Fixing 7,400 Bugs for 1$: Cheap Crash-Site Program Repair

Han Zheng
EPFL
Lausanne, Switzerland
han.zheng@epfl.ch

Ilia Shumailov
Google DeepMind
London, UK
iliashumailov@google.com

Tianqi Fan
Google
Zurich, Switzerland
tqfan@google.com

Aiden Hall
Google
New York, USA
aidenhall@google.com

Mathias Payer
EPFL
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Abstract

The rapid advancement of bug-finding techniques has led to the discovery of more vulnerabilities than developers can reasonably fix, creating an urgent need for effective Automated Program Repair (APR) methods. However, the complexity of modern bugs often makes precise root cause analysis difficult and unreliable. To address this challenge, we propose *crash-site repair* to simplify the repair task while still mitigating the risk of exploitation. In addition, we introduce a *template-guided patch generation* approach that significantly reduces the token cost of Large Language Models (LLMs) while maintaining both efficiency and effectiveness.

We implement our prototype system, WILLIAMT, and evaluate it against state-of-the-art APR tools. Our results show that, when combined with the top-performing agent CodeRover-S, WILLIAMT reduces token cost by 45.9% and increases the bug-fixing rate to 73.5% (+29.6%) on ARVO, a ground-truth open source software vulnerabilities benchmark. Furthermore, we demonstrate that WILLIAMT can function effectively even without access to frontier LLMs: even a local model running on a Mac M4 Mini achieves a reasonable repair rate. These findings highlight the broad applicability and scalability of WILLIAMT.

1 Introduction

Modern software systems are extremely complex. While this ever-increasing complexity enables richer functionality, it simultaneously introduces a significant number of (exploitable) vulnerabilities [32, 3, 4, 23, 41, 29, 52]. To proactively safeguard users from malicious threats, fuzzing [58, 12, 25, 62, 61] emerged to automate the bug finding. Over the years fuzzing demonstrated its extreme usefulness and efficiency even in the most complex and security-critical systems, including but not limited to web browsers [55, 63, 19, 50], Linux kernels [17, 13], MacOS [65, 57], Hypervisors [35, 27, 5, 28] and Trust Execution Environments [51]. However, the advancements in fuzzing also increase demands on developers tasked with fixing the discovered vulnerabilities. For example, despite extensive maintenance efforts, more than 1,500 kernel bug reports remain unresolved on Syzbot [44]. Modern fuzzers uncover vulnerabilities at a pace that far outstrips the ability of developers to address them. What is worse, the process of fixing bugs requires renewed fuzzing efforts to check the fixes, which in turn often reveals additional bugs.

This growing discrepancy between the bug discovery and bug fix rate has led to a backlog that exceeds developers' capacity to triage and remediate effectively [24]. These challenges underscore

Preprint. Under review.

the urgent need for Automated Program Repair (APR) solutions. Even imperfect, such solutions would significantly improve the security of our systems and allow the developers to focus on the most complex of bugs. Recent advancements in Large Language Models (LLMs) [47, 26, 20, 1, 34] offer a promising direction for addressing this challenge. Leveraging the capabilities of LLMs, a variety of novel Automatic Program Repair (APR) techniques have emerged [60, 22, 21, 45, 59].

Despite their promise, current APR agents struggle as they attempt to address bugs by identifying and fixing their root causes. This strategy is often impractical – root cause analysis is inherently complex, if not impossible, and typically requires expensive, heavyweight LLM backends, making it difficult for individual developers to adopt and deploy these solutions at scale. Moreover, root cause analysis relies on the LLMs' ability to accurately perform multiple reasoning tasks [59, 60], and imprecision in any of these stages can necessitate repeated attempts at repair [54], further increasing the number of LLM queries and the associated runtime costs.

In contrast to these imprecise, high-cost root-cause-oriented methods, in this paper we advocate for an alternative lightweight, ad-hoc repair approach that prevents immediate exploitation [8] rather than fixing the bug at the root cause. Root-cause fixes often require a deep understanding of the program and complex code changes in components that may not appear in the crash stack, whereas crash-site fixes are quicker and can block exploitation by adding a simple assertion right before crash site. This thereby grants developers more time for an in-depth investigation while the immediate threat is contained. To this end, we propose WILLIAMT, a new template-based APR agent specifically designed to repair memory corruption vulnerabilities given a Proof-of-Concept (PoC) input. WILLIAMT is inspired by the folk hero William Tell who had, under pressure, one shot to hit an apple on his kid's head. Similarly, WILLIAMT aims for "one-shot" fixes to promptly block exploitation. WILLIAMT operates by taking fuzzer-generated PoCs and Sanitizer outputs [40] as input. It employs a regular-expression-based method for both fault localization and patch generation, reserving LLM usage exclusively for root cause analysis when necessary. By minimizing reliance on LLMs and constraining their outputs via predefined bug templates, WILLIAMT substantially reduces query costs and enables deployment with smaller and cheaper LLMs. This makes WILLIAMT a scalable and practical APR solution, particularly suitable for resource-constrained environments.

We leverage ARVO [30], a ground-truth open source software vulnerability benchmark for our evaluation and compare WILLIAMT against state-of-the-art APR agents based on LLMs. While the CodeRover-S [60] is the best-performing agent, WILLIAMT reduces its token usage by 99.7% while retaining over 86.7% of the CodeRover-S. Moreover, when combined with CodeRover-S, *i.e.,* fixing using WILLIAMT and forward the unfixed bugs to CodeRover-S, the optimized pipeline reduces token cost by 45.9% and improves the fixing rate by 29.6% compared to CodeRover-S alone. We further extend WILLIAMT to work with both cutting-edge and local LLMs. Our findings indicate that WILLIAMT not only integrates effectively with frontier LLMs but also scales efficiently when deployed with local models running on a Mac Mini, demonstrating its broad applicability and scalability. To sum up, our contributions are:

- We propose *crash-site repair* to avoid complex and imprecise root cause analysis.
- We introduce a *template-guided patch generation* technique to reduce LLM inference cost.
- We implement and evaluate WILLIAMT, demonstrating that—when combined with a leading SoTA method—it achieves a 45.9% cost reduction and a 29.6% increase in fixing precision.
- We promise to fully release WILLIAMT upon paper acceptance to support open science.

2 Background

2.1 LLM-Based Program Repairing

The emergence of Large Language Models (LLMs) has inspired researchers to explore their potential in enabling agents for Automatic Program Repair (APR). In contrast to earlier approaches that relied on supervised model training or LLM fine-tuning [49, 7, 11, 22], agent-based methods benefit directly from the continual advancements in foundational LLMs [31, 18, 26, 20, 1, 48, 56], without necessitating additional manual tuning.

