TLR: Codebase-Level C Memory Management Error Repair with Large Language Models

XIAO CHENG*, Macquarie University, Australia
ZHIHAO GUO*, University of Technology Sydney, Australia
HUAN HUO, University of Technology Sydney, Australia
YULEI SUI, University of New South Wales, Australia

Memory management errors in C remain a leading source of software vulnerabilities due to the inherent complexity of manual memory handling. Traditional Automated Program Repair (APR) largely relies on rule- or template-based techniques, which require expert-crafted specifications and often struggle to generalize. Recently, Large Language Models (LLMs) have emerged as a complementary approach, leveraging broad exposure to codebases and programming idioms to synthesize fixes that can extend beyond existing templates and rules. This paper introduces TLR, a novel framework that augments LLM-based repair with typestate-guided context retrieval. By using a finite typestate automaton to track error-propagation paths and memory state transitions, our approach provides the LLM with focused, semantically rich context for codebase-level memory error repair, effectively addressing both interprocedural reasoning and LLM context window limitations. Our framework has successfully repaired 37 out of 49 real-world memory errors derived from 14 open-source projects that collectively comprise approximately 1.57 million lines of code. Compared to state-of-the-art memory error APR tools, SAVER and ProveNFix, our approach correctly fixes 14.50× and 2.36× more errors, respectively; and on the double-free and use-after-free subset, TLR repairs all 7 cases whereas the crash-constraint-driven CrashRepair repairs only 1. Moreover, TLR outperforms current open-source state-of-the-art LLM-based repair tools, repairing more errors than SWE-agent 1.0 and the tree-of-thought agent Sand2Patch, while introducing far fewer harmful patches. We have also successfully repaired three critical zero-day memory errors, with fixes that have been accepted and implemented by the original developers. These results highlight a promising paradigm for codebase-level program repair through program analysis-guided, retrieval-augmented LLMs, combining formal verification strengths with neural model adaptability.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Automated program repair, memory errors, typestate, LLM.

ACM Reference Format:
Xiao Cheng, Zhihao Guo, Huan Huo, and Yulei Sui. 2026. TLR: Codebase-Level C Memory Management Error Repair with Large Language Models. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE057 (July 2026), 24 pages. https://doi.org/10.1145/3797085

1 Introduction

Memory management errors (hereafter, memory errors) in C are pervasive and severe, enabling zero-day exploits that cause data corruption, denial of service, and information leakage [6, 8, 10,

*Both authors contributed equally to this research.

Authors' Contact Information: Xiao Cheng, Macquarie University, Macquarie Park, NSW, Australia, xiao.cheng@mq.edu.au; Zhihao Guo, guodududu@gmail.com, University of Technology Sydney, Sydney, NSW, Australia; Huan Huo, huan.huo@uts.edu.au, University of Technology Sydney, Sydney, NSW, Australia; Yulei Sui, y.sui@unsw.edu.au, University of New South Wales, Sydney, NSW, Australia.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE057
https://doi.org/10.1145/3797085

37, 38, 62, 67, 70, 77, 79, 81]. C's low-level semantics and complex object lifetimes make manual detection and repair difficult; consequently, automated program repair (APR) has become a key research topic in software engineering over the past decade [27, 35, 36, 48].

Traditional APR tools fundamentally depend on predefined repair templates and expert-crafted heuristics [25, 29, 38, 42, 43, 51, 63, 69], making rule creation labor-intensive and limiting repair effectiveness to the scope of human-specified rules. Even when memory errors are reproducible and their root causes are known, effective repair remains difficult without a thorough understanding of codebase-level context and program semantics [29]. For example, fixing memory leaks may require careful deallocation of complex data structures, while resolving use-after-free errors often involves analyzing pointer aliasing and restructuring control flow (see Figures 2 and 7). These challenges demand not only knowledge of memory management but also a holistic grasp of program semantics to ensure correctness beyond simply removing the immediate error.

Large Language Models (LLMs) have recently shown promise as a valuable complement to traditional rule-based approaches for automated program repair, particularly when the root cause of a bug has already been identified. Rather than replacing established static or heuristic-based methods, LLMs can offer a fresh perspective by leveraging their broad exposure to diverse codebases and programming patterns. This enables them to synthesize previously unseen repair strategies that may not be captured by existing templates or rules. While prior LLM-based approaches [32, 33, 73–76, 90] have demonstrated effectiveness in general bug-fixing tasks, their potential to enhance memory error repair remains underexplored.

Memory errors frequently manifest as interprocedural phenomena requiring comprehensive semantic understanding across entire codebases—a complexity that exceeds the capabilities of current approaches primarily focused on localized fixes within isolated functions or files. To illustrate this challenge empirically, consider a use-after-free error (detailed in Section 3). This error manifests across six distinct functions, with a correct repair requiring synchronized modifications at three separate locations. In our experimental evaluation, we found that LLMs, even when provided with both the error-triggering function and its complete calling chain alongside a precise error root cause analysis, consistently failed to generate semantically correct patches. This limitation stems fundamentally from the inherent complexity of codebase-level memory error repair, which demands a comprehensive understanding of long-range memory management contexts and interprocedural dependencies. This presents a fundamental challenge for LLMs, which are constrained by token limitations and exhibit performance degradation with extensive prompts—a phenomenon known as "lost in the middle" [40].

Drawing inspiration from established developer practices for memory error resolution, we propose an approach that methodically emulates this systematic debugging process. When addressing memory errors, developers typically follow three essential steps: 1) reproducing the error through test cases and specialized program analysis tools such as ASan [58]; 2) deploying debugging utilities like GDB [20] to establish strategic breakpoints and comprehensively analyze program dependencies and execution context; and 3) applying domain-specific knowledge of safety specifications to implement semantically robust fixes. Based on these observations, we present TLR, a novel LLM-based approach for automatic memory error repair in C programs that leverages context-aware retrieval augmented by typestate analysis [66]. Our architecture positions the LLM as a reasoning engine that operates synergistically with established program analysis techniques and debugging infrastructure to facilitate comprehensive error comprehension and correction. This integration effectively bridges the gap between LLMs' linguistic capabilities and the specialized contextual understanding required for effective memory error repair.

We formalize memory error semantics through tracking the error-propagation path and monitoring program context transitions. The propagation path encodes the temporal execution history

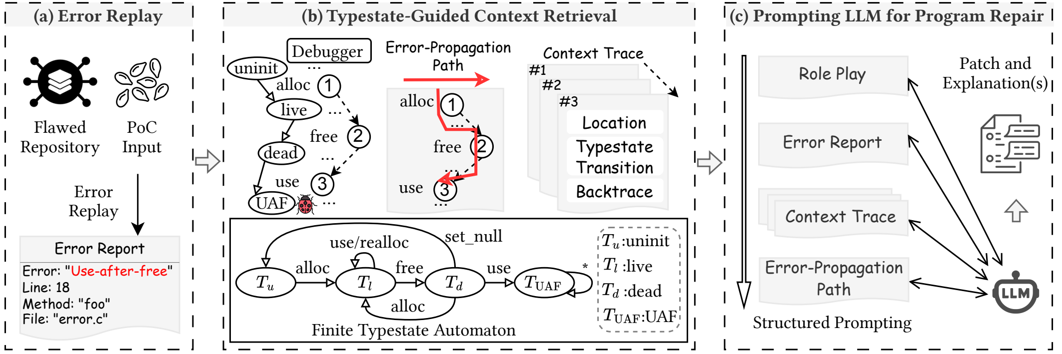

Fig. 1. An overview of our framework.

leading to error manifestation, while the context at each critical program point encapsulates both memory management states and detailed backtrace information of interprocedural calling chains. This comprehensive representation enables LLMs to model complex state evolution patterns across function boundaries by preserving both spatial (memory states) and temporal (execution history) dimensions of error behavior. The context collection is governed by a finite typestate automaton (FTA) [5, 11, 13, 17, 66]. Rather than exhaustively capturing contexts at every execution point—which would overwhelm LLMs' context windows—the FTA guides the collection process by monitoring object lifetime phases from allocation to error manifestation, triggering context snapshots exclusively at semantically significant state transitions. This selective approach generates a concise yet semantically rich context trace that preserves essential memory management semantics while enabling LLM-based repair to scale effectively to codebase-wide analysis.

Figure 1 provides an overview of our framework consisting of three phases:

(a) Error Replay. The input to this framework is a flawed codebase and its proof of concept (PoC) input, which then undergo a preliminary analysis that uses a dynamic analysis tool to replay the memory error and pinpoint the error-triggering location and the specific error type.

(b) Typestate-Guided Context Retrieval. Guided by a finite typestate automaton (FTA), we use a debugger to execute the program step-by-step, extracting the error-propagation path from the nearest related memory allocation to the error-triggering point. We then construct a context trace of the memory error, starting from the memory allocation and following each memory operation across the program. The context tracing is guided by the FTA, allowing for efficient tracking of program contexts only at typestate-changing breakpoints. Each context includes three elements: the location of the current point, the typestate transition, and the backtrace of the calling stack gathered at the current breakpoint.

