tree-of-thought}{llm} analysis," in *34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA: USENIX Association, 2025, pp. 4401–4419.

[182] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 103–111, ISBN: 9798400706851.

[183] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proceedings of the*

IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE '24, Lisbon, Portugal: Association for Computing Machinery, 2024, ISBN: 9798400702174.

[184] Y. Zhang et al., *Fixing security vulnerabilities with ai in oss-fuzz*, 2024. arXiv: 2411.03346 [cs.CR].

[185] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, pp. 1–38, 2021.

[186] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 524–535.

[187] J. Yang et al., *Swe-agent: Agent-computer interfaces enable automated software engineering*, 2024. arXiv: 2405.15793 [cs.SE].

[188] R. Shariffdeen, C. S. Timperley, Y. Noller, C. Le Goues, and A. Roychoudhury, "Vulnerability repair via concolic execution and code mutations," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–27, 2025.

[189] H. Zheng, I. Shumailov, T. Fan, A. Hall, and M. Payer, *Fixing 7,400 bugs for 1$: Cheap crash-site program repair*, 2025. arXiv: 2505.13103 [cs.SE].

[190] Y. Zhang, X. Gao, G. J. Duck, and A. Roychoudhury, "Program vulnerability repair via inductive inference," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, Virtual, South Korea: Association for Computing Machinery, 2022, pp. 691–702, ISBN: 9781450393799.

[191] D. Zan et al., *Multi-swe-bench: A multilingual benchmark for issue resolving*, 2025. arXiv: 2504.02605 [cs.SE].

[192] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 633–645, ISBN: 9781450384599.

[193] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing oss patch backporting with semantics," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 2366–2380, ISBN: 9798400700507.

[194] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 539–554.

[195] X. Xu, Y. Sui, H. Yan, and J. Xue, "Vfix: Value-flow-guided precise program repair for null pointer dereferences," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 512–523.

[196] A. Mathai et al., *Crashfixer: A crash resolution agent for the linux kernel*, 2025. arXiv: 2504.20412 [cs.SE].

[197] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 390–405, ISBN: 9781450383912.

[198] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 8–18, ISBN: 9781450362245.

[199] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, *Squad: 100,000+ questions for machine comprehension of text*, 2016. arXiv: 1606.05250 [cs.CL].

[200] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02, Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

[201] S. Ren et al., *Codebleu: A method for automatic evaluation of code synthesis*, 2020. arXiv: 2009.10297 [cs.SE].

[202] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 532–543, ISBN: 9781450336758.

[203] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software*

Engineering, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 789–799, ISBN: 9781450356381.

[204] S. Wang et al., "Automated patch correctness assessment: How far are we?" In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20, Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 968–980, ISBN: 9781450367684.

[205] J. Petke, M. Martinez, M. Kechagia, A. Aleti, and F. Sarro, "The patch overfitting problem in automated program repair: Practical magnitude and a baseline for realistic benchmarking," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 452–456, ISBN: 9798400706585.

[206] Y. Ouyang, J. Yang, and L. Zhang, "Benchmarking automated program repair: An extensive study on both real-world and artificial bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, Vienna, Austria: Association for Computing Machinery, 2024, pp. 440–452, ISBN: 9798400706127.

[207] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, " LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks," in *2024 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 862–880.

[208] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 955–963, ISBN: 9781450355728.

[209] KidFlo, *Range(9.9, '0') causes segmentation fault*, https://github.com/php/php-src/issues/13094, 2024.

[210] Q. Mi and J. Keung, "An empirical analysis of reopened bugs based on open source projects," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16, Limerick, Ireland: Association for Computing Machinery, 2016, ISBN: 9781450336918.

[211] PHP Documentation Group, *PHP:range Manual*. https://www.php.net/manual/en/function.range.php, 2024.

[212] PHP Group, *PHP: Hypertext preprocessor*, https://www.php.net/, 2024.

[213] Python Software Foundation, *CPython: The reference implementation of Python*, https://github.com/python/cpython, 2024.

[214] The HDF Group, *HDF5: High-performance data management suite*, https://www.hdfgroup.org/solutions/hdf5/, 2024.

[215] LLVM Project, *LLVM: The LLVM compiler infrastructure*, https://llvm.org/, 2024.

[216] Exiv2, *Exiv2: Image metadata library and tools*, https://exiv2.org/, 2024.

[217] Google, *V8 javascript engine*, https://v8.dev/, 2024.

[218] WebAssembly, *WABT: The WebAssembly binary toolkit*, https://github.com/WebAssembly/wabt, 2024.

[219] facebook, *Hermes: JavaScript engine for React Native*, https://hermesengine.dev/, 2024.

[220] Unicode Consortium, *ICU: International components for Unicode*, https://icu.unicode.org/, 2024.

[221] PcapPlusPlus Contributors, *PcapPlusPlus: Network packet processing library*, https://pcapplusplus.github.io/, 2024.

[222] F. Bellard, *QuickJS: Small JavaScript engine*, https://bellard.org/quickjs/, 2024.

[223] LibTIFF Contributors, *LibTIFF: TIFF library and utilities*, http://libtiff.org/, 2024.

[224] mruby Contributors, *Mruby: Lightweight Ruby implementation*, https://mruby.org/, 2024.

[225] JasPer, *JasPer: Image processing library*, https://jasper-software.github.io/jasper/, 2024.

[226] jqlang, *Jq: Command-line JSON processor*, https://jqlang.github.io/jq/, 2024.

[227] simdjson, *Simdjson: Parsing gigabytes of JSON per second*, https://simdjson.org/, 2024.

[228] Samtools Contributors, *HTSlib: High-throughput sequencing data library*, https://www.htslib.org/, 2024.

[229] Wireshark Foundation, *Wireshark: Network protocol analyzer*, https://www.wireshark.org/, 2024.

[230] Python Software Foundation, *Ast — Abstract Syntax Trees*, https://docs.python.org/3/library/ast.html, 2024.

[231] L. Hu et al., *Compileagent: Automated real-world repo-level compilation with tool-integrated llm-based agent system*, 2025. arXiv: 2505.04254 [cs.SE].

[232] I. Bouzenia and M. Pradel, "You name it, i run it: An llm agent to execute tests of arbitrary projects," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1054–1076, 2025.

[233] J. Yang et al., *Swe-smith: Scaling data for software engineering agents*, 2025. arXiv: 2504.21798 [cs.SE].

[234] I. Badertdinov et al., *Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents*, 2025. arXiv: 2505.20411 [cs.SE].

[235] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, *How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques*, 2019. arXiv: 1906.03447 [cs.SE].

[236] A. Ghanbari and A. (Marcus, "Shibboleth: Hybrid patch correctness assessment in automated program repair," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758.

[237] T. Le-Cong et al., "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3411–3429, 2023.

[238] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 772–781, ISBN: 9781467330763.

[239] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 691–701, ISBN: 9781450339001.

[240] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, May 2014.

[241] B.-C. Rothenberg and O. Grumberg, "Sound and complete mutation-based program repair," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds., Cham: Springer International Publishing, 2016, pp. 593–611, ISBN: 978-3-319-48989-6.

[242] T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin, "Automatic program repair using formal verification and expression templates," in *Verification, Model Checking, and Abstract Interpretation*, C. Enea and R. Piskac, Eds., Cham: Springer International Publishing, 2019, pp. 70–91, ISBN: 978-3-030-11245-5.

[243] F. Logozzo and T. Ball, "Modular and verified automatic program repair," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 133–146, 2012.

[244] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.