To further reduce the pretraining and fine-tuning costs, recent efforts have proposed agent-based APR frameworks [53, 54]. These frameworks typically consist of three core components, as depicted

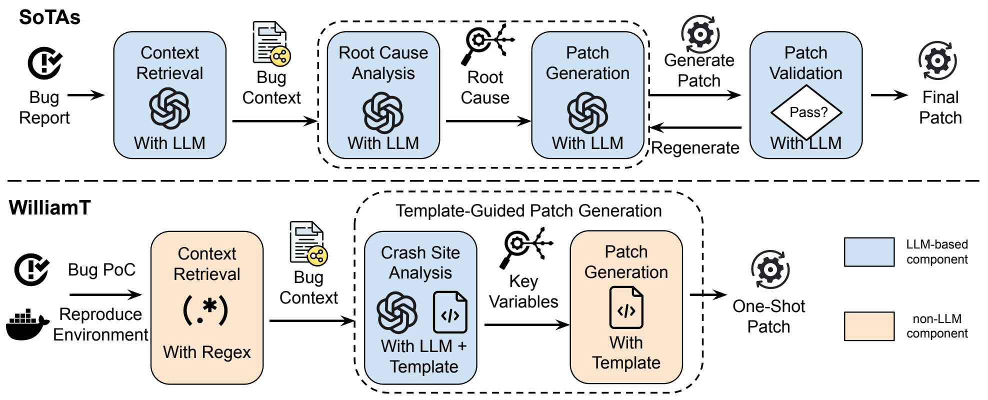


Figure 1: Workflow of Current LLM-based Program Repairing Agents vs WILLIAMT. 🟦 stands for the LLM components while 🟧 means non-LLM modules.

```
// stack/heap/global buffer
type *buf = get_buffer(SIZE);
// idx < 0 or >= SIZE
type value = buf[idx];
```

(a) Spatial Memory Corruption

```
// buffer is invalid
invalid_buffer(buf);
// reuse the content in buffer
type value = buf[idx];
```

(b) Temporal Memory Corruption

Figure 2: Example Memory Corruption Vulnerabilties.

in Figure 1. First, the agent processes the bug report, retrieves relevant context, and submits it to the LLM for root cause analysis and patch generation. The generated patch is then passed back to the LLM for validation. If the patch fails validation, the process iterates, returning to the patch generation phase, until a satisfactory patch is produced.

However, current agents delegate all four steps—retrieval, root cause analysis, patch generation, and validation—entirely to LLMs. This results in a substantial increase in token consumption. Moreover, the quality of each module's output is highly dependent on the size and capacity of the underlying LLM, making it impractical to deploy such agents using lightweight or locally hosted models. This challenge highlights the need for a more efficient agent architecture that can scale with smaller LLMs, while minimizing reliance on the model's capacity and reducing overall token consumption.

2.2 Memory Corruption Vulnerabilities and OSS-Fuzz Benchmarks

Over the past 30 years, memory corruption vulnerabilities remain critical in modern software systems [33, 46, 42]. This persistence underscores the importance of APR tools and their ability to address such vulnerabilities. Consequently, the capability to fix memory corruption bugs has become a key metric for evaluating APR tools.

Memory corruption vulnerabilities can be broadly divided into two major categories [36]: spatial and temporal memory corruptions (Figure 2). *Spatial memory corruption* occurs when a program accesses memory outside the bounds of an allocated buffer. These accesses can involve the stack, heap, or global memory regions. *Temporal memory corruption*, on the other hand, results from accessing a buffer that has been deallocated or is no longer valid at the time of access. This includes accessing an already-freed heap buffer or a returned stack buffer that is no longer valid.

Memory corruption bugs are prevalent in open-source software and often propagate into security-critical systems such as Chrome and iOS [38, 39]. To mitigate these risks, Google introduced OSS-Fuzz [16], a continuous fuzzing infrastructure designed to test open-source software (OSS) at scale and protect users against potential vulnerabilities. As of now, OSS-Fuzz has discovered and helped fix over 36,000 bugs— many of which were memory-related [6]. This rich and diverse dataset makes OSS-Fuzz an ideal benchmark for evaluating APR tools. ARVO [30], inspired by OSS-Fuzz,

```
// [1] crash site: yy_c >= sizeof(yy_current_state)
// /src/igraph/build/src/io/parsers/dl-lexer.c:3536
(yy_trans_info = &yy_current_state[yy_c])->yy_verify == yy_c;

// [2] root cause: wrong compile flag
// /src/CMakeLists.txt
- COMPILE_FLAGS "${flex_hide_line_numbers} -CF -8"
+ COMPILE_FLAGS "${flex_hide_line_numbers} -Cf -8"
```

Figure 3: Root Cause and Crash Site of Bug 66992.

reconstructs and curates a reproducible dataset of OSS bugs specifically tailored for APR evaluation. Compared to other ground-truth memory corruption datasets [14, 10, 2], ARVO automates the entire compilation pipeline and ensures reproducibility. This automation facilitates fair and consistent comparisons across different APR tools.

3 Primer on Bug Fixing

The objective of automated program repair (APR) tools can be broadly categorized into three progressive stages: gracefully crashing, bailing out and aborting the function, and fixing the root caus of the bug. In the first stage, the primary goal is to prevent potential exploitation. Patches at this level may terminate program execution immediately upon detecting a fault, thereby avoiding the execution of potentially dangerous code paths. In the second stage, patches aim to handle the error more gracefully by returning appropriate error codes or statuses to the caller. This ensures the program remains operational and that the presence of a bug does not compromise overall system availability. Finally, the most desirable stage involves addressing the root cause of the bug, thereby preventing the fault from occurring in the first place. This level of repair reflects a complete and semantically correct fix that preserves both safety and functionality.

While many agent-based APR tools aim to address vulnerabilities at their root cause, this is not always practical. In real-world scenarios, APR systems typically operate on limited inputs—either a natural language bug report or a sanitizer-generated crash report—which only provide information about the *crash site*. However, the *root cause* may reside in an entirely different source file or even in a separate component of the system. For instance, in the case illustrated in Figure 3, a global buffer overflow vulnerability arises in the *igraph* library due to the index variable `yy_c` exceeding the bounds of the global array `yy_current_state`. Importantly, this code was not manually written but automatically generated by `flex`. The true root cause lies in the incorrect use of compilation flags, which the developer ultimately resolved by modifying the CMakefile. This example demonstrates that effective repair sometimes requires cross-domain analysis across build systems, code generators, and runtime behaviors—capabilities that are beyond the scope of current large language models (LLMs), due to limitations in contextual capacity and abstract reasoning.

To address this challenge, we draw inspiration from Chrome's development guide [8], which recommends inserting CHECK statements (*i.e.,* assertions) to mitigate the risk of exploitation before a root cause fix is available. This strategy transforms potentially exploitable vulnerabilities into intentional, controlled crashes, giving developers time to implement a permanent solution.

Building on this principle, WILLIAMT directly inserts the patches before crash site by leveraging the predefined templates and *crash site analysis* (as illustrated in Appendix B). Specifically, we focus on four widely prevalent categories of exploitable vulnerabilities [30]: Heap-Buffer-Overflow (HBO), Global-Buffer-Overflow (GBO), Stack-Buffer-Overflow (SBO), and Heap-Use-After-Free (UAF). Each category is associated with a template that encapsulates common crash patterns and patch strategies. This *crash-site repair* paradigm significantly simplifies the complexity of bug fixing and circumvents the need for precise root cause localization. As a result, WILLIAMT achieves comparable repair effectiveness while requiring fewer than 1% of the LLM queries typically consumed by root-cause-based systems, making it more scalable to resource-constrained or local LLM environments.