(c) Prompting LLM for Program Repair. Finally, we design a multi-step structured prompting method that incrementally deliver role and task description [59], error report, context trace and error-propagation path to the LLM for generating an appropriate patch and explanation(s).

Our major contributions are as follows:

- We present TLR, the first codebase-level memory error repair system combining LLMs, context-aware retrieval, and typestate-guided analysis. Unlike prior APR methods, it leverages debuggers and formal verification to derive correct, semantics-aware patches.
- We introduce a typestate-guided retrieval method that captures critical context during typestate transitions, enabling LLMs to reason about memory error semantics while focusing on concise, relevant code, thereby improving repair accuracy and efficiency.

Table 1. Finite typestate automata of use-after-frees (UAF), double-frees (DF) and memory leaks (ML).

$\mathcal{A}_{\text{UAF}} = \langle \Sigma, \mathbb{T}, T_u, \delta, T_{\text{UAF}} \rangle$	$\mathcal{A}_{\text{DF}} = \langle \Sigma, \mathbb{T}, T_u, \delta, T_{\text{DF}} \rangle$	$\mathcal{A}_{\text{ML}} = \langle \Sigma, \mathbb{T}, T_u, \delta, T_{\text{ML}} \rangle$
$\mathbb{T} = \{T_u, T_l, T_d, T_{\text{UAF}}\}$ $\Sigma = \{$alloc, free, use, realloc, set_null$\}$	$\mathbb{T} = \{T_u, T_l, T_d, T_{\text{DF}}\}$ $\Sigma = \{$alloc, free, realloc, set_null$\}$	$\mathbb{T} = \{T_u, T_l, T_d, T_{\text{ML}}\}$ $\Sigma = \{$alloc, free, realloc, exit$\}$
T_u, T_l, T_d, T_{UAF}; alloc, use/realloc, free, use, set_null, alloc, *	T_u, T_l, T_d, T_{DF}; alloc, realloc, free, free, set_null, alloc, *	T_u, T_l, T_d, T_{ML}; alloc, realloc, free, alloc, exit, *
T_{UAF}: Use-after-free error state.	T_{DF}: Double-free error state.	T_{ML}: Memory leak error state.
T_u: Uninitialized state (memory object is not yet allocated); T_l: Live state (memory object is allocated and in use); T_d: Dead state (memory object is released).		
use: Use a heap object; set_null: Set the pointer pointing the heap object to null.		exit: Return from main function
alloc: Allocate a heap memory object; realloc: Reallocate a heap memory object; free: Free a heap object;		

- We build a large-scale memory error database of 49 errors, PoCs, and fixes across 14 projects (> $1M$ LOC). TLR repairs 37 errors, outperforming existing tools, and resolved three zero-day bugs with developer-accepted fixes.

2 Preliminaries and Problem Formulation

2.1 Memory Errors and Typestate Analysis

Memory errors in C programming often stem from improper memory management, necessitating careful tracking of memory's temporal properties along the program's control flow. Typestate analysis [5, 13, 17, 66] emerges as an effective method for detecting and understanding these errors, as it monitors execution logic by tracking the temporal state changes of memory objects. This approach represents different states of a given memory object and their transitions using a finite typestate automaton (Definition 1), allowing for precise modeling of memory object lifecycles.

Definition 1 (Finite Typestate Automaton)**.** A finite typestate automaton (FTA) for an error ET is a quintuple denoted as $\mathcal{A}_{\text{ET}} = \langle \Sigma, \mathbb{T}, T_u, \delta, T_{\text{ET}} \rangle$. The language Σ signifies the operations (e.g., function calls) that can be performed on the typestates. $\mathbb{T}$ encompasses all the possible typestates, with $T_u \in \mathbb{T}$ representing the initial state. $\delta : (\mathbb{T} \times \Sigma) \rightarrow \mathbb{T}$ is the state-transition table encoding the effects of operations in Σ. T_{ET} is the error typestate indicating a potential error detected. For a program statement s, we use $\texttt{op}(s)$ to retrieve its corresponding operation in Σ.

In this paper, we focus on three critical yet difficult-to-fix memory errors [29]: use-after-frees [47], double-frees [46], and memory leaks [45]. Table 1 presents the specifications of these errors in the form of finite typestate automata. The analysis process begins with an uninitialized typestate, denoted as T_u, and then may advance through different typestate transitions depending on the program statements encountered. For example, T_u transitions to T_l upon encountering a memory allocation statement, such as the primitive heap allocation API `malloc`. If released memory (T_d) is used or freed again, it transitions to T_{UAF} and T_{DF}, representing a use-after-free or double-free error as per $\mathcal{A}_{\text{UAF}}$ and $\mathcal{A}_{\text{DF}}$ respectively. Similarly, a live heap memory object (T_l) transitioning to T_{ML} at program exit as per $\mathcal{A}_{\text{ML}}$ indicates a memory leak.

2.2 Problem Formulation

Our objective is to automatically generate a patch to repair a flawed C repository with memory errors by leveraging the power of LLM and the prompts constructed based on FTA specifications (Table 1). The patch should fix the memory error and not introduce new bugs. Formally, let P be the original flawed C repository (the error type is ET) and I be a specific proof of concept (PoC)

input. We generate a set of prompts Q via the function $\mathbf{M}$:

$$Q = \mathbf{M}(P, \mathcal{A}_{\text{ET}}, I)$$

The LLM's role is to take Q to generate a correct patch ΔP:

$$\Delta P = \text{LLM}(Q)$$

The patch ΔP, when applied to P, should yield a updated repository $P' = P + \Delta P$ that fixes the error without introducing new bugs.

To ensure the patch's effectiveness and reliability, we impose these constraints:

(1) The patched repository P' should not exhibit the memory error when executed with I; (2) Let $\mathcal{I}$ represent the available test-suite for the repository. The patched repository P' should not introduce new bugs, meaning that for any test case in the provided test-suite $I' \in \mathcal{I}$, the execution should produce correct results.

We formulate our memory error APR problem as follows:

Given a flawed C repository P and a PoC input I, we design a method $\mathbf{M}$ to construct prompts Q that guide the LLM to generate a patch ΔP. The goal is to ensure that the patch ΔP, when applied to P, fixes the memory error for I and maintains correctness for its test-suite $\mathcal{I}$.

3 A Motivating Example

Figure 2 illustrates the pipeline of TLR, walking through the three phases depicted in Figure 1. These phases are demonstrated using a use-after-free error. A heap memory object is initially allocated by the `create_context` function, which wraps a `malloc` call at Line 27 in the `test.c` file (①). This memory is freed by invoking the `release_context` method, which calls a `free` function at Line 37 in `test.c` (②). However, the released memory is erroneously used in the method `clone_data` at Line 21 in `test.c` (③). This use-after-free error traverses six functions from allocation (①) to the error-triggering point (③).

Challenges. The successful repair of this memory error poses four fundamental challenges. First, it demands a thorough understanding of the allocated memory structure (`Context`), necessitating proper deallocation and null pointer validation mechanisms for both the base object (`Context`) and its associated fields (`data`). Second, the repair requires careful restructuring of operation sequences to maintain program correctness, particularly ensuring that `release_context` executes after its dependent operation `copy_ctx`, thus preventing critical logic in `copy_ctx` from being invalidated by premature null pointer checks. Third, the fix requires semantic comprehension of the codebase to implement a deep copy operation using `memcpy` for `src`, rather than a potentially unsafe shallow copy of `src->data`. Finally, the repair must correctly handle interprocedural interactions along the error-propagation path, ensuring consistent memory management across function boundaries.

State-of-the-art memory error APR tools, including ProveNFix [63] and SAVER [29], fail to repair this memory error because of insufficient modeling of the hierarchical structure of the `Context` object, consequently implementing only superficial null checks on `src` rather than addressing the whole structure. Their repair strategies further exhibit semantic miscomprehension by introducing premature returns after the `release_context` call which, while eliminating the immediate error, prevents the execution of subsequent critical operations and compromises program integrity. The LLM-based approach SWE-agent 1.0 [82] similarly fails to generate an appropriate patch due to its inability to identify and reason about the interprocedural execution logic of this error.

