Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz

Yuntong Zhang
zhang.yuntong@u.nus.edu
National University of Singapore
Singapore

Jiawei Wang
wangjw@comp.nus.edu.sg
National University of Singapore
Singapore

Dominic Berzin
dominic.berzin@u.nus.edu
National University of Singapore
Singapore

Martin Mirchev
martin.mirchev@sonarsource.com
SonarSource
Singapore

Abhik Roychoudhury
abhik@nus.edu.sg
National University of Singapore
Singapore

Abstract

Critical open source software systems undergo significant validation in the form of lengthy fuzz campaigns. The fuzz campaigns typically conduct a biased random search over the domain of program inputs, to find inputs which crash the software system. Such fuzzing is useful to enhance the security of software systems in general since even closed source software may use open-source components. Hence testing open source software is of paramount importance. Currently OSS-Fuzz is the most significant and widely used infra-structure for continuous validation of open source systems. Unfortunately even though OSS-Fuzz has identified more than 13,000 vulnerabilities across 1000 or more software projects, the detected vulnerabilities may remain unpatched, as vulnerability fixing is often manual in practice.

In this work, we explore the use of Large Language Model (LLM) agents for automated vulnerability remediation. To our knowledge, this is the first systematic study of LLM-assisted security patching on OSS-Fuzz. We adapt the AutoCodeRover agent, which typically fixes bugs from issue descriptions, to the security domain. Instead of issue text, our agent extracts vulnerability-relevant code elements through the execution of the exploit input, and augments patch generation with static typing information. We evaluate our agent in two settings. On a benchmark of historical vulnerabilities detected by OSS-Fuzz, our agent generates plausible patches for 61% to 72% of the cases. We then conduct the first evaluation of LLM agents on real-world, unpatched vulnerabilities reported by OSS-Fuzz. In this setting, the agent performs comparably to its benchmark results. Moreover, several agent-generated patches have already been merged into widely used open-source projects. These results demonstrate both the practicality of automated vulnerability remediation with LLM agents, and the feasibility of an end-to-end software protection cycle from detection to repair.

CCS Concepts

• **Software and its engineering → Automatic programming**; **Software evolution**; **Software maintenance tools**; • **Security and privacy → Software security engineering**.

Keywords

Software Security, Program Repair, AI Agents, OSS-Fuzz

ACM Reference Format:
Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, and Abhik Roychoudhury. 2026. Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786583.3786880

1 Introduction

Security vulnerabilities are one of the major threats to modern software systems. Once exploited by malicious attackers, security vulnerabilities can cause significant damage to the software and its users, incurring financial loss, data breaches, and more. In 2023, 30,927 new Common Vulnerabilities and Exposures (CVEs) are recorded by the National Vulnerability Database (NVD), and half of these vulnerabilities were classified as high or critical severity [39]. The number of new CVEs has increased by 17% compared to the previous year, underscoring the accelerated pace of vulnerability detection and the critical need for timely remediation. The recent advancement in automatic programming with generative AI could further exacerbate the security issues, since some parts of the application code could come from Large Language Models (LLMs) with little security assurance.

To safeguard the software systems, researchers and practitioners have made advances in both vulnerability detection and remediation. To detect security vulnerabilities before they are discovered/exploited by attackers, various techniques from static analysis [15, 38] to fuzzing [35, 47] have been developed and also adopted in the industry. Static analysis techniques can be applied to detect a wide range of vulnerabilities. However, they are known to report false-positive warnings since they are often based on abstraction and conservative approximation of the program semantics [25]. Fuzzing, on the other hand, employs a biased random search in the program's input space and dynamically executes the program. The dynamic nature of fuzzing ensures that a reported bug is a true positive. Fuzzing has been employed by major software companies to continuously scan for vulnerabilities in their development process [7, 10]. Google's OSS-Fuzz, announced in 2016, provides continuous fuzzing for various core open-source software [2]. As of May 2025, OSS-Fuzz has identified over 13,000 vulnerabilities across 1,000 projects [33].

While vulnerability detection techniques like fuzzing have shown to be both mature and effective, detection is only the first step in comprehensive software protection. A detected bug should be patched as soon as possible to reduce the time of exposure and the risk of being exploited. A previous study in 2021 has shown that the median time-to-fix (i.e. time from bug reporting to patch verification) to be 5.3 days for bugs detected by OSS-Fuzz [11], and 10% of the reported bugs are not fixed within the 90-day disclosure deadline. The rising number of detected vulnerabilities in recent years may require developers to invest even more time and effort in manually patching them. There is an urgent need for automated vulnerability remediation in continuous fuzzing pipelines to both ease the developers' workload and minimize the window of exposure.

Recent advancements in generative AI and LLM agents have shown promise in autonomous vulnerability remediation in programs [37, 44, 45, 49]. These LLM agents are designed for general software engineering tasks, including bug fixing and feature development. They operate in real-world scenarios where tasks are described by users in natural language. Using the task description and the software codebase as inputs, the agents generate code modification suggestions to fulfill the specified requirements. Since repairing security vulnerabilities is a specialized software engineering task, we hypothesize that with appropriate adaptation, general-purpose LLM agents for software engineering can be repurposed for this task. These repurposed agents can potentially be integrated into existing vulnerability detection pipelines such as fuzzing, where they can provide the remediation after detection and complete the software protection cycle.

CodeRover-S. In this paper, we present a large scale real-world study on using LLM agents for security vulnerability repair. To enhance the realism of our effort, we use as dataset the OSS-Fuzz projects, which seek to enhance the state of practice of open source security [33]. We repurposed the LLM agent AutoCodeRover [37, 49] to repair security vulnerabilities, and implemented a version named CodeRover-S (i.e. AutoCodeRover for security). With the vulnerability report and an exploit input produced by a fuzzing campaign, CodeRover-S autonomously generates patches that fix the detected vulnerability. In the process of adapting LLM agents for vulnerability repair, we identified that one challenge was the insufficient information contained in the auto-generated vulnerability report. Unlike human-written issue report for general software engineering tasks, vulnerability reports are often auto-generated by the fuzzer and only contain information like the bug type and crash stacktrace. To enrich the context for vulnerability repair, we extract dynamic call graph information from the exploit input found by fuzzing, which is then used to augment the report generated by the fuzzer. In addition, we perform a type-based analysis at the program locations identified as faulty by the agent, and use the additional type information to augment the patch generation process.

We evaluate the efficacy of CodeRover-S in both historical vulnerabilities and unpatched vulnerabilities reported by OSS-Fuzz. Each detected vulnerability comes with an exploit input that resulted in a crash from sanitizers (e.g., AddressSanitizer [19], MemorySanitizer [20]), and the crash report generated by the sanitizer. On 588 real-world historical vulnerabilities from a previously curated dataset [31], CodeRover-S can repair 61.1% of these vulnerabilities by resolving the crash from the exploit input, using o3-mini as the backend LLM. When switching to gemini-2.5-flash as the backend LLM, the efficacy further improved to 71.8%. Our ablation study shows that the newly introduced features such as the dynamic and static analysis augmentation improved the efficacy of CodeRover-S by 5.1 percentage points. Beyond the benchmark, we also evaluated CodeRover-S in repairing unpatched real-world vulnerabilities reported by OSS-Fuzz in open-source projects. We curated a dataset of 45 vulnerabilities that have been reported and disclosed to the public, but not yet patched by the open-source project maintainers. CodeRover-S generated plausible patches for 73.3% of the unpatched vulnerabilities, demonstrating that its efficacy can generalize from benchmarks to real-world deployment. We further conducted manual correctness analysis of the patches and submitted several patches to their corresponding project repositories. At the time of writing, five patches have already been merged into four different projects.