4 WILLIAMT Design

We propose WILLIAMT, a lightweight and scalable system designed to eliminate the excessive query cost associated with LLMs, scaling APR agents to smaller model. WILLIAMT takes as input a ClusterFuzz-generated Proof-of-Concept (PoC) [15] and a reproducible Docker image as input, finally produces a one-shot patch for the vulnerable program. WILLIAMT comprises two main components: **Regex-Based Context Retrieval** and **Template-Guided Patch Generation**, as illustrated in Figure 1.

Regex-Based Context Retrieval. By reproducing the PoC within the provided Docker environment, WILLIAMT utilizes the sanitizer report [40] to identify the crash site through regex-based matching. This approach offloads the context extraction task from token-intensive LLM queries, significantly improving performance and scalability. While the crash site is not always equivalent to the true root cause, generating a patch just before the crash point is sufficient to mitigate exploitation [8]. Therefore, WILLIAMT provides the LLM agent with a focused context surrounding the crash site. We include technical implementations in Appendix C.

Template-Guided Patch Generation. Directly prompting an LLM to generate a fix often results in arbitrary and inconsistent outputs, highly dependent on the LLM's internal capabilities. To address this, WILLIAMT categorizes each bug based on its memory corruption type (*e.g.,* , heap-buffer-overflow, global-buffer-overflow, stack-buffer-overflow, heap-use-after-free). For each bug category, WILLIAMT selects a predefined patch template and guides the LLM to identify only the key variables relevant to the patch. This template-based constraint significantly reduces the LLM's cognitive burden by offloading most of the patch generation logic, requiring the LLM only to perform minimal crash site analysis. We provide concrete details in Appendix B. This agent design enables smaller LLMs to achieve reasonable performance with significantly reduced resource requirements.

5 Evaluation

We evaluate WILLIAMT to answer the following research questions (RQs): 1) RQ1: Does WILLIAMT outperforms other SoTAs in terms of bug fixing capability? 2) RQ2: Does WILLIAMT scale on smaller LLMs? 3) RQ3: What kind of bugs does WILLIAMT and SoTAs fix?

Benchmark Selection. We utilize ARVO [30], a high-quality dataset comprising over 5,000 ground-truth memory corruption bugs drawn from more than 250 real-world software projects. In contrast to manually constructed benchmarks [9] or non-reproducible datasets [2, 10], ARVO ensures that all bugs are both ground-truth and reproducible. Moreover, each bug in ARVO originates from OSS-Fuzz [16], thereby closely aligning the dataset with real-world vulnerability discovery scenarios. Specifically, we choose all HOF, SOF, UAF and GOF bugs (358 bugs) that can be compiled within 15 minutes following the recommended practice [60].

Repair Metrics. We classify the fix into following categories [21, 60]: No Code: the code is generated by build system; No Patch: LLM failed to generate patch or the patched program does not compile; Implausible: patched program compiles but still crashes when running with PoC; Plausible: patched program does not crash when taking the PoC as input. We consider the plausible rate as the successful fixing rate to align with other SoTAs [60].

Evaluation Setup. All our agents are running on a Ubuntu 22.04 server equipped with AMD EPYC 7302P and 64GB RAM. We use the official APIs to access the frontier LLM models and run local LLMs on a RTX 4090 GPU. We also test the local model Gemma3:4b on Mac Mini M4 (16 GB RAM) and it performs on par to RTX 4090, illustrating WILLIAMT's scalability.

LLM Selection. We select DeepSeek (V3, R1), ChatGPT (4o, o3-mini), Claude (3.5-Haiku, 3.7-Sonnet) as frontier LLM baselines, and choose Gemma3 (27B, 12B, 4B, 1B) as local LLMs.

5.1 RQ1: Does WILLIAMT Outperforms Other SoTAs?

We compare WILLIAMT against state-of-the-art APR tools, including AutoCodeRover-S [60], Agentless [54] and VulMaster [64]. As AutoCodeRover-S is not open-source, we import the fixes provided by AutoCodeRover-S for all three SoTAs for the same configuration. Additionally, we include WILLIAMT with same LLM model (gpt-4o-2024-08-06) to ensure a fair comparision.

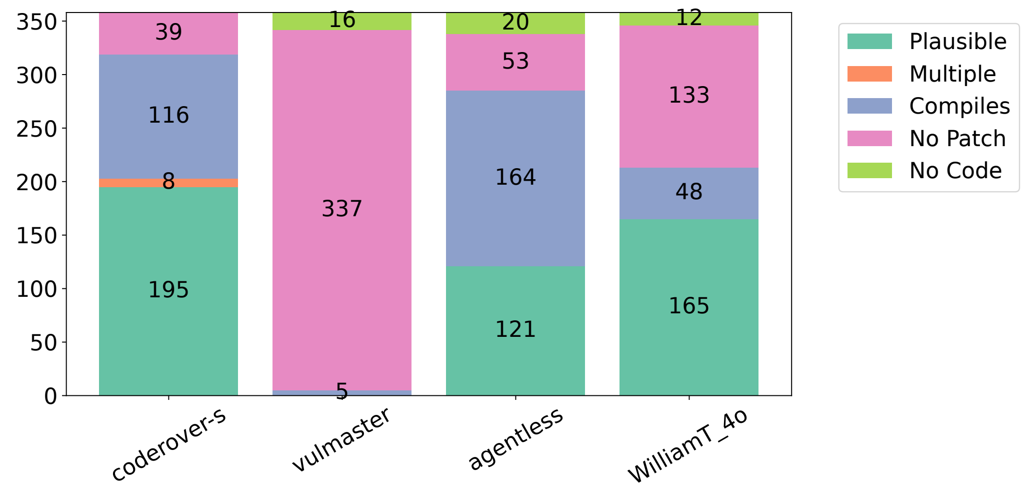


Figure 4: The fix performance of WILLIAMT (with GPT-4o) and other SoTAs [60, 64, 54]. Multiple means the CodeRover-S takes multiple attempts to find plausible fix.

Figure 4 presents the overall performance comparison. CodeRover-S achieves the highest plausible fixing rate at 54.5%, followed by WILLIAMT, which successfully fixes 46.1% of bugs. In contrast, Agentless performs significantly worse, with a fixing rate of 33.8%, and VulMaster resolves only 5 bugs in total. Although WILLIAMT achieves slightly lower fixing performance than CodeRover-S, it requires a magnitude lower resources. CodeRover-S attempts one patch per trial and conducts up to three trials per bug [60], which, in total, requires up to **18** attempts. In contrast, WILLIAMT performs a single trial and applies only **one** patch throughout the entire fixing pipeline. This one-shot design significantly reduces computational and token costs compared to the SoTAs.

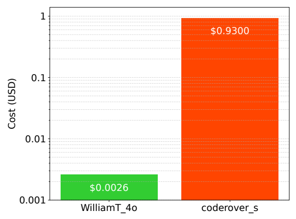
Figure 5a compares the token and time costs between WILLIAMT and CodeRover-S. Both systems use the GPT-4o model (as of 2024-08-06). However, WILLIAMT incurs an average cost of only 0.0026$ per bug, whereas CodeRover-S requires approximately 0.93$. In other words, WILLIAMT is able to fix **357 times** more bugs per dollar spent, given the same backend model.