Our Solution. In Phase (a), the use-after-free error is replayed and confirmed. Phase (b) identifies the relevant memory operations according to the finite typestate automaton $\mathcal{A}_{\text{UAF}}$ in Table 1, extracting the error-propagation path and relevant program contexts. Phase (c) feeds the error

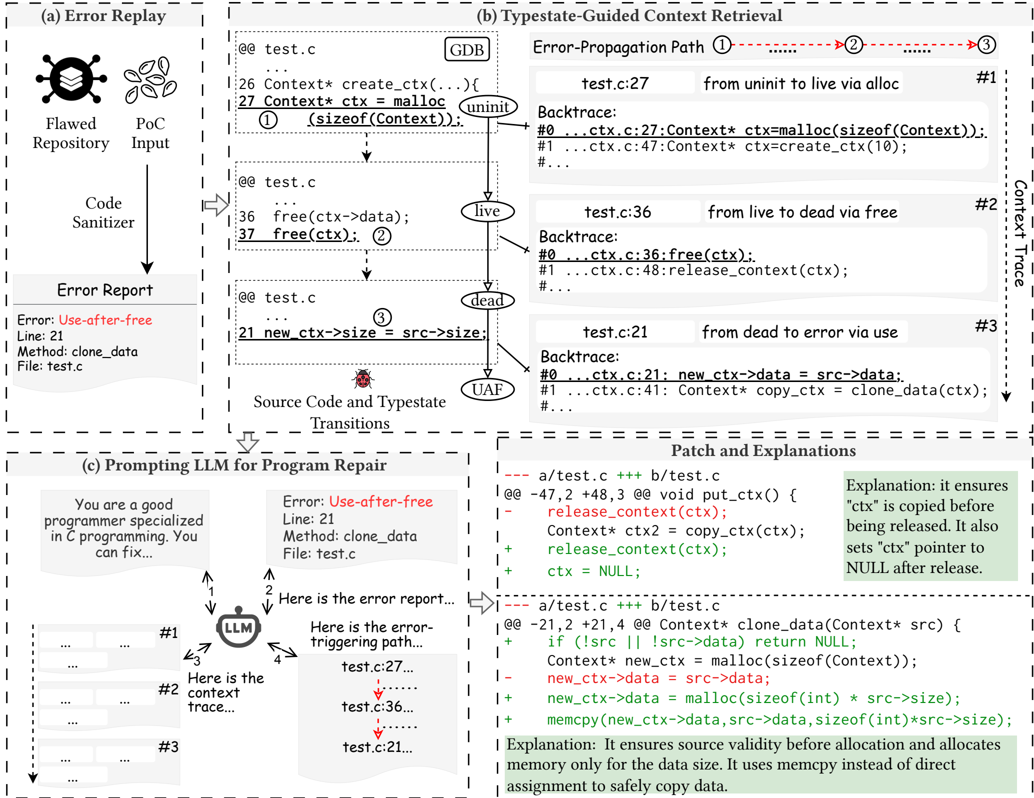

Fig. 2. A motivating example illustrating how TLR repairs a use-after-free error.

report, context trace, and error-propagation path from the previous phases into the LLM, which infers the correct patch and provides clear explanations for its decisions, as shown at the bottom of Figure 2. This solution aligns precisely with the ground truth that eliminating this error while preserving logical correctness of the project.

(a) Error Replay. Understanding the root cause of a memory error often requires replaying the error. As illustrated in Figure 2(a), we reproduce this error using its proof of concept (PoC) input, which is a specific assembly file. We employ Valgrind [50], a widely-used dynamic analysis tool, to capture the core dump of the PoC input at the moment the error is triggered—specifically, at Line 21 in the `test.c` file. This core dump provides a snapshot of the memory state at the time of the crash. Based on the core dump, we generate an *error report* that offers the specific error type and error location to the LLM.

(b) Typestate-Guided Context Retrieval. We employ a finite typestate automaton ($\mathcal{A}_{\text{UAF}}$ in Table 1) to facilitate the extraction of the *error-propagation path* using the GDB debugger [20]. This path begins with the memory allocation at ①, proceeds through the memory free at ②, and ultimately stops at the error-triggering point at ③, as indicated by an error typestate.

The key transitions in this model are triggered by the `malloc`, `free`, and `use` statements, which occur at ①, ②, and ③, respectively. Upon invoking the `malloc` function, denoted as the `alloc` operation in our FTA, the typestate of the memory object shifts to `live`, indicating that the memory is currently in use. Subsequently, when the `free` API is called, the typestate changes to dead, signifying that the memory has been released. However, if the same memory object is used after it is released, the typestate changes to `error`, indicating a use-after-free error. At each typestate

change point, we extract its associated context, which includes the typestate transition, the location, and the backtrace. The program contexts at the three typestate change points collectively form a *context trace*. For instance, in the final error context, the typestate transition indicates a shift from `dead` to `error` due to a `use` operation. The location, `test.c:21`, specifies the file path and line number. The backtrace, gathered from the debugger at this breakpoint, provides the call stack details, illustrating how the `use` operation was invoked by `clone_data` and other higher-level callers in the codebase.

(c) Prompting LLM for Program Repair. In this phase, depicted in Figure 2(c), we employ structured prompting [28] to break down the prompts into four structured steps: role and task description [59], error report, context trace, and error-propagation path. These segments of information are systematically fed to the LLM step by step. This method effectively deconstructs the task of program repair into increasingly detailed and specific stages, allowing the LLM to progressively comprehend and tackle the error.

4 TLR Approach

In this section, we detail our TLR approach. We first identify the specifics of the memory error (Section 4.1), then utilize typestate-guided context retrieval to understand its semantics (Section 4.2). Finally, we leverage this information to generate a patch and explanation(s) via LLM (Section 4.3).

4.1 Error Replay

In this paper, our emphasis is not on detecting memory errors but on repairing validated true errors. The first step of our approach involves reproducing the error and identifying the associated bug report, which includes the type of memory error and its location. To achieve this, we utilize a dynamic analysis tool (DAT) such as ASan [58] or Valgrind [50] during compilation and generate debuggable files following the DWARF 5 [14] standard. We then use specific proofs of concept (PoCs) to reproduce the error and employ DAT to generate detailed error reports. At the error-triggering point, the DAT outputs information about the type of memory error, the location, and the specific memory address that is erroneously accessed (i.e., the error address).

4.2 Typestate-Guided Context Retrieval

This section elaborates on how we enhance LLM prompts with a deep understanding of memory error semantics while optimizing token usage. We achieve this by employing typestate finite state automata (Definition 1) to guide the derivation of a comprehensive yet precise error-propagation path (Definition 5) and context trace (Definition 6) that capture the interprocedural contextual evolution linked with the memory error.

We first demonstrate how to construct the full execution path of the error (Definition 2) and the typestate-changing context map (Definition 4) using Algorithm 1. The context map records the program context (Definition 3) specifically at the statements introducing typestate changes. Using the full path and context map as a foundation, we then illustrate how to build the error-propagation path as explained in the rule in Figure 4 and the context trace as outlined in Figure 5.

Definition 2 (Full Execution Path π)**.** A full execution path, denoted as $\pi = (s_i)_{i=1}^{n}$, is a chronologically ordered sequence of program statements, each assigned an index, such that $s_i = \langle \texttt{sc}(s_i), i \rangle$. Here, $\texttt{sc}(s_i)$ retrieves the source code associated with s_i, while i represents the position of this statement in the sequence, indicating the execution order. This sequence starts from the program's entry point and extends all the way to the error-triggering statement.

Note that for any s_i, s_j in π where $i \neq j$, their corresponding source code can be identical, i.e., $\texttt{sc}(s_i) = \texttt{sc}(s_j)$, because a single code line can be invoked or executed multiple times throughout the program's run, depending on the control flow of the program.

Definition 3 (Program Context *Ctx*)**.** Given a full execution path π, for any statement $s_i \in \pi$, the program context of s_i is defined as:

$$Ctx_{s_i} = \langle lc, tr, cp \rangle$$

where:

- *lc* denotes the program location (file path and line number) of s_i,
- *tr* represents the typestate transition at s_i, encapsulating the pre- and post-typestates and the memory operation, and
- *cp* signifies the backtrace of the call path, showing the call sequence leading up to the current point of execution. Specifically, it records each function call along with its source location and the corresponding source code at that location.

Definition 4 (Typestate-Changing Context Map *tMap*)**.** A typestate-changing context map *tMap* associates statements inducing typestate changes with their respective program contexts. Importantly, each memory object manipulated by the statements in *tMap*'s key set must be aliased with the object at the error-triggering point. This aliasing is evidenced by their shared error address.

Full Execution Path and Typestate-Changing Context Map Construction. Algorithm 1 outlines the construction of the full execution path (π) and the typestate-changing context map (*tMap*) using the GNU Debugger (GDB) [20]. The algorithm initializes GDB and runs the program with input I. Within the loop (Lines 7-19), it constructs π and *tMap* step-by-step until an error state occurs. Each loop iteration, representing a program execution step, appends the statement to π (Line 16), and collects data from the current stack frame. If an operation manipulates the error address and causes a typestate change (Lines 10-11), the algorithm inserts the program context, including the current source code line, typestate transition and backtrace, into *tMap* (Lines 13-14).