In summary, our contributions are as follows:

- We explore the feasibility of adapting general-purpose LLM programming agents for the repair of security vulnerabilities. We integrate call graph information and type-based analysis to provide richer context for LLM agent-based vulnerability repair, resulting in improved patch quality. Our approach is implemented as a new agent CodeRover-S which is specialized for security vulnerability repair.
- We conduct an empirical study on the use of LLM agents to repair real-world security vulnerabilities identified by the industrial fuzzing service OSS-Fuzz. Our findings on existing benchmarks indicate that leveraging LLM agents is a promising approach for security vulnerability remediation.
- We conduct the first study of using LLM agents to repair unpatched vulnerabilities reported by OSS-Fuzz. Our results indicate that LLM agents' efficacy in producing plausible patches generalizes beyond benchmark instances to unpatched vulnerabilities in the wild. Furthermore, we demonstrate the potential of integrating agent-generated patches into software projects and deploying agent-based vulnerability remediation.

2 Background

We discuss background on the OSS-Fuzz project and LLM agents for software engineering.

2.1 Overview of OSS-Fuzz project

Fuzz testing [6] is a popular method for detecting software security vulnerabilities, via a biased random search over the domain of program inputs. Launched by Google in 2016, OSS-Fuzz is an open-source initiative designed to continuously detect security vulnerabilities across more than 1,250 open-source software projects. The participating projects provide a fuzzing harness to test specific API functions. OSS-Fuzz monitors the reliability of software projects' repositories by continuously testing them with a wide range of fuzzers (e.g., AFL++ [12], libfuzzer [35], Honggfuzz [21]) and sanitizers (e.g., AddressSanitizer [19] and UndefinedBehaviorSanitizer [41]). It automatically reports any crashes identified by the fuzzers and periodically verifies whether the project has resolved the reported issues. As of May 2025, the OSS-Fuzz cluster has discovered over 13,000 bugs across all projects. On average, OSS-Fuzz has reported 22 bugs for each participating project, with some projects such as *ffmpeg* having over 400 bugs reported. In this work, we focus on vulnerability remediation for C/C++ vulnerabilities detected by OSS-Fuzz.

2.2 LLM Agents for Software Engineering

Recent advances in Large Language Models' (LLMs) context windows have significantly improved their ability to process complex text sequences. This enhancement, combined with their capacity for task planning, has led to the development of agent-based systems designed to tackle a broad spectrum of problems. One area where notable success was demonstrated is software engineering. Here, an agentic system is provided with a natural language description of a task, such as issue descriptions in software repositories like GitHub. The issue can describe a bug or new features to be added to a codebase. To solve the issue, the LLM can invoke external tools, allowing it to interact with the environment and gather more data before presenting a solution in the form of a patch. These tools encompass actions fundamental to software engineering, such as Abstract Syntax Tree (AST) search, e.g., getting a function or class definition, file system navigation, and executing commands such as compiling the project or running the test suite. By integrating such tools, the agent can analyze the codebase, invoke tools to gather additional information about the failure, and make modifications while keeping track of the original task. Examples of LLM agents for software engineering include AutoCodeRover [49], SWE-Agent [45], RepairAgent [5], and Agentless [44]. Although many software engineering agents have been proposed, few studies have investigated their adaptation for security vulnerability repair. Furthermore, to the best of our knowledge, no prior work has conducted a large-scale, systematic study of LLM agents for remediating vulnerabilities reported by industrial bug detection services such as OSS-Fuzz.

Significance. Repairing vulnerabilities detected by fuzzing is vital for enhancing software security and reliability, as evidenced by the efforts from both software engineering research [8, 14, 48] and industry [28]. According to a recent study by Mei et al. [31], the number of vulnerabilities identified by OSS-Fuzz is growing steadily despite the gap between reproducible vulnerabilities and their fixes, posing a significant security risk. Furthermore, the rising number of unpatched vulnerabilities over time implies that some vulnerabilities might not receive immediate attention. Therefore, it is essential to propose reliable solutions for vulnerability remediation.

3 CodeRover-S

To study whether LLM agents for general software engineering tasks can be specialized for vulnerability remediation, we adapted the open-source agent AutoCodeRover for security vulnerability repair. In this section, we first provide an overview of AutoCodeRover, and subsequently discuss how we repurposed it for vulnerability repair.

3.1 Agent for Issue Resolution

AutoCodeRover [49] is an LLM agent designed for software engineering tasks like bug fixing and feature addition. It aims to resolve software engineering issues in a realistic setup, where only a natural-language description of the issue/requirement is available. One such setup is GitHub issues, in which users submit bug reports or feature addition requests to a software project.

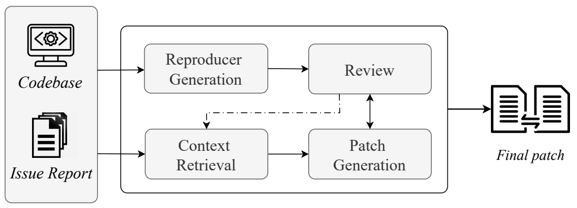
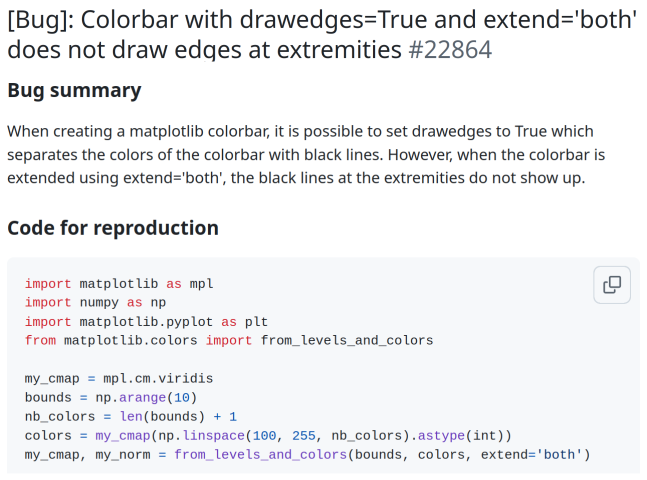


Figure 1: Workflow of AutoCodeRover for issue resolution.