This efficiency directly contributes to the significant disparity in repair time between the two systems. As shown in Figure 5b, WILLIAMT achieves substantial time savings compared to CodeRover-S. On average, WILLIAMT completes the entire process—including preprocessing and LLM-based analysis—in under one minute. In contrast, CodeRover-S requires approximately 43.5 minutes to complete one task. The key distinction lies in their respective repair strategies. CodeRover-S employs a multi-iteration repair loop, which not only prolongs compilation time but also imposes a considerable load on system resources. In contrast, WILLIAMT leverages a single-shot patching approach that avoids the overhead of repeated recompilation. This design choice significantly reduces CPU consumption and expedites the overall repair workflow.

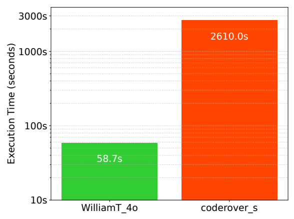
RQ1: WILLIAMT saves 97.7% CPU usage and 99.7% Token cost, while preserving over 86.7% performance.

5.2 RQ2: What kind of bugs does WILLIAMT fix?

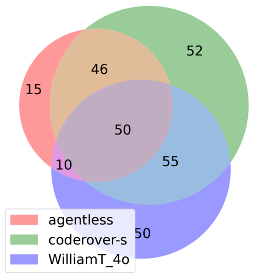
We begin by analyzing the overlap of plausible fixes among SoTA systems, as shown in Figure 5c. Although CodeRover-S fixes slightly more bugs overall than WILLIAMT, the sets of bugs each system addresses are largely disjoint. In other words, WILLIAMT and CodeRover-S tend to fix different bugs. Specifically, WILLIAMT successfully resolves 50 bugs that neither CodeRover-S nor Agentless addresses, while CodeRover-S exclusively fixes 52 bugs. Agentless contributes 15 unique plausible fixes not covered by either WILLIAMT or CodeRover-S.



(a) Average Price Cost between WILLIAMT and CodeRover-S.



(b) Average Time Cost between WILLIAMT and CodeRover-S.



(c) Venn Diagram of Plausible Patches. VulMaster failed to generate any plausible patches. 95 bugs are not fixed by any tools.

This low overlap highlights the complementary nature of WILLIAMT, suggesting it can be effectively used in combination with other agents. When using GPT-4o as its backend, WILLIAMT can fix all 358 bugs for under 0.68$—less than the cost of fixing a single bug with CodeRover-S.

Moreover, an optimized pipeline—where executing WILLIAMT is first, and applying CodeRover-S to cases that WILLIAMT fails—achieves 60 additional plausible fixes, *i.e.,* 29.6% improvement in fixing rate compared to CodeRover-S standalone, while reducing the total cost by 45.9%.

RQ2: Combining WILLIAMT and SoTA saves 45.9% cost while fixing 29.6% more bugs.

5.3 RQ3: Does WILLIAMT scale on other LLMs?

To prove WILLIAMT's generality, we try WILLIAMT on different set of LLMs, including DeepSeek (R1, V3), ChatGPT (4o, o3-mini), Claude (3.5-Haiku and 3.7-Sonnet), and local LLMs that can be deployed on an 4090 GPU or even Mac Mini M4, including Gemma3 (1B, 4B, 12B, 27B).

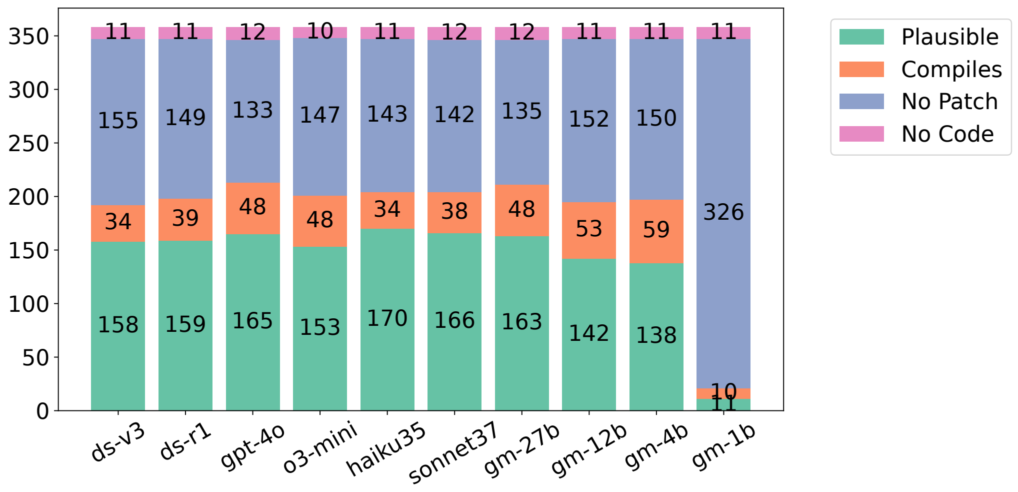
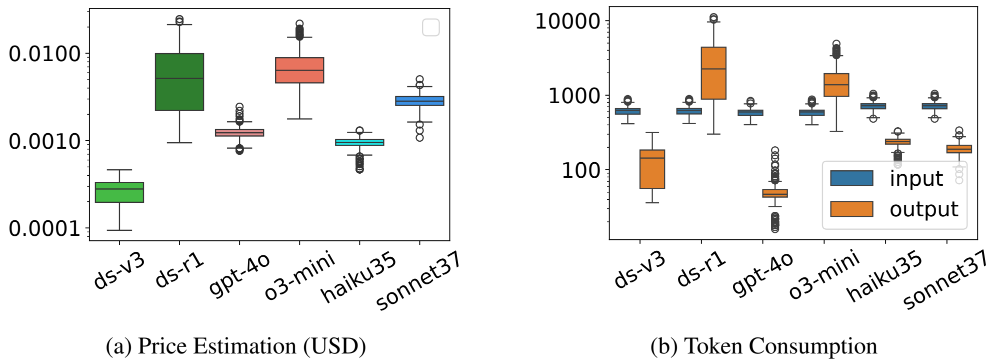


Figure 6: The WILLIAMT performance with different LLMs. ds: DeepSeek, gm: Gemma3.

Figure 6 shows the performance of different LLMs. Among all models, Claude35-haiku achieves the highest fixing rate at 47.5%. Claude37-sonnet and GPT-4o follow closely, with fixing rates of 46.4% and 46.1%, respectively. DeepSeek also performs well, with 159 successful fixes using the reasoner and 158 using the chat. GPT-o3-mini performs slightly lower, fixing 153 bugs and achieving a success rate of 43.9%.

Focusing on smaller language models, particularly the Gemma3 series, we observe that they perform comparably well relative to frontier models. Notably, Gemma3:27B ranks second overall, achieving 96.4% of GPT-4o's performance. While the 12B and 4B versions of Gemma3 exhibit a noticeable drop in performance, they still manage to fix 39.4% and 38.0% of bugs, respectively.