Example 1. Figure 3(a) shows a double-free error at ℓ_3 (invoked from ℓ_{15}). In this example, the program execution flows through 10 steps as follows:

$$\pi = (s_i)_{i=1}^{10} = (\langle \ell_9, 1\rangle, \langle \ell_6, 2\rangle, \langle \ell_{10}, 3\rangle, \langle \ell_{11}, 4\rangle, \langle \ell_{12}, 5\rangle, \langle \ell_6, 6\rangle, \langle \ell_{13}, 7\rangle, \langle \ell_{14}, 8\rangle, \langle \ell_{15}, 9\rangle, \langle \ell_3, 10\rangle)$$

where ℓ_n refers to the source code at Line n for this example. The process of typestate transition and the corresponding results are illustrated in Figure 3(b). The statements that induce typestate changes are as follows:

$$s_2 : \langle \ell_6, 2\rangle, \quad s_3 : \langle \ell_{10}, 3\rangle, \quad s_4 : \langle \ell_{11}, 4\rangle, \quad s_6 : \langle \ell_6, 6\rangle, \quad s_8 : \langle \ell_{14}, 8\rangle, \quad s_{10} : \langle \ell_3, 10\rangle$$

$s_7 : \langle \ell_{13}, 7\rangle$ does not belong to these statements because it operates on a different address from $addr_e$. Accordingly, we construct the typestate context map *tMap* as follows:

$$tMap = \{s_2 \mapsto Ctx_{s_2}, s_3 \mapsto Ctx_{s_3}, s_4 \mapsto Ctx_{s_4}, s_6 \mapsto Ctx_{s_6}, s_8 \mapsto Ctx_{s_8}, s_{10} \mapsto Ctx_{s_{10}}\}$$

where Ctx_{s_i} denotes the program context at statement s_i. The mapping ($s_n \mapsto Ctx_{s_n}$) represents the association between the statement s_n and its corresponding program context Ctx_{s_n}, as defined in Definition 4. For instance, the program context Ctx_{s_6}, depicted in Figure 3(c), includes the statement's location, detailed typestate transition information, and backtrace data. The backtrace clearly shows the call stack from the `main` function to the `xmalloc` function as well as the exact line of source code at each stack frame.

Algorithm 1: Full execution path and typestate-changing context map construction.

Input: $addr_e$: Memory address erroneously accessed (error address); P: Target program; I: PoC input; $\mathcal{A}_{\text{ET}} = \langle \Sigma, \mathbb{T}, T_u, \delta, T_{\text{ET}} \rangle$: Finite typestate automaton;

Output: π: Full execution path; $tMap$: Map from typestate-changing statements to program contexts

```
1  Function constructPi(addr_e, P, I, A_ET):
2      GDB.execute("file "+ P);                         // Set the program to execute
3      GDB.execute("set arg "+ I);                      // Set the program PoC input
4      GDB.execute("start");
5      π ← (); tMap ← {}; T ← T_u; i ← 1;
6      frame ← GDB.selected_frame();
7      while T ≠ T_ET do
8          ℓ ← frame.code_line;
9          s ← ⟨ℓ, i⟩;
10         if op(s) ∈ Σ ∧ frame.addr = addr_e then     // Whether the operation of s belongs to
            the FTA language and s manipulates the error address
11             if (T′ ← δ(T, op(s))) ≠ T then           // Typestate changes
12                 backtrace ← GDB.execute("backtrace");
13                 Ctx_s ← {frame.location, transition(T, T′, op(s)), backtrace};
14                 tMap ← tMap ∪ {s ↦ Ctx_s};// Record typestate-changing context in tMap
15                 T ← T′;
16         π ← π ∘ s;                                   // Append s to π
17         i++;
18         GDB.execute("step");
19         frame ← GDB.selected_frame();
20     return π, tMap;
```

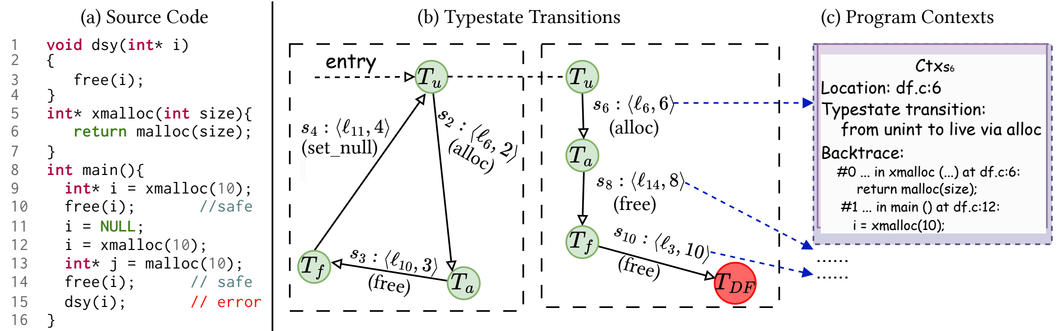

Fig. 3. An example of typestate transitions and program contexts.

Definition 5 (Error-Propagation Path π_e). An error-propagation path $\pi_e = (s_i)_{i=m}^{n}$ is a subsequence of indexed program statements that extends from the memory allocation statement s_m, where $0 \leq m < n \wedge \text{op}(s_m) = \text{alloc}$, to the error-triggering point s_n. s_m is the closest memory allocation to s_n on the full path π where the allocated memory address is the error address accessed at s_n.

Typestate-Guided Error-Propagation Path Extraction. The inference rule depicted in Figure 4 is used to extract the error-propagation path, π_e, from the full execution path, π (Definition 2). This rule helps reduce the code lines and isolate the error code from the executed code. It works

$$[\mathtt{ETP}]\ \frac{\pi = (s_i)_{i=1}^{n}\ s_m \in \mathsf{KS}(tMap) \wedge \mathsf{op}(s_m) = \text{alloc} \wedge (\forall k, m < k \leq n : s_k \notin \mathsf{KS}(tMap) \vee \mathsf{op}(s_k) \neq \text{alloc})}{\pi_e \leftarrow (s_i)_{i=m}^{n}}$$

Fig. 4. The inference rule for typestate-guided error-propagation path extraction. $\mathsf{KS}(tMap) = \{s_i \mid (s_i \mapsto Ctx_{s_i}) \in tMap\}$ represents the key set of *tMap*.

by identifying a particular statement, s_m, associated with an operation that changes the typestate allocation, guided by the *tMap*, which tracks these typestate changes (as per Definition 3). The rule ensures that any statements on the execution path π falling between s_m and s_n do not manipulate the error address or involve memory allocation. This isolates the statements likely to be the root cause of the error, similar to a debugging process that narrows down and focuses only on problematic code segments, effectively reducing the length of the message input into the large language model.

Example 2. The error-propagation path of Example 1 is:

$$\pi_e = (s_i)_{i=6}^{10} = (\langle \ell_6, 6\rangle, \langle \ell_{13}, 7\rangle, \langle \ell_{14}, 8\rangle, \langle \ell_{15}, 9\rangle, \langle \ell_3, 10\rangle)$$

The path starts from s_6, which is the nearest allocation operation preceding the error-triggering point s_{10} on π. Although $s_7 : \langle \ell_{13}, 7\rangle$ lies between these two points, it is not designated as the starting point because the variable j does not point to the error address. The operation at $s_8 : \langle \ell_{14}, 8\rangle$ does manipulate the error address, but it does not qualify as a starting point because it does not involve a memory allocation operation.

Definition 6 (Context Trace $\widetilde{Ctx_e}$)**.** The context trace of an error-propagation path π_e is formally defined as $\widetilde{Ctx_e} = (Ctx_{s_i} \mid s_i \in \pi_e \wedge \mathtt{Sel}(s_i))$, where each Ctx_{s_i} is the program context at s_i, and the sequence follows the order of statements in π_e. Without loss of generality, the selection function `Sel` is a predicate over the statements in π_e. In our definition, $\mathtt{Sel}(s_i)$ returns true if s_i introduces typestate changes of the memory object operated at the error-triggering point e.

$$[\mathtt{CXT}]\ \frac{s_i \in \pi_e \quad s_i \in \mathsf{KS}(tMap)}{\widetilde{Ctx_e} \leftarrow \widetilde{Ctx_e} \circ tMap[s_i]}$$

Fig. 5. The inference rule for typestate-guided context trace construction. $tMap[s_i]$ retrieves the program context related to s_i in *tMap*.

Typestate-Guided Context Trace Construction. Figure 5 shows the inference rule for constructing the context trace, denoted as $\widetilde{Ctx_e}$, which captures the program contexts associated with typestate-changing statements along an error-propagation path π_e. The rule iterates through each statement s_i in π_e from index m to n. For each statement s_i, if it is a typestate-changing operation, then the corresponding context is appended to $\widetilde{Ctx_e}$.

Example 3. Figure 6 presents the context trace of Example 2. By iterating through the error-propagation path, we append the relevant context information to the trace whenever a typestate-changing operation is encountered in *tMap*. The resulting context trace, depicted in Figure 6, consists of three program contexts, clearly showing the lifecycle of the memory, starting from its allocation, through its release, and finally to its erroneous access.

4.3 Prompting LLM for Error Repair

This section discusses how we utilize the information collected from the typestate-guided analysis to prompt LLM for efficient memory error repair. The process begins by employing the role-play technique [59] to define and motivate our task, followed by the use of structured prompting [28] to direct the LLM in crafting a patch and explanation(s).