Figure 1 illustrates the workflow of AutoCodeRover in resolving GitHub issues. Given a codebase *C* and a natural-language (NL) issue report *R*, AutoCodeRover autonomously produces a patch that aims to resolve the issue described in *R*. From the issue report *R*, AutoCodeRover begins the main loop with *context retrieval* and *reproducer generation*. Since an issue typically only contains NL descriptions and no executable test to reproduce the issue, AutoCodeRover first attempts to generate a candidate reproducer test for the given issue. This reproducer test serves as an additional specification for the patch generation later on. Other than the reproducer test, AutoCodeRover also starts the *context retrieval* stage from the issue report *R*. The goal of context retrieval is to extract code snippets relevant to the issue *R* from a large codebase, enabling the LLM to better understand the issue in relation to the code. AutoCodeRover performs context retrieval by designing a set of program structure-aware *search tools* (such as `search_class(...)`, `search_method_in_class(...)`), and allowing the LLM to interact with a local codebase through these tools. For example, given the example issue shown in Figure 2a, the LLM would likely invoke the tool `search_class("Colorbar")` to obtain more context about this class. Upon receiving this tool invocation, the backend of AutoCodeRover searches for the actual code/signature of the class `Colorbar` from an Abstract Syntax Tree (AST) representation of the codebase, and returns the code/signature back to the LLM. This process of tool invocation and code context collection happens iteratively, until the LLM deems that the current code context is sufficient for understanding the issue. At the end of the context retrieval stage, the LLM decides on a few *buggy locations* from the current code context. These buggy locations are provided to a *patch generation* module to craft candidate patches that aim to resolve the issue.

After a candidate patch is generated, AutoCodeRover attempts to examine whether it resolves the issue in a *review* module. If the patch is deemed to resolve the issue, the workflow ends with it being the final patch. Otherwise, a natural-language "suggestion" on how to improve the current patch is sent back to the patch generation module to iteratively improve the patch. A natural way to decide whether a patch resolves the issue is to execute the reproducer test on the patched program. The review module in AutoCodeRover takes into consideration the generated candidate patch, the reproducer test, and the issue descriptions to determine whether the patch successfully resolves the issue. The candidate patch is then subject to iterative refinement between the patch generation and



[Bug]: Colorbar with drawedges=True and extend='both' does not draw edges at extremities #22864

Bug summary

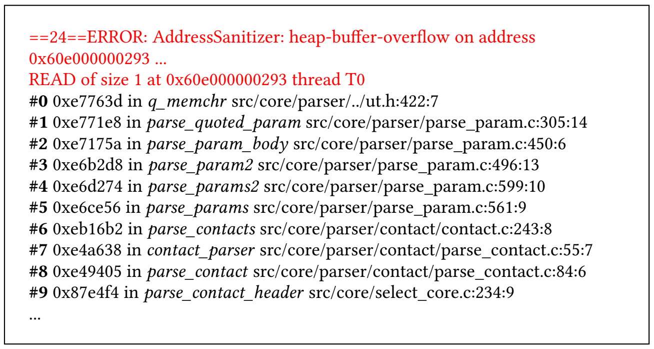
When creating a matplotlib colorbar, it is possible to set drawedges to True which separates the colors of the colorbar with black lines. However, when the colorbar is extended using extend='both', the black lines at the extremities do not show up.

Code for reproduction

```
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import from_levels_and_colors

my_cmap = mpl.cm.viridis
bounds = np.arange(10)
nb_colors = len(bounds) + 1
colors = my_cmap(np.linspace(100, 255, nb_colors).astype(int))
my_cmap, my_norm = from_levels_and_colors(bounds, colors, extend='both')
```

(a) An example GitHub issue.[1]



```
==24==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x60e000000293 ...
READ of size 1 at 0x60e000000293 thread T0
#0 0xe7763d in q_memchr src/core/parser/../ut.h:422:7
#1 0xe771e8 in parse_quoted_param src/core/parser/parse_param.c:305:14
#2 0xe7175a in parse_param_body src/core/parser/parse_param.c:450:6
#3 0xe6b2d8 in parse_param2 src/core/parser/parse_param.c:496:13
#4 0xe6d274 in parse_params2 src/core/parser/parse_param.c:599:10
#5 0xe6ce56 in parse_params src/core/parser/parse_param.c:561:9
#6 0xeb16b2 in parse_contacts src/core/parser/contact/contact.c:243:8
#7 0xe4a638 in contact_parser src/core/parser/contact/parse_contact.c:55:7
#8 0xe49405 in parse_contact src/core/parser/contact/parse_contact.c:84:6
#9 0x87e4f4 in parse_contact_header src/core/select_core.c:234:9
...
```

(b) Example of sanitizer report for Kamailio-38065[2]**.**

Figure 2: Examples of GitHub issue and sanitizer report.

the review module. If no acceptable patches were generated after several rounds of review, AutoCodeRover goes back to the context retrieval stage to re-discover buggy locations and a new set of patches. Upon reaching a pre-defined numebr of rounds, AutoCodeRover selects the most promising patch generated so far and returns it as the final output.

3.2 Agent for Security Vulnerability Repair

The AutoCodeRover workflow presented in Section 3.1 is designed for resolving software engineering issues with natural language descriptions. We next discuss the adaptation of AutoCodeRover into the context of repairing vulnerabilities detected by fuzzing campaigns such as in OSS-Fuzz. This adaptation results in an LLM agent for security vulnerability repair, which we call CodeRover-S.

We observe various differences between resolving GitHub issues and repairing vulnerabilities detected by fuzzers. Firstly, sanitizer reports produced by fuzzers contains less elaboration and natural language descriptions of the bug. Figure 2b shows an example of sanitizer-generated report. These auto-generated reports contain the error type and a stack trace when executing the exploit input, but does not contain natural language elaboration of the root cause and context of this bug. In contrast, GitHub issue reports (as shown in Figure 2a) are typically human-written and contain descriptions on relevant program components and additional details of the issue. As a result, GitHub issue reports usually contain more diverse information for LLM agent to start exploring the relevant components of the software, while sanitizer reports focus more on a specific crash. This difference highlights the need for providing additional context to LLM agents when repairing vulnerabilities found by fuzzing.

In addition, vulnerabilities found by fuzzing are always accompanied by a Proof-of-Vulnerability exploit input. In contrast, although GitHub issues may contain steps for reproduction in the description, these steps are often not executable out-of-box. Since fuzzing always provides reproducible exploit input, an LLM agent can employ more extensive *retries* in generating candidate patches, using the exploit input as an oracle to validate the candidate patches.

Leveraging these differences, we propose CodeRover-S, an LLM agent built on top of AutoCodeRover but tailored for security vulnerability repair. Figure 3 presents the workflow of CodeRover-S. Compared to AutoCodeRover, CodeRover-S generates *additional program context* beyond the sanitizer report, so that more relevant program locations can be explored by the agent. We generate additional program context through a combination of static and dynamic analysis, which respectively provide static typing information and dynamic call graphs to the LLM. Moreover, since there is an exploit input available, CodeRover-S directly leverages it as an oracle for validating candidate patches during the Patch Review stage, instead of attempting to generate a new reproducer input. After the Patch Review stage, if no candidate patches pass the exploit input oracle, an iteration of agent run concludes. In CodeRover-S, we construct an *iteration feedback* that summarizes the locations and patches explored in the current iteration, and provide this feedback to the next iteration. CodeRover-S will output a final patch if the patch passes the oracle in any iteration, or output the best patch so far (based on heuristics such as whether the patch can be compiled).

In the remainder of this section, we elaborate on the novel features of CodeRover-S in greater detail.