Importantly, the Gemma3:27B model is lightweight enough to run on a single consumer-grade GPU, such as an RTX 4090, or even on a Mac Mini M4. This demonstrates not only the scalability of WILLIAMT but also its practical feasibility for deployment on local developer machines.



(a) Price Estimation (USD) (b) Token Consumption

Figure 7: Token and Price Cost for different LLMs. ds: DeepSeek, gm: Gemma3.

We further analyze the cost efficiency of various frontier LLM backends. All cost are measured in cent USD, *i.e.,* 0.01 USD. As shown in Figure 7, DeepSeek-V3 achieves the highest efficiency, with a cost of less than 0.03 cent per bug. Claude35-Haiku, while attaining the best fixing performance, maintains a moderate cost of approximately 0.09 cent per bug. In contrast, reasoning models such as DeepSeek-R1 and GPT-o3-mini exhibit significantly higher costs, at 0.70 cent and 0.72 cent per bug, respectively—5.7× and 5.9× more expensive than Claude35-Haiku—without providing notable improvements over non-reasoning LLMs.

The primary reason for the increased cost is the excessive output token. Although all models consume a similar number of input tokens, their output token consumption varies significantly. DeepSeek-V3 and GPT-4o generate an average of 131 and 49 output tokens per bug, respectively, whereas Claude35-Haiku averages 236 output tokens. Among reasoning models, DeepSeek-R1 and GPT-o3-mini produce substantially more output, averaging 3,060 and 1,564 output tokens per bug, respectively. An exception is Claude-3-7-Sonnet, a hybrid reasoning model that performs on-demand reasoning. Its average of 191 output tokens suggests that our task setup largely avoids heavy reasoning. Importantly, this large increase in output tokens does not correlate with better fixing performance. In fact, Claude35-Haiku, a non-reasoning model, achieves the best results across all models.

These findings highlight the potential of WILLIAMT as an agent: by providing a strong template, it can eliminate the need for expensive reasoning, enabling non-reasoning models to outperform their reasoning-based counterparts.

We further decompose the time cost associated with bug analysis in WILLIAMT. As a one-shot Automated Program Repair (APR) agent, WILLIAMT requires only a preprocessing phase to extract necessary contextual information—independent of the choice of LLM backend—and a subsequent LLM-based key variable analysis. The breakdown is illustrated in Figure 8.

On average, the preprocessing takes less than 1 minute, primarily due to Python dependency compilation and source code analysis. We also evaluate the execution time of state-of-the-art LLMs used for the key variable analysis. Non-reasoning models typically complete this step in under 10 seconds, whereas reasoning-capable models range from 13 seconds (*i.e.,* o3-mini) to 120 seconds (*i.e.,* DeepSeek-R1). For local execution, we benchmarked the Gemma-3 variants on an RTX 4090 GPU, observing runtimes between 1 and 13 seconds. Overall, the majority of WILLIAMT 's execution time is attributed to preprocessing. Even when using the slowest reasoning model, WILLIAMT consistently completes the bug-fixing task in under 3 minutes—illustrating a substantial improvement over CodeRover-S, which requires approximately 43.5 minutes.

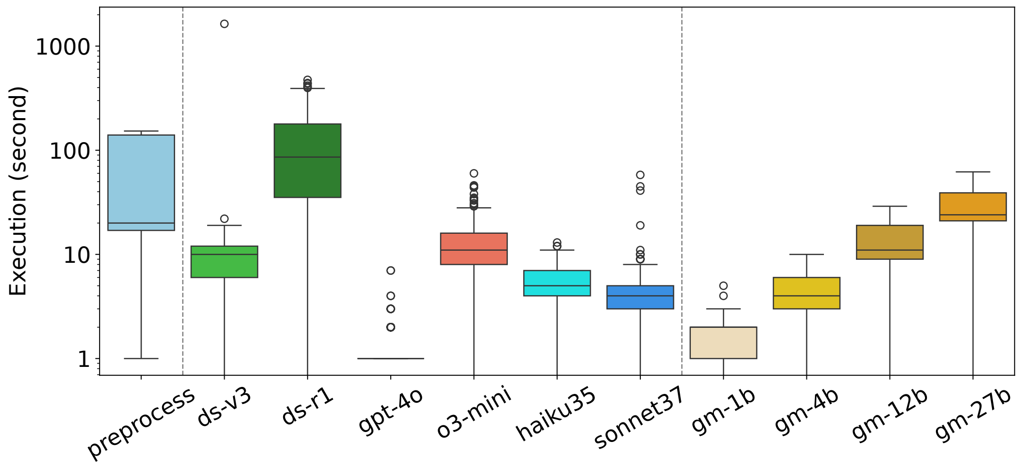


Figure 8: Execution cost for preprocessing and different LLMs to analysis variable. The dot lines split the preprocessing, frontier LLMs and local LLMs.

RQ3: WILLIAMT performs well with frontier models while remaining scalable to smaller, more cost-effective LLMs.

6 Discussion

WILLIAMT produces a correct patch when both the crash site analysis accurately identifies relevant variables and the patch insertion is semantically valid within the target code. In this section, we examine potential sources of false positives in WILLIAMT, focusing on internal and external threats to patch plausibility. Internally, inaccuracies may arise from *incorrect crash site analysis* or *semantically disruptive patch insertion*. Externally, the *imprecise plausible fix metric* used during evaluation may fail to distinguish correct from incorrect patches.

Incorrect Crash Site Analysis. While WILLIAMT demonstrates a promising plausible fix rate, we observe cases where key variables returned by the LLM are inaccurate. This issue can stem from two primary factors. First, the code snippet provided to the LLM may lack the correct variable due to limited context, especially when the required variables lies beyond the final crash frame. Second, the precision of variable identification is constrained by the LLM's capabilities and the effectiveness of prompt engineering. Addressing these limitations—potentially through enhanced context retrieval or improved prompt strategies—is left as future work. We further discuss its impact in Appendix A.

Semantically Disruptive Patch Insertion. WILLIAMT currently inserts the generated patch one line above the crash site. While this approach may work in principle, it can lead to semantic errors and compilation failures in practice. For instance, if the crash occurs within a conditional block (*e.g.,* inside an `if (...)` statement), naive insertion can break control flow and invalidate the program structure. In our evaluation, 'No Patch' failures account for approximately 37% of all bugs, with many caused by such semantic violations. Incorporating LLM-guided semantic-aware insertion strategies may offer a viable solution to mitigate this issue.

Imprecise Plausible Metric. Our evaluation follows the standard plausibility criterion from prior work [60], which verifies that a generated patch prevents the original crash. However, this metric does not ensure that the program's functionality is preserved. Consequently, it may overestimate the number of truly correct fixes and introduce false positives across all evaluated systems. This limitation highlights the need for more rigorous metrics that account for semantic correctness and broader behavioral preservation.

7 Conclusion

The increasing number of discovered security bugs creates a strong need for better automated program repair (APR). Instead of relying on complex root cause analysis like most existing APR tools, we propose *crash-site repair* to simplify the repair process and improve precision. We also introduce a *template-guided patch generation* technique to reduce the high cost of using LLMs, while preserving the repairs effectiveness. We implement our prototype WILLIAMT, and evaluate it against state-of-the-art tools. Our results show that combining WILLIAMT with the best-performing tool, CodeRover-S, reduces token usage by 29.6% and improves the fixing rate by 45.9% compared to CodeRover-S alone. In addition, WILLIAMT works well with local LLMs on a Mac Mini, showing its potential for low-cost, local development. We promise to fully open-source WILLIAMT upon paper acceptance.