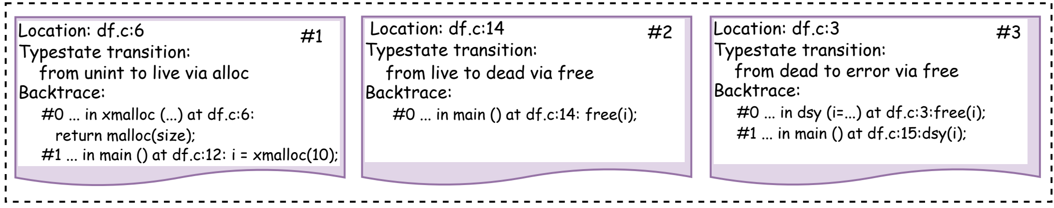

Fig. 6. An example of context trace by revisiting Example 1.

Role Play. Role play is an effective strategy known to enhance the precision of LLM outputs [59]. In our case, we configure the LLM to function as an APR tool, where it interprets error information from the typestate-guided analysis and generates an appropriate patch with intuitive explanations.

Structured Prompting. We structure prompts to guide the LLM in generating patches by providing the necessary information to know the error location, understand the error context and path, and then create a resolution. The error reports, along with the context trace and error-propagation path guided by the typestate analysis, are formatted as prompts and fed into the LLM.

The error report guides the LLM in pinpointing the exact location of the memory error in the codebase (Section 4.1). The initiation of the error report typically begins with a statement like, "Here is the location of the use-after-free error in the provided code snippet". The second component is the context trace $\widetilde{Ctx_e}$ (Definition 6) that captures the context sequence of memory state transitions leading to the error, enabling the LLM to comprehend the execution logic of the error. The final component presents the error-propagation path π_e (Definition 5) to the LLM. This path, a critical subsequence of the full execution path π, extends from the nearest memory allocation statement to the error-triggering point on π, showing the program dependencies along an intact erroneous memory management. Upon presenting all the necessary information for understanding the error semantics and triggering logic, the LLM is then tasked with generating a patch to repair the error.

5 Evaluation

This section evaluates TLR's performance in repairing memory errors in real-world projects by comparing it with two state-of-the-art memory error APR tools: SAVER [29] and ProveNFix [63]. TLR successfully repairs 14.50× and 2.36× more memory errors than SAVER and ProveNFix, respectively, while introducing no new errors. We also compare TLR with LLM-based baselines and conduct an ablation analysis to understand the contribution of each component.

5.1 Datasets and Implementation

Datasets. We first compare TLR with SAVER and ProveNFix using the same dataset as SAVER [29]. We meticulously reverse-engineer the vulnerability triggering conditions from SAVER's vulnerability set and construct proof of concept inputs to reproduce identical errors. From the original collection, we exclude multi-threaded programs because our debugger-based approach cannot effectively trace typestate transitions across multiple execution contexts, resulting in eight projects for comparative evaluation. However, we identify limitations in SAVER's dataset, including its homogeneity and insufficient representation of real-world vulnerabilities. For instance, the same version of `recutiles` [22] contains CVE-2019-6455 [53], which is absent from SAVER's dataset. Furthermore, fixes in SAVER's dataset rely solely on manual verification without the crucial validation provided by developer acceptance in real-world scenarios.

To address these limitations, we further construct a comprehensive benchmark comprising 14 real-world, open-source C projects, encompassing approximately 1.57 million lines of code across diverse application domains including system utilities, network protocols, and media processing

Table 2. The statistics and description of the real-world projects used in the evaluation. LoC stands for lines of code. #File and #Error represent the number of files and memory errors, respectively.

No.	Project (version)	LoC	#File	#Error	Description
1	`ls_extended` (9d899c8) [15]	1,352	25	3	ls with coloring and icons
2	`xHTTP` (72f82d) [19]	1,493	6	1	HTTP server library
3	`tree` (v1.8) [21]	3,435	13	3	utility to display a tree view of folders
4	`chibicc` (90df7f) [56]	9,688	71	5	C compiler
5	`stb` (v2.8) [57]	12,076	2	3	single-file public domain libraries for C/C++
6	`scrot` (b5e5f0d) [68]	13,130	36	1	command line screen capture utility
7	`mjs` (b1b6eac) [9]	32,116	202	1	embedded JavaScript engine
8	`SmallerC` (b120a9c) [2]	58,535	510	14	C compiler
9	`MyHTML` (90a853e) [1]	63,617	168	2	Fast C/C++ HTML 5 Parser
10	`quickjs` (d378a9f) [16]	86,281	53	2	embedded JavaScript engine
11	`recutiles` (v1.8) [22]	92,000	757	5	tools and libraries to access recfiles
12	`wasm3` (139076a) [65]	111,616	696	4	WebAssembly interpreter
13	`Yasm` (ffbd22c) [55]	201,975	945	4	assembler
14	`radare2` (8644a29) [23]	879,785	2,953	1	reverse engineering framework
Total		1,567,099	6,437	49	

libraries. Table 2 presents detailed statistics and descriptions for these projects. All memory errors in our benchmark have been *confirmed by respective project developers*, with 9 vulnerabilities assigned CVE identifiers and others documented in official commit histories. To establish reliable ground truth for validating correct fixes, we collect patches approved and implemented by project maintainers. Additionally, we curate error-inducing test cases (proof of concept inputs) to consistently reproduce each error during evaluation.

Implementation. Our experiments are conducted on an Ubuntu 22.04 server with a 24-cores 5.60 GHz Intel CPU and 64 GB of memory. We employ Valgrind (version 3.18.1) [50] and ASan [58] for error replay (Section 4.1). We use the GDB Python API (with GDB version 12.1) and the GNU C Library (version 2.35) to perform typestate-guided context retrieval (Section 4.2). To reduce the time overhead for extracting error-propagation paths and context traces, we set the initial breakpoint at memory allocations instead of the program's entry point. We use Claude 3.5 Sonnet [4, 12] as our LLM (Section 4.3). For each prompt, we apply the chain-of-thought [72] method to encourage the model to think step-by-step, thereby generating coherent and logically consistent responses. Given the inherent randomness of LLMs, it is necessary to ensure the reliability of our results. Therefore, we conduct each experiment five times for every project. We only consider a result to be valid if it exhibits consistency in at least four out of the five runs. Patch generation for each bug is capped at 180 minutes. We adopt a rigorous two-stage verification protocol. First, every candidate patch is run against the project's native test suite. We then apply continuous fuzz testing based on AFL++ [18] to exercise the patched binaries, with each patch undergoing at least 50 fuzzing cycles to confirm the absence of regressions.

5.2 Baselines

We compare TLR with eight baselines spanning traditional state-of-the-art memory APR tools and LLM-based approaches.

SAVER and ProveNFix. To the best of our knowledge, SAVER [29] and ProveNFix [63] represent the current state-of-the-art in memory error APR, capable of addressing memory leaks, use-after-frees, and double-frees. Beyond repair, both systems also perform static, sound memory-error detection without executing the target program. In this evaluation, however, we focus exclusively on their repair functionality on reproducible memory errors.

Table 3. Comparing TLR with SAVER and ProveNFix using SAVER's dataset [29].

Project	*#E*	SAVER [29]	*#E*	ProveNFix [63]	**TLR**
`rappel` (ad8efd7)	1	1	1	1	1
`lxc` (72cc48f)	3	3	23	22	22
`WavPack` (22977b2)	1	0	12	12	12
`flex` (d3de49f)	3	0	4	4	4
`p11-kit` (ead7ara)	33	24	28	27	27
`recutils` (v1.8)	10	8	42	36	37
`snort` (v2.9.13)	16	10	42	13	18
`grub`	0	0	1	1	1
Total (Fixing Rate)	67	46 (68%)	153	116 (75.8%)	**122 (79.7%)**

We employ their open-source implementations for comparative analysis. These tools rely on detection results from the static analysis tool Infer [44]. To ensure fair comparison, we augment these tools with *precise error locations*, identical to those provided to TLR, to direct targeted repairs. Futhermore, to mitigate potential limitations in static analysis, we configure SAVER and ProveNFix with *the most advanced parameters* for Infer as described in their respective publications. This configuration encompasses whole-program analysis of the linked program, flow-sensitive analysis distinguishing control flows, and comprehensive header file parsing [29, 63].

CrashRepair. CrashRepair [61] is a symbolic, crash-constraint-driven APR tool that synthesizes patches for double-free and use-after-free errors from crash-triggering inputs via concolic execution and constraint solving. Unlike SAVER and ProveNFix, it operates on crash traces rather than static typestate analysis, making it a natural point of comparison for our trace-guided repair.

SWE-agent and San2Patch Existing LLM-based APR tools primarily target Java and Python bugs and predominantly address single-hunk program repairs [78]. We evaluate TLR against SWE-agent [82], an open-source state-of-the-art LLM-based approach that employs Chain-of-Thought reasoning and sophisticated prompt engineering to autonomously invoke tools for error repair. To ensure methodological consistency and experimental validity, we provide SWE-agent with identical reproducible errors, comprising comprehensive reproduction workflows, toolchains, proof-of-concept inputs, compiled error versions, and precise trigger and compilation commands. San2Patch [34] is an LLM-based APR tool that leverages tree-of-thought reasoning to generate patches.