3.2.1 *Dynamic Call Graph.*

As shown in Figure 2b, the sanitizer report records the stack trace when the program crashes due to the manifestation of the bug. However, the crash stack trace is only a small part of the entire execution and may not serve as a good starting point for the LLM agent to gather context. For example, the bug Kamailio-38065 shown in Figure 2b was patched by the developer with the changes shown in Figure 4. The developer patch modifies the `skip_name` function, which does not appear in the stack trace but is invoked in other parts of the execution. It could be challenging for the agent to use the sanitizer report as the starting point of context retrieval and navigate to this function in the codebase. To address this challenge, we take advantage of the available exploit input, and generate a dynamic call graph from the execution of the exploit input. This dynamic call graph is used to augment the sanitizer report and provides more contextual information for the agent to navigate the codebase.

[1]Issue #22864 from the matplotlib project. https://github.com/matplotlib/matplotlib/issues/22864
[2]Bug #38065 from the Kamalio project, detected by OSS-Fuzz. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=38065

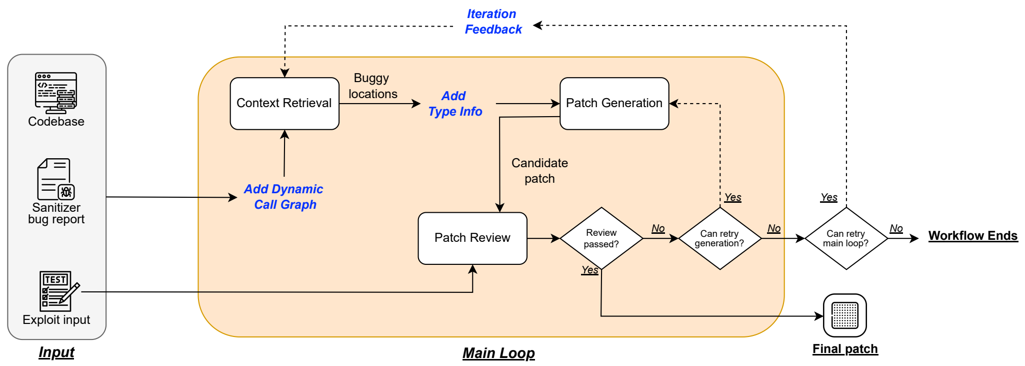


Figure 3: Workflow of CodeRover-S for repairing security vulnerabilities.

```
index 8c6ebdd6bb..345167022f 100644
--- a/src/core/parser/contact/contact.c
+++ b/src/core/parser/contact/contact.c
@@ -147,10 +147,10 @@ static inline int skip_name(str* _s)
        return 0;
      }
- if (*p == ':') {
+ if (*p == ':' || *p == ';') {
      if (last_wsp) {
-          _s->s = last_wsp;
           _s->len -= last_wsp - _s->s + 1;
+          _s->s = last_wsp;
      }
      return 0;
  }
```

Figure 4: The developer's patch for fixing Kamailio-38065.[3]

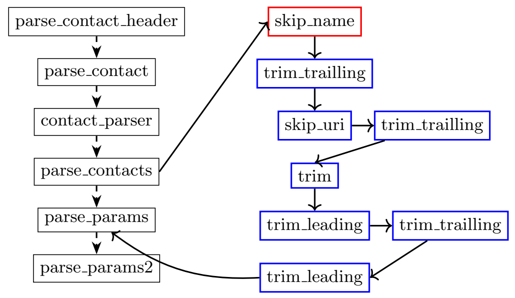


Figure 5: An example of generated dynamical call graph. The dashed lines (on the left) represent the order of function calls on stack trace and the solid lines augment them to show the actual dynamical call graph. The red colored function call is the fix location selected by the developer.

To construct this dynamic call graph, we instrument the buggy program during compile time to insert hooks at every function entry and exit points. These hooks record the memory addresses of the functions, as well as the calling relationships between callers and callees. The instrumented program is then executed with the exploit input to trigger the vulnerability. During the execution, the function entry/exit hooks log an edge list comprising pairs of function call addresses. Following this procedure, we map the memory addresses to their original function names and source code locations (e.g. filename and line number). In practice, we utilize *addr2line* [16], *gdb* [17], and *nm* [18] to accomplish such mappings.

Figure 5 shows an example of the constructed dynamic call graph for the bug Kamailo-38065. The red and blue-colored function calls display the part of the dynamic call graph beyond the crash stack trace. These function calls serve as additional starting points for the agent to explore the codebase. To make the call graph available to the LLM, we concatenate the list of additional function calls to the sanitizer bug report as "other functions executed by the bug-triggering input". The additional list of function calls enriches the auto-generated bug report from sanitizers, and provides more context for code retrieval in CodeRover-S.

3.2.2 Type-assisted Patching. The output of the context retrieval stage in CodeRover-S is a list of buggy locations (e.g. functions). With the code snippets at the buggy program locations, the patch generation stage attempts to craft patches that fix the vulnerability. However, a generated patch may not always be compilable. This is because the code of the buggy function itself may not contain the necessary patch ingredient. For example, if the type definition of a struct variable s is not within the buggy function, the LLM may hallucinate some field names of s and use those names in the patch, which will make the patch not compilable. A straightforward solution is to provide the entire file content around the buggy functions to the LLM. However, code files can be large in real-world C/C++ projects, and may not fit in the context window of the LLM. Even if the entire file fits within the context window, the relevant type definitions might be absent, as they could be defined in separate header files.

To ensure the relevant context is present for patch generation, we introduce a type-assisted patch generation prompt that includes all existing variables and their types in the scope of the buggy function. To craft such a prompt, we parse the C/C++ source files to capture critical language constructs, such as structs, classes, typedefs, and enums. Using this information, we construct file stubs to document the relationships between types, variables, and function definitions. These file summaries also encapsulate function signatures without any implementation details, so that a concise

[3]https://github.com/kamailio/kamailio/commit/20db418f1e35f31d7a90d7cabbd22ae989b7266c

To ensure the patch is compilable, please use only existing variables at the specified bug locations. Here's a list of available variables and their types:

```
variables in method: parse_param_body
Variables declarations:
- name: p , type:  param_t*
       typedef: param_t original_type:struct param ...
- name: _s , type:  str*
       typedef: str original_type:* json_key
       ...
- name: separator , type:  char
- name: _c , type:  pclass_t
       typedef: pclass_t original_type:enum pclass ...
```

When writing your patch, make sure to:
1. Use variables in a way that's consistent with their types.
2. Do not introduce imaginary variables that do not exist within the existing code snippet or the provided context.
Write a patch for the vulnerability, based on the relevant code context. First explain the reasoning, and then write the actual patch.
When writing the patch, remember the following:
- You don't have to modify every location - just make the necessary changes.
- Other than the vulnerability to fix, your patch should preserve the program functionality as much as possible. ...

Figure 6: Augmented repair prompt using type information.

summary on the relevant types can be provided to the LLM. These specialized file summaries represent essential contextual knowledge before patching. Our design enhances contextual understanding prior to patching stage, thereby reducing compilation-related errors and improving patch quality. An example patch generation prompt for the example vulnerability Kamailio-38065 is shown in Figure 6.