References

[1] X AI. Grok. `https://x.ai/`, 2025.

[2] Guru Bhandari, Amara Naseer, and Leon Moonen. 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*, pages 30–39, 2021.

[3] Liu Bohan and Shi Haibin. Escape modern web-based app sandbox from site-isolation perspective. `https://i.blackhat.com/Asia-24/Presentations/Asia-24-Liu-The-Hole-in-Sandbox.pdf`, 2024.

[4] Liu Bohan and Wang Zheng. Reviving jit vulnerabilities: Unleashing the power of maglev compiler bugs on chrome browser. `https://i.blackhat.com/EU-23/Presentations/EU-23-Liu-Reviving-JIT-Vulnerabilities.pdf`, 2023.

[5] Alexander Bulekov, Qiang Liu, Manuel Egele, and Mathias Payer. {HYPERPILL}: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 919–935, 2024.

[6] Oliver Chang and OSS-Fuzz team. Taking the next step: Oss-fuzz in 2023, 2023.

[7] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[8] chromium. Top security things for chromies to remember. `https://chromium.googlesource.com/chromium/src/+/lkgr/docs/security/checklist.md`, 2025.

[9] DARPA. Cyber grand challenge - datasets, 2025.

[10] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512, 2020.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, pages 10–10, 2020.

[13] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, 2023.

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

[15] google. Clusterfuzz. `https://google.github.io/clusterfuzz/`, 2023.

[16] Google. Oss-fuzz - continuous fuzzing for open source software, 2025.

[17] Google. syzkaller is an unsupervised coverage-guided kernel fuzzer. `https://github.com/google/syzkaller`, 2025.

[18] Google. Welcome gemma 3: Google's all new multimodal, multilingual, long context open llm. `http://huggingface.co/blog/gemma3`, 2025.

[19] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[20] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[21] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174. IEEE, 2023.

[22] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. Template-guided program repair in the era of large language models. ICSE, 2025.

[23] Rong Jian and Guang Gong. Another way to talk with browser : Exploiting chrome at network layer. `https://i.blackhat.com/USA-22/Thursday/US-22-Rong-Another_Way_to_Talk_with_Browser_Exploiting_Chrome_at_Network_Layer.pdf`, 2022.

[24] Angelos Keromytis. Recommendations from the workshop on open-source software security initiative. `https://bpb-us-e1.wpmucdn.com/sites.gatech.edu/dist/a/2878/files/2022/10/OSSI-Final-Report.pdf`, 2022.

[25] libfuzzer. libfuzzer. `https://llvm.org/docs/LibFuzzer.html`, 2023.

[26] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[27] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. Videzzo: Dependency-aware virtual device fuzzing. In *2023 IEEE Symposium on security and privacy (SP)*, pages 3228–3245. IEEE, 2023.

[28] Zheyu Ma, Qiang Liu, Zheming Li, Tingting Yin, Wende Tan, Chao Zhang, and Mathias Payer. Truman: Constructing device behavior models from os drivers to fuzz virtual devices.

[29] Bogaard Martijn and Geist Dana. Achieving linux kernel code execution through a malicious usb device. `https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard_Geist_Achieving_Linux_Kernel_Code_Execution_Through_A_Malicious_USB_Device.pdf`, 2021.

[30] 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. Arvo: Atlas of reproducible vulnerabilities for open source software. *arXiv preprint arXiv:2408.02153*, 2024.

[31] Meta. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`, 2025.

[32] Wang Nan and Xiao Zhenghang. Exploit chrome and firefox four times. `https://i.blackhat.com/BH-US-24/Presentations/US24-Xiao-Super-Hat-Trick-Exploit-Chrome-and-Firefox.pdf`, 2024.

[33] Aleph One. Hacking the stack for fun and profit. *Phrack Magazine*, 1996.

[34] OpenAI. Chatgpt. `https://chatgpt.com`, 2025.

[35] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2197–2213, 2021.

[36] Mathias Payer. *Software Security: Principles, Policies, and Protection*. HexHive Books, 0.37 edition, July 2021.

[37] Alexander Potapenko. Addresssanitizercallstack, 2015.

[38] External reporter. Security: heap-buffer-overflow in libavif when decode the crafted avif file, 2024.

[39] Apple Security. ios 16.1.1 and ipados 16.1.1, 2024.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[41] Röttger Stephen. Breaking the chrome sandbox with mojo. `https://i.blackhat.com/USA-22/Wednesday/US-22-Roettger_Breaking_the_Chrome_Sandbox_with_Mojo.pdf`, 2022.

[42] Jeffrey Vander Stoep. Memory safe languages in android 13, 2022.

[43] Maddie Stone. The more you know, the more you know you don't know - a year in review of 0-days used in-the-wild in 2021, 2022.

[44] Syzbot. Syzbot open bugs, 2025.

[45] Hao Tang, Keya Hu, Jin Zhou, Si Cheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. Code repair with llms gives an exploration-exploitation tradeoff. *Advances in Neural Information Processing Systems*, 37:117954–117996, 2024.

[46] Chromium Security Team. Memory safety in chromium, 2025.

[47] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[48] Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[50] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. Dumpling: Fine-grained differential javascript engine fuzzing.

[51] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, et al. Syztrust: State-aware fuzzing on trusted os designed for iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2310–2387. IEEE, 2024.

[52] Reguła Wojciech and Fitzl Csaba. 20+ ways to bypass your macos privacy mechanisms. `https://i.blackhat.com/USA21/Wednesday-Handouts/US-21-Regula-20-Plus-Ways-to-Bypass-Your-macOS-Privacy-Mechanisms.pdf`, 2021.

[53] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.

[54] Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[55] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[56] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[57] Tingting Yin, Zicong Gao, Zhenghang Xiao, Zheyu Ma, Min Zheng, and Chao Zhang. {KextFuzz}: Fuzzing {macOS} kernel {EXTensions} on apple silicon via exploiting mitigations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5039–5054, 2023.

[58] Michal Zalewski. american fuzzy lop. `https://lcamtuf.coredump.cx/afl/`, 2013.

[59] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[60] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, Dongge Liu, Abhishek Arya, Oliver Chang, and Abhik Roychoudhury. Fixing security vulnerabilities with ai in oss-fuzz. *arXiv preprint arXiv:2411.03346*, 2024.

[61] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. Mendelfuzz: The return of the deterministic stage. 2025.

[62] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1343–1360, 2023.

[63] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: browser api fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1135–1147, 2022.

[64] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David 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*, pages 1–13, 2024.

[65] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. Crossfire: Fuzzing macos cross-xpu memory on apple silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3749–3762, 2024.

A The Plausible Fix and Actual Fix

In this section, we discuss the differences between a *plausible fix* and a *successful fix*.