TLR-F and TLR-M. To evaluate the capabilities of the base LLM used in TLR without any contextual retrieval from error trace, we implement two comparative baselines. The first baseline, TLR-F, incorporates the error report alongside program files directly implicated in the error-triggering point. The second baseline, TLR-M, extends this approach by including all functions identified in the error backtrace in addition to the error report. Both baselines employ the same structured role-play and prompting techniques delineated in Section 4.3 that are utilized in TLR, thus isolating the effect of context retrieval.

TLR-NT. To understand the relative contributions of typestate-guided context retrieval to the overall performance of TLR, we conduct comprehensive ablation studies using TLR-NT (TLR without context trace). TLR-NT omits the context trace in the prompts while maintaining all other components unchanged (such as bug report and error-propagation path), allowing us to measure the specific impact of typestate-guided context trace information.

5.3 Evaluation Metrics

We collect the number of patches generated by each APR tool, denoted as #Δ. The number of correct patches is represented as #$\Delta_{\checkmark}$. To be classified as correct, a patch must: 1) successfully fix the memory error; 2) preserve the expected outcomes for the project's test suite; and 3) be manually validated to align with the ground truth, adhering to the standards outlined in [39]. We use #$\Delta_{\text{✗}}$ to denote the count of patches introducing new errors, as confirmed through fuzzing tests [18]. Conversely, #Δ_{O} denotes the quantity of patches that, while failing to correct the error, do not introduce new ones. Finally, #$E_{\checkmark}$ represents the number of fixed errors. In terms of performance evaluation, a higher value for both #$\Delta_{\checkmark}$ and #$E_{\checkmark}$ suggests superior performance, as it implies a greater number of errors have been correctly repaired. Conversely, #$\Delta_{\text{✗}}$ should ideally be minimized to avoid the introduction of new errors. Although #Δ_{O} does not pose a severe risk of introducing new memory errors, a smaller value is still preferable. Note that we count memory errors by logical "blocks" rather than by individual pointers. For memory leaks, each distinct allocation of a data structure that fails to be deallocated is counted as a separate error, as each requires an independent deallocation operation. This block-based accounting methodology accurately reflects the granularity at which memory management operations must be applied in practice. Consequently, a single patch that properly deallocates a complex data structure may fix multiple memory error blocks simultaneously. Therefore, it is possible that the number of correct patches #$\Delta_{\checkmark}$ is smaller than the number of fixed errors #$E_{\checkmark}$.

5.4 Research Questions

We address the following research questions in our evaluation:

- RQ1 **Traditional APR:** How does TLR compare to SAVER and ProveNFix in fix rate and introduced errors?
- RQ2 **LLM baseline:** How does TLR perform vs. SWE-agent [82]?
- RQ3 **Ablation:** What is the contribution of typestate-guided context retrieval?

5.5 Comparison with Traditional APR Tools (RQ1)

5.5.1 Results. Table 3 summarizes results on SAVER's dataset: TLR fixes 122/153 errors and outperforms both baselines across all projects. Table 4 reports our 14-project benchmark: TLR fixes 37/49 errors (14.50× more than SAVER and 2.36× more than ProveNFix) while introducing no new errors. Per-project, TLR matches or exceeds both tools with near-zero error introduction. Table 5 further compares TLR with the symbolic crash-driven repair tool CrashRepair [61] on the double-free (DF) and use-after-free (UAF) cases from our dataset. Given the same crash reports, TLR correctly repairs all 5 double-free and 2 use-after-free errors, whereas CrashRepair repairs none of the double frees and only 1 use-after-free. This gap confirms that CrashRepair's crash-constraint-driven synthesis struggles with multi-hunk, aliasing-sensitive memory errors, while TLR's typestate-guided context retrieval captures the cross-procedural semantics required to synthesize correct fixes.

5.5.2 Analysis and Case Study. We showcase the superior performance of our tool compared to SAVER and ProveNFix through four typical code scenarios depicted in Figure 7.

Complex Data Structure. Figure 7(a) illustrates TLR's in-context repair capability for handling complex data structures. In this case, the structure `db` must be released via the structure-specific destructor `rec_db_destroy()`. SAVER and ProveNFix fail because they do not model, nor can they correctly release, *complex data structures with nested memory allocations*.

In contrast, TLR correctly invokes `rec_db_destroy(db)`, which recursively deallocates all internal components of `db` before releasing the top-level object. By analyzing *allocation-deallocation pairings*

Table 4. Comparing TLR with SAVER and ProveNFix using the 14 real-world projects in Table 2. For simplicity, we omit the project versions. $\#E$ denotes the number of memory errors.

		SAVER [29]					ProveNFix [63]					**TLR**				
Project	$\#E$	$\#\Delta$	$\#\Delta_{\checkmark}$	$\#\Delta_{\circ}$	$\#\Delta_{\times}$	$\#E_{\checkmark}$	$\#\Delta$	$\#\Delta_{\checkmark}$	$\#\Delta_{\circ}$	$\#\Delta_{\times}$	$\#E_{\checkmark}$	$\#\Delta$	$\#\Delta_{\checkmark}$	$\#\Delta_{\circ}$	$\#\Delta_{\times}$	$\#E_{\checkmark}$
`ls_extended`	3	1	1	0	0	1	1	1	0	0	3	1	1	0	0	**3**
`xHTTP`	1	0	0	0	0	0	1	0	1	0	0	1	1	0	0	**1**
`tree`	2	0	0	0	0	0	0	0	0	0	0	1	1	0	0	**2**
`chibicc`	5	0	0	0	0	0	2	1	0	1	3	3	2	1	0	**5**
`stb`	3	0	0	0	0	0	0	0	0	0	0	3	3	0	0	**3**
`scrot`	1	0	0	0	0	0	1	0	1	0	0	1	1	0	0	**1**
`mjs`	1	1	1	0	0	1	1	1	0	0	1	1	1	0	0	**1**
`smallC`	14	1	0	0	1	0	2	1	0	1	1	7	4	2	1	**12**
`MyHTML`	2	0	0	0	0	0	1	1	0	0	2	1	1	0	0	**2**
`quickjs`	2	0	0	0	0	0	0	0	0	0	0	2	2	0	0	**2**
`recutiles`	6	0	0	0	0	0	0	0	0	0	0	3	2	1	0	**4**
`wasm3`	4	0	0	0	0	0	0	0	0	0	0	1	1	0	0	**2**
`yasm`	4	0	0	0	0	0	0	0	0	0	0	2	1	1	0	**2**
`radare2`	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	**1**
Total	49	3	2	0	1	2	10	6	2	2	11	28	22	5	1	**37**

Table 5. Comparing TLR with CrashRepair [61] on our dataset.

Vulnerability Type	**Errors**	**TLR**	**CrashRepair [61]**
Double Free	5	5	0
Use-After-Free	2	2	1

along the context trace, TLR recognizes when generic deallocation routines are insufficient and recommends the appropriate domain-specific alternative. Moreover, in this example the allocation and deallocation occur in different functions. TLR's interprocedural trace analysis follows the object's lifetime across function boundaries, enabling it to synthesize semantically integrated patches that eliminate the memory leak.

Intertwined Logic and Memory. Figure 7(b) illustrates an example of intertwined logic and memory repair that presents significant challenges for traditional rule-based methods. Addressing this memory leak necessitates a three-step coordinated modification across disjoint code regions: (1) *introducing a temporary variable* (`str`) to track memory allocated by `strdup(tmpl)`, (2) *inserting deallocation instructions* (`free(str)`) before the function returns, and (3) *restructuring the control/data flow* by placing `free(str)` after the use of `filename` and storing the return value in an intermediate variable `result` rather than immediately returning it. This case represents a complex interdependence between program logic and memory management, where an incorrect modification to either aspect could result in functional errors.

TLR precisely repairs both the file-handling logic and the memory error. Neither SAVER nor ProveNFix can fix the original code, where allocation occurs in a nested call. To probe their limits, we refactored the statement at ℓ_{365}, `char *filename = basename(strdup(tmpl));`, into two statements: `char *str = strdup(tmpl);` and `char *filename = basename(str);`. Even with this simplification, both tools placed `free(str);` before `return format("%s%s", filename, extn);` at ℓ_{369}, introducing a use-after-free because `filename` aliases memory within `str`. In contrast, TLR defers deallocation until after the last use of alias-derived pointers, producing a semantically correct patch that removes the leak without changing behavior.