3.2.3 Iteration Feedback. CodeRover-S enters a retry loop if no plausible patch was generated in the first iteration. Instead of starting a retry from a fresh state, we integrated feedback from the previous iteration before starting a retry. This is to encourage the agent to explore diverse program locations and patches, so that it does not repeatedly revisiting the same locations when retrying. At the end of each iteration, we collect all the buggy locations from the previous unsuccessful repair attempts and provide a summary to the next iteration. Specifically, we prepare a prompt that contains the used file names, function names, and unsuccessful patches, and explicitly encourage the agent to explore alternative locations and fix strategies at the beginning of the next iteration.

4 Evaluation

To study the effectiveness of automated tools in real-world vulnerability remediation, we evaluate various systems on real vulnerabilities detected by OSS-Fuzz. We aim to examine the efficacy of LLM agents and learning-based vulnerability repair systems under a realistic vulnerability repair scenario. Specifically, we would like to answer the following research questions:

- RQ1: What is the efficacy of CodeRover-S in repairing real-world security vulnerabilities compared to other tools?
- RQ2: How do the new components in CodeRover-S affect its overall efficacy?
- RQ3: Can CodeRover-S repair new vulnerabilities found by fuzzing that have not been patched before?

Benchmark in RQ1,2. In RQ1 and RQ2, we utilize the recently introduced reproducible benchmark dataset, ARVO [31]. The ARVO dataset contains 5,001 C/C++ vulnerabilities detected by OSS-Fuzz across 273 projects, and each vulnerability comes with an environment to rebuild the buggy project and a bug-triggering exploit input. Since a large number of vulnerabilities have been discovered by OSS-Fuzz in the past years and are contained in the ARVO dataset, we randomly sampled a subset of 588 bugs with 99% confidence level and margin of error of 5%. If any of the buggy programs took too long to be compiled (i.e., more than 15 minutes), we randomly sample new bugs to replace them. This is to ensure the builds during the repair/validation process can finish within a reasonable amount of time. The final set of 588 vulnerabilities from ARVO is used as our dataset in RQ1-2. Figure 7 presents the distribution of vulnerabilities in our dataset based on their Common Weakness Enumeration (CWE) type. The most common CWE types are buffer overflows (CWE-121 heap-based overflow and CWE-122 stack-based overflow), segmentation faults (CWE-476), use-after-free (CWE-416), and use of uninitialized value (CWE-457).

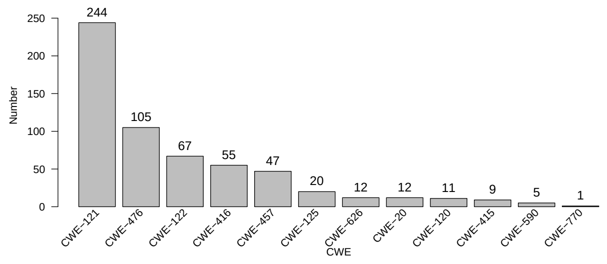


Figure 7: Vulnerability distribution in our dataset by CWE types.

Metric. We evaluate the final patch generated by a tool by applying it to the buggy program, compiling the patched program, and executing the exploit input on the patched program. We then classify the patches into the following categories: (1) No Patch (**NP**): A patch is not generated by the tool. (2) Compilation error (**CE**): A patch is generated, but caused compilation errors when building the program. (3) Implausible (**IP**): The generated patch compiles without errors, but the original exploit input still triggers the vulnerability. (4) Plausible (**P**): The patch compiles successfully, and the original exploit input can no longer trigger the bug.

Baseline tools. We compare CodeRover-S with two baseline tools:

(1) Agentless [44]: Agentless is a widely used LLM-based system designed for program repair tasks. Agentless employs a fixed three-phase workflow of localization, repair, and patch validation. The localization phase happens on multiple granularities, such as classes, methods, and file names. The repair phase generates multiple patches at the identified fix locations and then a patch validation phase is invoked to choose the final patch. Agentless was originally designed for Python codebases. To use Agentless for the C/C++ projects in OSS-Fuzz, we have

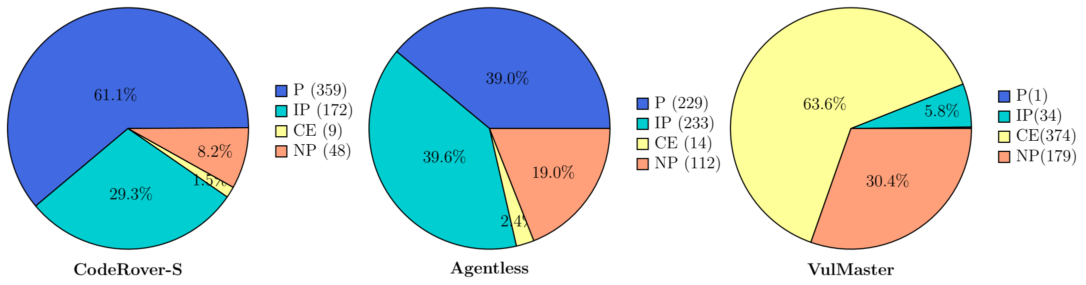


Figure 8: Comparison of results from CodeRover-S, Agentless and VulMaster. Results of CodeRover-S and Agentless are obtained with o3-mini as the backend LLM. Results of CodeRover-S with gemini-2.5-flash will be discussed in Section 4.2.

re-implemented the project structure generation, class/function parsing, and edit location mapping for C/C++. For patch validation/selection, since there is an exploit input available for each vulnerability, we use the following rule to select the final patch: *A plausible patch is strictly preferred to implausible one, and an implausible patch is strictly preferred to the one with compilation error.* If there are multiple plausible patches, the first one is chosen as the final patch.

(2) VulMaster [51]: VulMaster is a learning-based vulnerability repair tool. VulMaster finetunes a CodeT5 model using vulnerability repair data such as CWE identifiers, vulnerability descriptions, exemplar repairs and the vulnerable program fragment to be repaired. Learning-based vulnerability repair tools like VulMaster usually assume patch locations (e.g., statement/expression) to be given. To use VulMaster in a realistic vulnerability repair scenario, we employ the standard Spectrum-based Fault Localization (SBFL) to obtain a few candidate fix locations, and run VulMaster over these locations. This SBFL process takes in the exploit input as the failing test, and other non-crashing inputs generated from the OSS-Fuzz fuzzing campaign as the passing tests, and computes a list of suspicious locations using the Ochiai metric [1]. We use the top-five suspicious locations (line-level) from SBFL as the fix locations, and use VulMaster to generate a patch at each of these locations. Since the final trained model of VulMaster was not publicly released, we follow VulMaster authors' suggestions to fine-tune a CodeT5 checkpoint using the same training dataset in the original paper. Since there are multiple patches produced (one at each location), we select the final patch by following the same process as that in Agentless.

For each vulnerability, the inputs to CodeRover-S and Agentless are:

(1) An exploit input that triggers the vulnerability.
(2) The sanitizer-generated bug report for this vulnerability.

For VulMaster, we follow its original setup [51] and give it the following inputs:

(1) CWE identifier extracted from the sanitizer-generated report.
(2) A few exemplar fixes for the corresponding CWE type, extracted from the CVEFixes dataset [4].
(3) The function containing the fix location identified from SBFL.

For agent systems (CodeRover-S and Agentless) used in our evaluation, we used the OpenAI `o3-mini` as the backend LLM in RQ1. For CodeRover-S, we set the maximum number of feedback iterations to be three.