Current practice [60] uses the *plausible fix* as the primary metric to evaluate the capability of APR tools. Specifically, a patch is considered plausible if the patched program, when run with the proof-of-concept (PoC) input, does not crash. However, this metric is insufficient. We summarize the potential issues with this approach in Table 1.

In particular, the current *plausible fix* metric only verifies that, for the given PoC input, the program is no longer exploitable. On the one hand, the program may still exhibit issues such as *early exit* or

Metric	Taking PoC as Input			Other Input
	Not Exploitable	No Early Exit	Same Results	
Plausible	✓	✗	✗*	✗
Same Execution	✓	✓	✓	✗
Manual Review	✓	✓	✓	✓

Table 1: Comparision between different metrics. *: WILLIAMT guarantees that any plausible patch yeild same results if not early exit.

different execution results when given the same PoC. On the other hand, other inputs may still bypass the patch and trigger the original vulnerability. This situation highlights the need for a more robust metric—one that ensures the program, whether executed with the PoC or with other inputs, is *no longer exploitable*, avoids *early exits*, and produces *consistent execution results*.

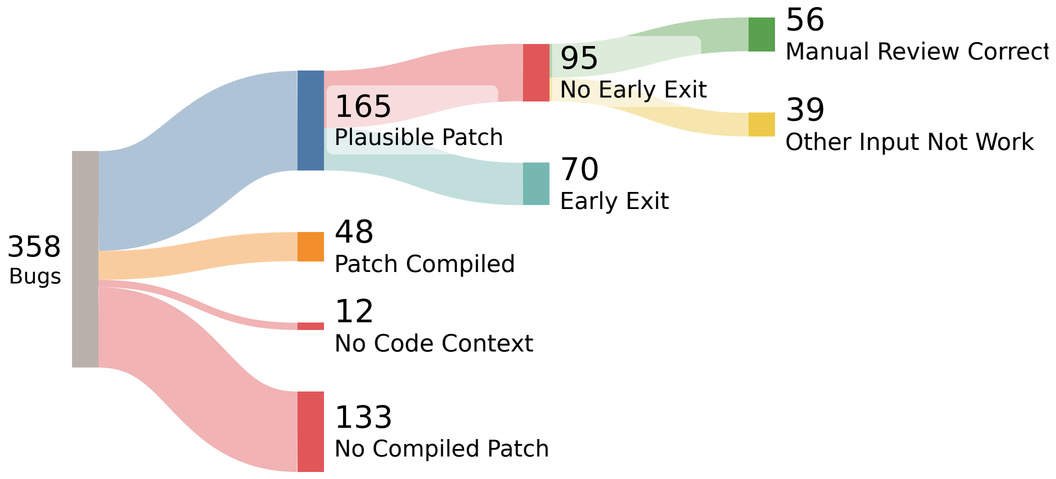


Figure 9: The actual fix ratio of WILLIAMT-GPT-4o.

To better understand this gap, we analyze the patches generated by WILLIAMT-GPT-4o as an example, manually evaluating all *plausible patches* as shown in Figure 9. Specifically, we first automatically validate whether the *plausible patches* maintain the same execution path as the unpatched program when using the PoC as input. We then manually verify the patches to assess their effectiveness against a broader range of inputs. We do not perform this verification for CodeRover-S or other state-of-the-art (SoTA) agents, as their patches often involve heavy modifications—such as code deletion or major rewrites—making automated validation infeasible.

Consistent Execution. WILLIAMT only inserts checks without deleting or rewriting existing code. This patching strategy ensures that if the patched program does not exit prematurely, it preserves the original functionality, meaning it follows the same execution path as the unpatched program. Therefore, to validate the patch, we only need to verify whether the PoC reaches the crash site at the same point in both the patched and unpatched programs. Figure 10 shows how we automate this PoC execution comparison by injecting a counter. We instrument both the patched and unpatched program source with the counter and compare the printed counter values in stdout. If the values are identical, it indicates that the patch does not introduce an early exit. Due to the nature of WILLIAMT, the patch guarantees the *consistent execution* as well.

Manual Review. However, this automated validation only ensures consistent behavior when using the PoC input. To further evaluate patch robustness, we manually review the patches that if their broader is correctly set, to determine their behavior with a broader set of inputs.

Successful Fixes in WILLIAMT. Figure 9 presents the results of our analysis. Among the 165 plausible fixes, only 95 patches avoid introducing early exits when tested with the PoC input. After

```
// insert a counter
printf("[SYS_INFO] hit crashsite for %d times\n", counter ++);
// optional: the patch may insert a check there.
stop_spatial(br->buffer, malloc_usable_size(br->buffer),
             &br->buffer[cwords]);
// actual crash site
b = br->buffer[cwords] << br->consumed_bits;

```

Figure 10: Example to automating the PoC execution comparision.

manually inspecting these patches, we find that 56 of them also avoid early exits across different inputs, while the remaining 39 may block some valid inputs (*i.e., Early Exit*). Nevertheless, all patches preserve program functionality as long as the program continues execution, which highlights a major advantage of WILLIAMT.

Takeaway: Developer need to manually verify the fix, even its classified as plausible.

B The Vulnerability Fixing Template

Template-guided patch generation is based on a set of predefined vulnerability templates. To construct these templates, we first analyze the distribution of bug types uncovered in real-world fuzzing campaigns, and subsequently select the top four categories to guide our template development.

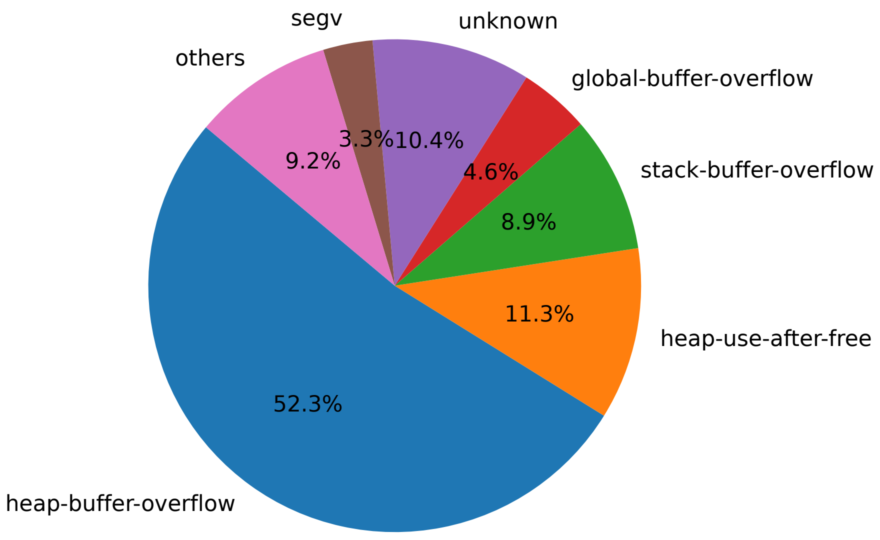


Figure 11: Exploitable Bug Types Distribution in ARVO. We merge the bug categories less than 3% into 'others'.