(a) Complex Data Structure

```
--- a/recutl.c
+++ b/recutl.c
   rec_db_t db;
   db = rec_db_new (); // alloc
@@ -331,7 +331,6 @@recutl_build_db (...)
   if (!recutl_parse_db_from_file (...))
   {
-      free (db);
-      db = NULL;
+      rec_db_destroy(db);
   }
```

(b) Intertwined Logic and Memory

```
--- a/main.c
+++ b/main.c
@@ -365,11 +365,13 @@replace_extn(...) {
-   char *filename = basename(strdup(tmpl));
+   char *str = strdup(tmpl);
+   char *filename = basename(str);
    char *dot = strrchr(filename, '.');
    if (dot)
        *dot = '\0';
-   return format("%s%s", filename, extn);
+   char *result = format("%s%s", filename, extn);
+   free(str);
+   return result;
```

(c) Subtle Pointer Aliasing

```
--- a/xhttp.c
+++ b/xhttp.c
@@ -835,8 +835,9 @@parse(..., req)
   req->headers=headers; // alias
   req->method =xh_string_new(...);
+  headers.list = NULL;
  ...

866  if (...){
867     free(headers.list); // free
868     return FAILURE(...);
```

(d) Cyclic Allocation

```
--- a/src/note.c
+++ b/src/note.c
    while (token) {
       case ...:
@@ -119,7 +119,10 @@scrotNoteNew(...)
      switch (type) {
       case 'f':
+        if (note)
+          free(note->font);
         // note->font alloc in parseText
         note->font = parseText(&token, end);
```

Fig. 7. TLR's patches for: (a) a memory leak in the `recutiles` [22] project; (b) a memory leak in the `chibicc` [56] project; (c) a double-free vulnerability (CVE-2023-38434 [54]) in the xHTTP [19] project; and (d) a memory leak error in the `scrot` [68] project.

Subtle Pointer Aliasing. Figure 7(c) exemplifies a subtle memory management issue where two pointers, `headers` and `req->headers` at ℓ_{835}, reference the same memory region, potentially resulting in double deallocation when memory is freed in both the `close_connection()` and `parse()` functions. The challenge here lies not in the syntactic complexity of the code, but in tracing pointer relationships across function boundaries and execution paths for identifying shared memory references across different control points and contexts.

SAVER reports "failed to convert labeling operators," indicating it cannot locate the proper deallocation site. ProveNFix inserts an early return before the first `free()` at ℓ_{867} instead of enforcing correct memory-lifecycle management (e.g., nullifying the shared pointer), masking the crash but causing premature termination with potential leaks and incomplete behavior. While effective at detecting error conditions, ProveNFix lacks the aliasing semantics needed to synthesize correct fixes and defaults to conservative avoidance. In contrast, TLR uses context-trace alias reasoning to place a nullification of `headers.list` immediately after the pointer copy, rather than after `free()`, thereby eliminating the double-free.

Cyclic Allocation. Figure 7(d) illustrates TLR's effectiveness in addressing cyclic allocation. The error manifests within a while loop where, upon consecutive iterations through the same conditional branch, the program repeatedly allocates memory for the `note->font` pointer without first deallocating previously allocated memory resources. This implementation deficiency creates *orphaned memory blocks*, as each subsequent allocation overwrites the reference to previously allocated memory without proper deallocation, thereby causing a persistent memory leak. Unlike conventional memory management errors where resources simply remain unreleased, this particular error requires understanding that deallocation must precede reallocation within a cyclic execution context. The non-local nature of the required fix—inserting logic before allocation rather than at the typical post-operation release points—exceeds the pattern-matching capabilities of SAVER and ProveNFix.

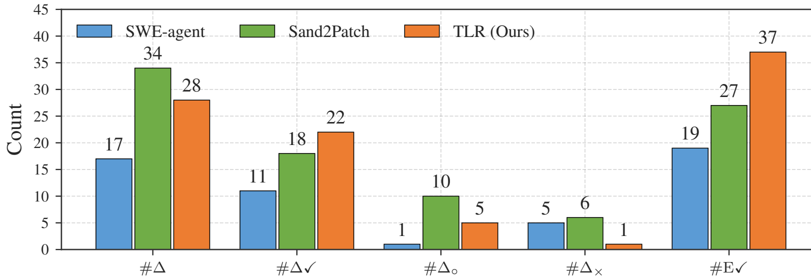

Fig. 8. Comparison of fixing effectiveness among TLR, SWE-agent [82], and Sand2Patch [34] on our dataset.

5.6 Comparison with LLM-based Approach (RQ2)

5.6.1 Results. As illustrated in Figure 8, TLR successfully repairs 37 memory errors, representing a 94.7% increase compared to the 19 errors addressed by SWE-agent. Furthermore, TLR substantially reduces the generation of harmful patches that introduce new errors, with only 1 instance compared to SWE-agent's 5. Figure 8 also contrasts TLR with Sand2Patch [34], a tree-of-thought LLM-agent repair tool driven by sanitizer crash logs. Because Sand2Patch reasons primarily over the crash stack, it repairs noticeably fewer errors than TLR and, like SWE-agent, tends to introduce more harmful patches on use-after-free cases whose root cause propagates beyond the immediate crash site. In contrast, TLR's typestate-guided context trace recovers the full error-propagation path, allowing it to synthesize correct fixes on precisely the cases where crash-log-only approaches fail. These results empirically demonstrate that augmenting LLMs with typestate-guided context retrieval yields significantly higher repair precision than relying solely on the LLM's intrinsic reasoning capabilities.

5.6.2 Analysis and Case Study. SWE-agent is unable to repair the memory leak in Figure 7(a) because it lacks awareness of the database object's typestate transitions context. SWE-agent cannot reliably determine whether `rec_db_destroy(db)` constitutes the appropriate replacement, as it must consider potential risks such as double-free vulnerabilities. Similarly, SWE-agent fails to comprehend the intricate pointer alias relationships in Figure 7(c) without the execution context that captures memory references and state transitions. Moreover, it demonstrates inadequate performance when addressing the cyclic allocation scenario in Figure 7(d), due to its inability to perform interprocedural memory lifecycle tracking and to identify the recurring allocation pattern that characterizes this particular vulnerability.

5.6.3 Token Usage. In TLR, each context trace records a memory error's start, occurrence, and end points together with its full propagation path. Among all subjects, `recutiles` exhibits the longest propagation path at 4,052 steps and consequently incurs the highest token consumption for TLR; as Figure 9 shows, it is likewise the most token-intensive subject for SWE-agent. Even so, TLR consumes 97% fewer tokens than SWE-agent across the dataset.

5.7 Ablation Analysis (RQ3)

5.7.1 Results. Figure 10 presents the ablation analysis result comparing TLR with three variants (see Section 5.2): TLR-M (TLR with the methods containing the error), TLR-F (TLR with the file containing the error) and TLR-NT (TLR without context trace). Overall, the complete system consistently outperforms all variants. (a) Correct patches: TLR-F and TLR-M produce 7 and 11 correct patches, respectively; TLR-NT increases this to 15. The complete TLR achieves 22 correct patches, a 47% improvement over TLR-NT and 214% over TLR-F. (b) New errors: TLR-F, TLR-M, and TLR-NT introduce 3, 6, and 4 new errors, whereas TLR introduces only 1—an 83% reduction.

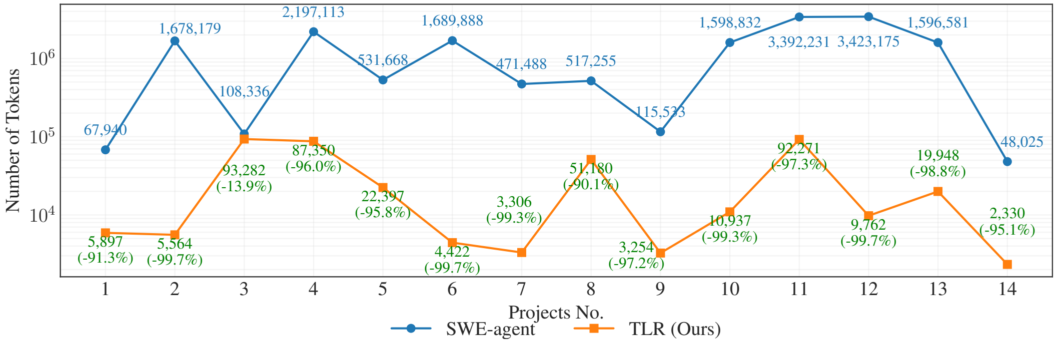

Fig. 9. Comparison of the token consumed among TLR, SWE-agent [82], and Sand2Patch [34] on our dataset.

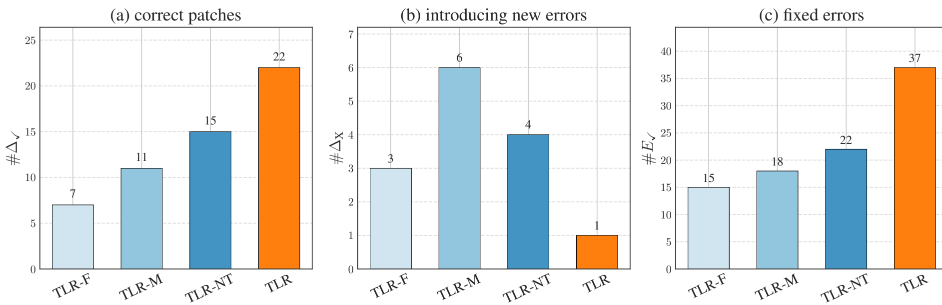

Fig. 10. Ablation analysis result.