4.1 RQ1: Repair Efficacy

In this RQ, we evaluate the efficacy of CodeRover-S, Agentless, and VulMaster in generating plausible patches on the ARVO dataset.

Results. The pie charts in Figure 8 provide a comparative analysis of the patches for the three systems. All three charts categorize patches based on the aforementioned criteria, which are plausible (P), implausible (IP), compilation errors (CE), and No Patch (NP). As shown in the left figure, CodeRover-S generated plausible patches for 61.1% of the vulnerabilities, showing that it can plausibly fix a large proportion of vulnerabilities with o3-mini as the backend LLM. For the other 29.3% of the vulnerabilities, CodeRover-S generated compilable but implausible patches. The non-compilable patches account for a low percentage of the dataset (1.5%), suggesting that the agent can generated compilable patches to a large extent if it is given sufficient code context and type information. For the remaining 8.2%, CodeRover-S did not generate a patch due to errors in the code search or other exceptions happened during the execution.

In comparison, Agentless generated plausible patches for fewer bugs (39.0% versus 61.1% with CodeRover-S). This lower plausibility rate corresponds to higher rates of implausible patches (39.6% versus 29.3% with CodeRover-S) and non-compilable patches (19.0% versus 8.2%). Furthermore, we examine to what extent plausible patches generated from both tools overlap. As seen in Figure 9, the overlap between CodeRover-S and Agentless is significant, with most of the patches (84.3%) found by Agentless also present in that of CodeRover-S. Overall, we observe that CodeRover-S achieves better efficacy than Agentless in repairing security vulnerabilities. To some extent, this is expected since CodeRover-S is targeted for security patching while Agentless was designed for program repair in general.

Unlike the agent systems that generated plausible patches for around 39-61% of the vulnerabilities, VulMaster could not generate plausible patches for most of the vulnerabilities. The results from VulMaster demonstrate that 63.6% of the vulnerabilities have only non-compilable patches, the highest of all tools. In addition, 30.4% of the vulnerabilities do not have a patch that can be applied

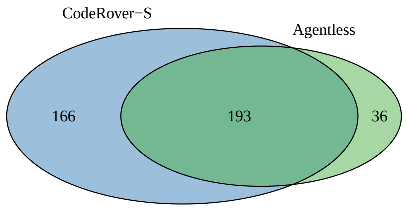


Figure 9: Venn diagram of plausible patches produced by CodeRover-S and Agentless.

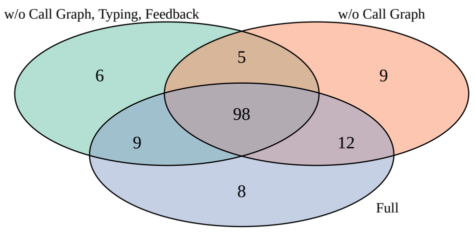


Figure 10: Venn diagram of plausible patches from the three configurations in the ablation study.

to the program. Only 5.8% of the vulnerabilities have an implausible patch. Lastly, there is a single plausible patch, which is a function call removal. This suggests that learning-based repair tools such as VulMaster is significantly less capable of generating plausible patches for vulnerabilities detected by OSS-Fuzz, compared to LLM-based agents.

4.2 RQ2: Analysis of CodeRover-S Results

In this section, we provide further analysis of patches generated by CodeRover-S.

Ablation Study. To assess the contribution of the newly introduced features in CodeRover-S to its overall efficacy, we conducted an ablation study on the features described in Section 3.2. We evaluated two ablation configurations:

(1) *w/o Call Graph*: Disables dynamic call graph generation (Section 3.2.1) in CodeRover-S.
(2) *w/o Call Graph, Typing, Feedback*: Disables dynamic call graph generation (Section 3.2.1), type-assisted patching (Section 3.2.2), and iteration feedback (Section 3.2.3). This configuration removes all specialized features, reducing CodeRover-S to an agent comparable to AutoCodeRover.

We ran these configurations of CodeRover-S on a randomly selected 30% subset of the dataset used in RQ1 to reduce cost. In addition to evaluating the effect of the new features, we also aimed to examine whether the efficacy of CodeRover-S generalizes to different backend LLMs. To this end, we performed the ablation experiments with gemini-2.5-flash and compared the results against o3-mini in RQ1.

The results of the ablation study are presented in Table 1. Comparing to the original CodeRover-S, removing all specialized features (c.f. Row 'w/o Call Graph,Typing,Feedback') reduces the plausible rate from 71.8% to 66.7%. Removing these features also increased the proportion of non-compilable patches generated by the agent (from 1.1% to 5.1%). When only the call graph component is removed (c.f. Row 'w/o Call Graph'), the proportion of 'No Patch' increases (7.9% -> 9.6%), whereas the proportion of 'Implausible' patches decreased (19.2% -> 17.5%). This result suggests that the call graph primarily helps the agent generate candidate patches by identifying more potential program locations to modify, but not necessarily makes the patches plausible. Features such as typing information and feedback play a more critical role in improving the quality of generated patches, thereby increasing the plausible rate. In addition, as shown in Figure 10, each configuration in the ablation study uniquely addresses a number of vulnerabilities. Furthermore, switching from o3-mini (released in January 2025) to gemini-2.5-flash (released in June 2025) improves the plausibility rate from 61.6% to 71.8%. As more advanced models are released, the prospects for agent-based vulnerability remediation become increasingly promising.

Patch Correctness. We have so far examined the plausibility of patches generated by the agent. However, although a plausible patch may prevent the program from crashing with the exploit input, it is not necessarily correct, as it may not generalize to other inputs. This issue is known as the *patch overfitting* problem [40] in program repair. To evaluate the correctness of the plausible patches, we performed a manual inspection of a large sample. We use two criteria in our manual inspection: (1) whether the generated patch is *semantically equivalent* to the patch written by developers, and (2) whether we think the patch correctly fixes the vulnerability, regardless of equivalence to the developer's patch. For inspection, we consider all plausible patches generated by CodeRover-S (gemini-2.5-flash) in RQ2. We excluded vulnerabilities without a reliably labeled developer patch, as these lack a clear ground truth for assessing semantic equivalence. In total, we manually inspected 88 vulnerabilities with agent-generated plausible patches.

Our manual inspection confirmed that 45.5% (40/88) of the plausible patches were correct. Agent-generated patches can be incorrect for several reasons. Firstly, some patches only focus on fixing the specific scenario demonstrated by the exploit input, but missed other edge cases such as strings consisting solely of whitespace. Secondly, the agent-generated patches can introduce unnecessary program logic changes that are unrelated to the vulnerability. We further compared the plausible patches against the developer patches, and identified 14.8% (13/88) of the agent-generated patches were semantically equivalent to the developer patch. Non-equivalent but correct cases typically occur when the agent-generated patch resolves the vulnerability directly, whereas the developer patch fixes the issue by rejecting invalid or problematic inputs earlier in the execution. A vulnerability can usually be fixed in multiple ways – a patch that is not semantically equivalent to the developer patch can still be correct, as shown from our manual inspection.