We begin by examining the distribution of bug types among real-world vulnerabilities discovered through OSS-Fuzz, using the ARVO dataset [30]. ARVO comprises over 5,000 bugs identified by Google's OSS-Fuzz [16], providing a reliable and representative ground truth for our analysis. Our focus is on AddressSanitizer (ASan) bugs, as they remain the most exploited class of vulnerabilities

```c
void stop_temporal(void *ptr) {
    // temporal memory corruption happens
    if (is_destroyed(ptr)) {
        exit(0);
    }
}
// vulnerable code example
free(ptr);
...
+ stop_temporal(ptr);
memcpy(dst, ptr, size);
```

Figure 12: Repairing Template For Temporal Memory Corruptions.

```c
void stop_spatial(void *buf, size_t buf_size, void * ptr) {
    // spatial memory corruption happens
    if (ptr >= buffer + buf_size || ptr < buf) {
        exit(0);
    }
}
// vuln code example
for (int idx = 0; i < size; i ++) {
    // Global/Stack-Buffer-Overflow: crash site
+   stop_spatial(buf, sizeof(buf), &buf[i]);
    // Heap-Buffer-Overflow: crash site
+   stop_spatial(buf, malloc_usable_size(buf), &buf[i]);
    result_buf[i] = buf[i];
}
```

Figure 13: Repairing Template For Spatial Memory Corruptions.

in recent in-the-wild 0-day attacks [43]. Figure 11 presents the distribution of ASan bug types within ARVO.

Overall, Heap Buffer Overflow (HBO) emerges as the most prevalent ASan vulnerability, accounting for 52.3% of all ASan reports. Following this, Heap Use-After-Free (UAF) is the most common form of temporal memory corruption, representing 11.3% of the reports. Stack Buffer Overflow (SBO) and Global Buffer Overflow (GBO) contribute 8.9% and 4.6% of the cases, respectively. We also observe instances labeled as "Unknown" or "Segv," often caused by out-of-bounds accesses that occur far from ASan redzone, making precise classification difficult.

Based on this analysis, we identify HBO, UAF, SBO, and GBO as the most representative ASan bug categories. Among them, HBO, SBO, and GBO are classified as spatial memory corruptions, while UAF represents a temporal memory corruption. Consequently, we develop templates that address both spatial and temporal memory error classes.

Temporal Memory Corruption Template. To mitigate *temporal memory corruption*, the template `stop_temporal` is introduced. This template takes a pointer currently in use by the program and checks whether it has already been deallocated. The `is_destroyed` varies depending on the memory allocator used. Figure 12 illustrates an example in which WILLIAMT inserts a call to `stop_temporal (ptr)` immediately before the crash site, thereby preventing potential UAF exploitation.

Spatial Memory Corruption Template. To address *spatial memory corruptions*, the corresponding template requires three arguments: `buf`, the base address of the accessed buffer; `buf_size`, the size of the buffer; and `ptr`, the pointer performing the access that may be out-of-bounds. An example is shown in Figure 13. This template is applicable to SBO, GBO, and HBO. The main difference lies in how the buffer size is determined: for SBO and GBO, `sizeof` is used, while for HBO, `malloc_usable_size` is required to retrieve the dynamically allocated buffer's actual size.

```
==19==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000071da0a8 at
    pc 0x000001b03a92 bp 0x7ffdb1079ef0 sp 0x7ffdb1079ee8
READ of size 8 at 0x0000071da0a8 thread T0
SCARINESS: 33 (8-byte-read-global-buffer-overflow-far-from-bounds)
    #0 0x1b03a91 in wassp_match_strval /src/wireshark/epan/dissectors/packet-wassp.c
    :4384:32
    #1 0x1b03a91 in dissect_wassp_sub_tlv /src/wireshark/epan/dissectors/packet-
    wassp.c:4779
    #2 0x1b0348a in dissect_wassp_sub_tlv /src/wireshark/epan/dissectors/packet-
    wassp.c
    #3 0x1b06a96 in dissect_wassp_tlv /src/wireshark/epan/dissectors/packet-wassp.c
    #4 0x1b084b5 in dissect_unfragmented_wassp /src/wireshark/epan/dissectors/packet
    -wassp.c:5873:12
    #5 0x1b084b5 in dissect_wassp /src/wireshark/epan/dissectors/packet-wassp.c:6021
...
```

Figure 14: AddressSanitizer Error Report for bug 20004.

C The Regex and Prompt Preparation

This section introduce some technical details on how our *regex-based context retrieval* works and how we prepare the LLM prompt for the *crash site analysis*.

Regex-Based Context Retrieval. WILLIAMT takes AddressSanitizer error reports [40] as input, which follow a well-defined format [37]. To process these reports, WILLIAMT implements a *regex-based context retrieval* module. This module begins by extracting the pattern prefixed with `"AddressSanitizer: "` to determine the bug category. Depending on whether the bug is temporal or spatial in nature, WILLIAMT further extracts the crash and allocate call stack, omitting entries associated with library code (*e.g.,* frames containing `"compiler-rt"` in the file path). It then identifies the last stack frame corresponding to the user program and retrieves the associated source file and line number where the crash occurred. Finally, WILLIAMT extracts a window of code context, consisting of two lines before and after the crashing line in the identified source file. Figure 14 illustrates the stack trace of bug 20004. In this example, WILLIAMT correctly identifies the bug as a `global-buffer-overflow`, locates the crashing source file as `packet-wassp.c`, and extracts lines 4382 through 4385 as the surrounding context.

Prompt Preparation. Building on the retrieved context information, WILLIAMT performs *crash-site analysis* by leveraging a predefined prompt in conjunction with the contextual code. Specifically, as illustrated in Figure 15, WILLIAMT first issues a structured prompt and then embeds the extracted context within an `<issue>` tag to facilitate LLM analysis.

```
You are a software developer maintaining a large project.
You are working on an issue submitted to your project.
The issue contains a description marked between <issue> and </issue>.
Another developer has already collected code context related to the issue for you.
Your task is to find potential root cause for the issue by answer the following
questions.

The bug is a global-buffer-overflow and should be described in the following:
  - g_buffer: the global buffer that program trying to access, format is (void *)
  - g_buffer_size: the end of the global_buffer, presenting as sizeof($g_buffer)
  - g_ptr: the address of element visit when the program crash, format is (void *)

REMEMBER:
- Output in ‘‘‘json ‘‘‘ format.
- for g_ptr, ONLY USE VARIABLES THAT EXISTS IN <context> ... </context>
- for g_buffer, USE VARIABLES EITHER IN <context> ... </context> or <log> ... </log>
- for g_buffer, if <log_type> indicate its a struct, use (void *) ($g_buffer) to
    represent, else use (void *) ($g_buffer)
- DO NOT USE ANY INTERGER in the answer.

For example, given the following input:
// example issue

You are suppose to return:
// example output

Here are your inputs:
<issue>
    <type>global-buffer-overflow</type>
    <crash_location>
    line 4384, column 32</crash_location>
    <log_type>array</log_type>
    <log>XXX is located 8 bytes to the right of global variable ’tlvMainTable’</log>
    <context>
    4373: static const char* wassp_match_strval(...)
    4374: {
    // ...
    4382:   }
    4383:
    4384:   return in_ptr->entry[in_type].name;
    4385: }
    4385: }
    </context>
</issue>
```

Figure 15: LLM Prompt to analyze bug 20004.