4.3 RQ3: Repairing Unpatched Vulnerabilities

In addition to evaluating CodeRover-S on repairing known vulnerabilities from the ARVO dataset, we further evaluated its effectiveness in repairing unpatched vulnerabilities. Unpatched vulnerabilities refer to those detected by OSS-Fuzz, reported to project

Table 1: Results of Ablation Study. The main results are obtained with gemini-2.5-flash as the backend LLM. Results with o3-mini on the same set of vulnerabilities are shown as reference.

Tool Configuration	LLM	Plausible	Implausible	Compilation Error	No Patch
CodeRover-S- *w/o Call Graph, Typing, Feedback*	gemini-2.5-flash	66.7%	17.5%	5.1%	10.7%
CodeRover-S- *w/o Call Graph*	gemini-2.5-flash	70.1%	17.5%	2.8%	9.6%
CodeRover-S (Full)	gemini-2.5-flash	71.8%	19.2%	1.1%	7.9%
CodeRover-S (Full)	o3-mini	61.6%	32.6%	1.7%	4.1%

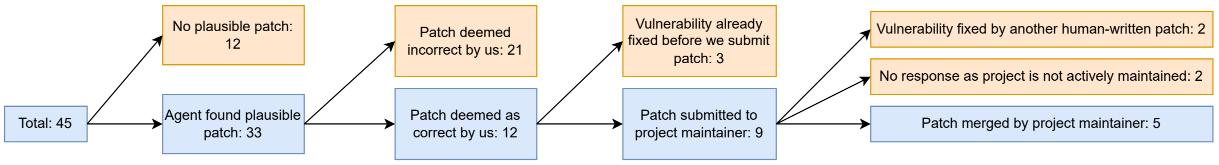

Total: 45
No plausible patch: 12
Agent found plausible patch: 33
Patch deemed incorrect by us: 21
Patch deemed as correct by us: 12
Vulnerability already fixed before we submit patch: 3
Patch submitted to project maintainer: 9
Vulnerability fixed by another human-written patch: 2
No response as project is not actively maintained: 2
Patch merged by project maintainer: 5

Figure 11: Results of CodeRover-S on 45 unpatched vulnerabilities detected by OSS-Fuzz.

maintainers, publicly disclosed, but not yet patched. To construct this dataset, we collected issues from the OSS-Fuzz issue tracker [22] which were disclosed between January and April 2025 that remained unpatched at the time of our experiment. We then excluded vulnerabilities that (1) could not be reproduced in our environment, and (2) belong to the projects not implemented in C/C++. Finally, we obtained a dataset of 45 unpatched vulnerabilities across 28 projects for RQ3. We applied CodeRover-S on this dataset of unpatched vulnerabilities to generated plausible patches, and conducted manual inspection similar to that in RQ2 to determine whether the plausible patches are correct. Because these vulnerabilities had not yet been patched, we submitted agent-generated patches that we judged to be correct to the corresponding project repositories, enabling remediation and helping to reduce the window of exposure.

Results. Figure 11 summarizes the results of this study. CodeRover-S produced plausible patches for 33/45 (73.3%) of the vulnerabilities. This high plausibility rate indicates that the efficacy of agents like CodeRover-S can extend beyond benchmark settings and generalize to real-world deployment. We manually inspected all plausible patches and identified 12 of them as correct. These patches were suitable for submission to the corresponding project repositories for resolving the fuzzer-detected vulnerabilities. At the time of patch submission, three vulnerabilities had already been patched independently (i.e., patched between the time of dataset collection to patch submission), thus we excluded them from the submissions. We submitted the remaining nine patches to their corresponding GitHub or GitLab project repositories by opening a Pull Request (PR). In each PR, we explicitly disclosed that the patches were generated by an LLM agent, and they aim to address vulnerabilities detected by OSS-Fuzz. Of the nine submitted patches, five have already been merged by project maintainers into four different projects: Assimp [30], Fluent Bit [3], LibTIFF [36], and ReadStat [43]. The remaining four patches were not merged for the following reasons: (1) two vulnerabilities were independently fixed by human-written PRs submitted by other developers, or (2) the maintainers had not yet processed the PRs (with the master branch most recently updated in December 2024). The successful acceptance and integration of these agent-generated patches demonstrate their high quality and highlight the strong potential of LLM agents for automated vulnerability remediation.

Reducing manual effort. While the process from fuzzer reports to agent-generated patch submission is largely automated, assessing the correctness of the plausible patches still remains a manual task. This step is essential to prevent developers from spending time reviewing a large number of incorrect patches. To reduce manual efforts in assessing plausible patches, we further investigated whether an *LLM-as-a-judge* [24] approach could be employed to automatically assess the correctness of plausible patches. We note that the aim of this study is to understand the feasibility of such an approach and to highlight future research directions, rather than to propose a sophisticated LLM-as-a-judge method.

Our study was conducted as follows. For each vulnerability and its plausible patch, we constructed a context consisting of:

- The sanitizer report.
- The plausible patch.
- File contents of all files modified by the plausible patch.
- 50 lines of surrounding code for each line that appeared in the crash stack trace.

We then provide an LLM with this context and the instructions describing the vulnerability repair scenario and the differences between plausible and correct patches. The LLM is tasked to produce a binary decision on whether the patch is correct and its reasoning. We used our manual inspection results as the ground truth and evaluated Sonnet-4 and GPT-5 (with different reasoning efforts) with this LLM-as-a-judge approach.

The results are presented in Table 2. Overall, Sonnet-4 outperforms GPT-5 in terms of F1-score (0.73 versus 0.52-0.57) and accuracy (0.73 versus 0.61-0.67). However, both models exhibit relatively low precision, ranging from 0.47 to 0.57. This indicates that several incorrect patches would be wrongly classified as correct when using LLM as a judge. Despite this low precision, Sonnet-4 demonstrates promising potential: it correctly identified all the true positive patches (TP=12), and mainly suffers from false positives. There is potential for improvement through fine-tuning the LLM or

Table 2: Performance of LLM-as-a-judge in determining patch correctness.

LLM	TP	FP	TN	FN	F1	Prec.	Recall	Acc.
GPT-5 (low)	7	8	13	5	0.52	0.47	0.58	0.61
GPT-5 (medium)	7	6	15	5	0.56	0.54	0.58	0.67
GPT-5 (high)	8	8	13	4	0.57	0.50	0.67	0.64
Sonnet-4	12	9	12	0	0.73	0.57	1.00	0.73

integrating test-based patch correctness assessment methods [42] to reduce false positives. In summary, this study reveals that there is a substantial gap in using LLMs as judges to automatically distinguish correct patches from plausible but incorrect ones, highlighting a promising direction for future research.

5 Related Works

Automated Program Repair (APR) techniques [23] seek to repair faulty programs by automatically generating patches for developers. In this section, we discuss APR for security vulnerabilities based on program analysis, machine learning, and LLMs, respectively.

5.1 Program Analysis based Approach

A wide range of analysis-based approaches have been explored to repair software vulnerabilities [14, 26, 48]. SenX [26] repairs program vulnerabilities using pre-defined safety properties and techniques such as access range analysis and loop cloning. ExtractFix [14] employs crash-free constraints to bootstrap the repair process. It first infers crash-free constraints for the given vulnerability, which are then propagated to the fix location and used to synthesize patches. VulnFix [48] employs a counter-example guided inductive inference approach to construct patch invariants at potential fix locations, and generates patches based on the invariants. Analysis-based vulnerability repair approaches often ground patch generation based on properties or constraints, which gives guarantees to the generated patches. However, the analysis involved can be resource-intensive and requires complex environmental setups [29].

5.2 Machine Learning based Approach

Machine learning (ML) based APR formulates the repair problem as neural machine translation (NMT) which translates buggy code to fixed code. General-purpose ML-based APR [9, 27] trains a model with sequence-to-sequence learning on datasets consisting pairs of buggy and fixed programs. Beyond general-purpose repair tools, several ML-based approaches have been proposed for repairing security vulnerabilities. VRepair [8] is a vulnerability repair tool that is based on transfer learning. It is first trained on a large bug fix corpus, and then finetuned on a smaller vulnerability fix dataset. Instead of training a model and adapting it to vulnerability repair with transfer learning, VulRepair [13] finetunes a pre-trained CodeT5 model for repairing vulnerabilities. VulMaster [51] is a recently proposed tool that integrates diverse information such as code structure and expert knowledge for vulnerability repair, and it is studied in our work.

5.3 LLM-based Approach

Our work is closely related to research on LLM-based vulnerability repair. Pearce et al. [34] investigate the use of code LLMs for zero-shot vulnerability repair, showing that LLMs can generate fixes when given a carefully constructed prompt. Beyond the zero-shot setting, various prompting strategies have been explored for vulnerability repair. APPATCH [32] employs adaptive prompting and vulnerability semantics reasoning to suggest bug fixes. San2Patch [29] utilizes Tree of Thought prompting to explore multiple locations and patches at each step, enabling a diverse search of potential solutions. Concurrent with our work, PatchAgent [46] and WilliamT [50] propose agentic approaches for vulnerability repair. PatchAgent [46] integrates various interaction optimizations into an LLM agent to mimic human expertise in vulnerability repair. WilliamT [50] introduces a template-guided patch generation approach using LLM to repair vulnerabilities at the crash site instead of the root cause. In contrast, our work focuses on adapting an existing agent designed for issue resolution to vulnerability repair by incorporating additional program analysis, such as dynamic call graphs and type information.

6 Perspectives

Automated program repair technologies typically involve search, semantic reasoning and machine learning to automatically rectify bugs and vulnerabilities. Typically such techniques are driven by a given test-suite as an indicator of correctness, and hence are difficult to apply for security vulnerabilities where only one test (the exploit) may be available. Recent emergence of Large Language Models (LLMs) have put the focus on fixing "issues" where natural language bug reports may be used to produce rectifying program modifications via LLM agents. In this work, we have demonstrated the feasibility of using a repurposed LLM agent CodeRover-S for rectifying security vulnerabilities found by continuous fuzzing taken from the widely used OSS-Fuzz infrastructure [33]. The main experience gained points us to the feasibility of using LLM agents as back-ends to fuzzers for zero-day patching of security vulnerabilities. We demonstrated that full software protection from detection to repair is feasible with fuzzers and LLM agents, and there are future research potentials in patch correctness assessment to further improve the degree of automation.

Acknowledgments

This work is partially supported by the Singapore Ministry of Education (MoE) Tier3 research grant "Automated Program Repair" MOE-MOET32021-0001. This work is partially supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of Singapore Ministry of Education, National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2024. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. Accessed: 2024-10-03.

[3] The Fluent Bit Authors. 2025. Fluent Bit. https://github.com/fluent/fluent-bit. Accessed: 2025-09-29.

[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2188–2200.

[6] M. Böhme, C. Cadar, and A. Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021).

[7] Justin Campbell and Mike Walker. 2024. Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale. https://www.microsoft.com/en-us/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/. Accessed: 2024-10-03.

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

[9] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transaction on Software Engineering* (2019).

[10] Chromium. 2024. The Chromium Projects - Security Bugs. https://www.chromium.org/Home/chromium-security/bugs/. Accessed: 2024-10-03.

[11] Zhen Yu Ding and Claire Le Goues. 2021. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.

[13] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. doi:10.1145/3540250.3549098

[14] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages. doi:10.1145/3418461

[15] GitHub. 2024. CodeQL. https://codeql.github.com/. Accessed: 2024-10-03.

[16] GNU Project. 2024. *addr2line (GNU Binutils)*. https://linux.die.net/man/1/addr2line Version 2.40.

[17] GNU Project. 2024. GNU Debugger (GDB). https://sourceware.org/gdb/. Accessed: 2024-10-02.

[18] GNU Project. 2024. nm - List symbols from object files (GNU Binutils). https://linux.die.net/man/1/nm. Accessed: 2024-10-02.

[19] Google. 2024. AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed: 2024-10-31.

[20] Google. 2024. MemorySanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer. Accessed: 2024-10-31.

[21] Google. 2025. Honggfuzz. https://github.com/google/honggfuzz.

[22] Google. 2025. OSS Fuzz Issue Tracker. https://issues.oss-fuzz.com/issues. Accessed: 2025-09-29.

[23] C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62 (2019). Issue 12.

[24] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594* (2024).

[25] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering* (2023).

[26] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 539–554.

[27] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. doi:10.1109/ICSE43902.2021.00107

[28] Jan Keller and Jan Nowakowski. 2024. *AI-powered patching: the future of automated vulnerability fixes*. Technical Report.

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

[30] Open Asset Import Library. 2025. Open Asset Import Library (assimp). https://github.com/assimp/assimp. Accessed: 2025-09-29.

[31] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, Brendan Dolan-Gavitt, et al. 2024. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software. *arXiv preprint arXiv:2408.02153* (2024).

[32] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated Adaptive Prompting Large Language Models for {Real-World} Software Vulnerability Patching. In *34th USENIX Security Symposium (USENIX Security 25)*. 4481–4500.

[33] OSS-Fuzz. 2024. OSS-Fuzz. https://google.github.io/oss-fuzz/. Accessed: 2024-10-03.

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

[35] LLVM Project. 2024. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-10-03.

[36] LibTIFF project. 2025. LibTIFF: TIFF Library and Utilities. https://gitlab.com/libtiff/libtiff. Accessed: 2025-09-29.

[37] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 963–974.

[38] Semgrep. 2024. Semgrep. https://semgrep.dev/. Accessed: 2024-10-03.

[39] Skybox. 2024. Vulnerability & Threat Trends Report 2024. Accessed: 2024-10-03.

[40] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. doi:10.1145/2786805.2786825

[41] The Clang Team. 2024. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: 2024-10-25.

[42] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[43] WizardMac. 2025. ReadStat: Read (and write) data sets from SAS, Stata, and SPSS. https://github.com/WizardMac/ReadStat. Accessed: 2025-09-29.

[44] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824.

[45] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[46] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. Patchagent: A practical program repair agent mimicking human expertise. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security'25), Seattle, WA, USA*.

[47] Michal Zalewski. 2024. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2024-10-03.

[48] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 691–702. doi:10.1145/3533767.3534387

[49] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[50] Han Zheng, Ilia Shumailov, Tianqi Fan, Aiden Hall, and Mathias Payer. 2025. Fixing 7,400 Bugs for 1$: Cheap Crash-Site Program Repair. *arXiv preprint arXiv:2505.13103* (2025).

[51] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 88, 13 pages. doi:10.1145/3597503.3639222