(c) Total errors fixed: TLR-F and TLR-M fix 15 and 18 errors, and TLR-NT fixes 22, while TLR fixes 37, improving on TLR-NT by 68% and on TLR-F by 147%.

5.7.2 Analysis and Case Study. In Figure 7 (c) and (d), neither these variants can successfully repair these complex memory errors, as they lack the necessary execution context. Without typestate-guided context tracing, LLMs cannot identify critical semantic relationships—specifically, that pointer aliasing causes the double-free vulnerability in case (c), and that `note->font` undergoes multiple allocations without proper deallocation in case (d). These examples demonstrate the fundamental necessity of context tracing in memory error repair. In Figure 7 (c), context tracing enables the system to observe subtle changes in multiple pointers' states and track function call propagation across execution boundaries. Similarly, in Figure 7 (d), it facilitates the tracking of `note->font`'s allocation state across diverse execution paths.

6 Discussion

Repair Scope. Our approach does not attempt to detect or repair every memory error in a repository; instead, it targets errors that can be reproduced using a specific proof-of-concept (PoC). This focus limits generalizability to errors not exercised by the PoC. To ensure a fair and rigorous evaluation, we provide identical PoC inputs to all baselines. Importantly, the correctness of any generated patch is subsequently confirmed through a strict validation process, combining fuzzing tests with manual inspection. Under these controlled and meticulously verified conditions, our approach consistently outperforms them.

Dataset. To mitigate data contamination, our benchmark includes memory errors primarily drawn from bugs reported and patched after the Claude 3.5 Sonnet knowledge cutoff, making it unlikely that their code or fixes appear in training data. While some evaluated projects or patches may still be present, potentially introducing bias, this limitation applies equally to all LLM baselines. Despite

this, our typestate-guided context retrieval consistently achieves substantial gains, indicating that its effectiveness extends beyond any advantage from data exposure.

LLM Selection. Our evaluation primarily utilizes Claude 3.5 Sonnet, although empirical evidence suggests that comparable LLMs (e.g., GPT-4o) demonstrate similar performance on memory error APR tasks. While more sophisticated models might offer improvements, the focal point of our research is the enhancement of memory error APR through typestate-guided context retrieval, rather than a comparative assessment of performance across different LLM architectures.

Concurrency. TLR currently targets *single-threaded* memory errors, consistent with prior work [29, 63]. Errors from thread interleavings (e.g., concurrent use-after-free or double-free from data races) are out of scope, as our debugger-based tracing captures typestate transitions along a single execution. Extending to concurrent settings requires per-thread typestate tracking with shared ownership and synchronization reasoning, plus non-deterministic PoC reproduction due to scheduling; we leave this to future work.

7 Related Work

Automated Memory Error and Vulnerability Repair. Repairing memory errors is a complex task due to the non-local nature of memory management and its temporal properties. Various efforts have been proposed to address this issue [25, 29, 38, 43, 49, 51, 52, 63, 69]. SemFix [51] and Angelix [43] are general-purpose repair techniques that, while broadly applicable, demonstrate lower efficacy compared to specialized approaches tailored for specific error categories such as memory errors [29] and null dereferences [80]. AddressWatcher [49] can only fix memory leaks and cannot be applied to use-after-frees and double-frees. While MemFix [38] is effective for small-scale programs, it struggles to scale up for larger applications and fails to generate patches that include conditional deallocation for safety checks. FootPatch [69] requires templated annotations at the bug locations and can inadvertently introduce double-free errors when addressing memory leaks. SAVER [29] is more scalable for larger applications, but it does not take advantage of the intermediate bug information provided by Infer, which inhibits its effectiveness. ProveNFix [63] uses temporal property-based specifications, referred to as future conditions, to repair memory errors and other temporal bugs. However, a common limitation among these existing memory error APR tools is their dependence on manually crafted specifications. Beyond these tools, classical symbolic AVR systems for C, including ExtractFix [26], CrashRepair [61], and EffFix [89], rely on crash-constraint extraction, concolic execution, or incorrectness separation logic with expert-crafted strategies, and like SAVER and ProveNFix, struggle with the subtle cases in Section 5. LLM-agent-based systems SAN2PATCH [34] and PatchAgent [85] target codebase-level C vulnerabilities: SAN2PATCH struggles with UAFs whose root cause lies beyond the crash stack, whereas TLR's typestate-guided tracing recovers global error semantics and enables cross-file repair; PatchAgent's human-like interaction incurs substantially higher token usage, while TLR's agentless, analysis-guided retrieval achieves better effectiveness at lower cost. Learning-based methods VulRepair [24] and VulMaster [94] operate at function level and fall outside our codebase-level scope. Orthogonal to AVR, CPR [60] combats overfitting via concolic path exploration; integrating it with typestate-guided retrieval is a promising future direction. Overall, existing solutions often struggle with codebase-level multi-hunk errors and lack in-context repair beyond primitive APIs. To our knowledge, TLR is the first to integrate *memory-operation traces* to guide codebase-level memory error repair.

LLMs for Automated Program Repair. Recent advancements in Automated Program Repair (APR) [36] have witnessed the emergence of LLM-based techniques, which can be categorized

into two primary approaches: Open-Source-LLM-based [30, 31, 41, 74, 83, 84, 86, 88] and Closed-Source-LLM-based [32, 33, 64, 71, 73, 75, 76, 82, 87, 90]. Open-Source-LLM-based methods typically necessitate substantial additional data for model fine-tuning. For instance, Mashhadi et al. [41] curated over 600,000 samples to enhance Java single-statement bug repair capabilities. However, such extensive datasets are particularly challenging to assemble for memory error repairs due to their specialized nature. Our approach aligns more closely with Closed-Source-LLM-based methodologies, which leverage the inherent capabilities of pre-trained LLMs as their foundation, subsequently augmenting them with external contextual information [33, 75] or sophisticated decision-making chains [76, 90]. Recent research [7, 71, 82, 87] has further evolved this paradigm by employing agent-based frameworks that enhance the repair process through interactive engagement with isolated computational environments. While our approach shares the utilization of Closed-Source LLMs with existing methods, it fundamentally differs in both focus and implementation. Prior approaches predominantly target general bug fixing (mostly in Java and Python) but face limitations when addressing memory errors due to the extensive contextual information required. Our novel contribution lies in the development of typestate-guided context retrieval, which effectively compresses error-related contexts to lengths suitable for LLM processing while preserving critical semantic information necessary for accurate memory error repair.

8 Conclusion

The paper presents TLR, a novel approach to automated memory error repair in C programs using Large Language Models (LLMs). By leveraging LLMs' extensive knowledge of code and natural language, guided by a finite typestate automaton and structured prompting, TLR addresses the limitations of traditional automated memory error repair methods and previous deep learning approaches. Our tool demonstrates significant success in repairing real-world memory errors across large-scale open-source projects, outperforming existing state-of-the-art tools and even fixing three zero-day memory errors [91–93]. This approach shows promise in advancing the field of automated program repair, particularly for complex memory-related errors in C programming.

Data Availability

To facilitate and promote future research, we have made our implementation and all associated data publicly available at [3].

Acknowledgments

This work is supported by the Australian Research Council (DP250101396) and UTS IDP funding (PRO25-24165). We thank Wenkang Zhong (Nanjing University) and the anonymous reviewers for their valuable feedback. GPT-based tools (e.g., ChatGPT) were used for language polishing.

References

[1] Alexander Borisov. 1999. Fast C/C++ HTML 5 Parser. https://github.com/lexborisov/myhtml
[2] Alexey Frunze. 2021. Smaller C is a simple and small single-pass C compiler. https://github.com/alexfru/SmallerC
[3] Anonymous. 2025. *(Artifact) TLR: Codebase-Level C Memory Management Error Repair with Large Language Models.* https://github.com/EvilMemory/TLR-Memory-APR
[4] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet
[5] Eric Bodden. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE '12).* ACM.
[6] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. 2007. Denial of service or denial of security?. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07).* ACM.
[7] 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 Computer Society, Los Alamitos, CA, USA, 2188–2200. https://doi.org/10.1109/ICSE55347.2025.00157

[8] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*.

[9] Cesanta Software Limited. 2023. mJS: Restricted JavaScript engine. https://github.com/cesanta/mjs

[10] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM.

[11] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing. *Proc. ACM Softw. Eng.* FSE (2024).

[12] CISA. 2023. The Urgent Need for Memory Safety in Software Products. https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

[13] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation (PLDI '02)*. ACM.

[14] DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format Version 5. https://dwarfstd.org/doc/DWARF5.pdf

[15] Electrux. 2024. ls with coloring and icons. https://github.com/Electrux/ls_extended

[16] Fabrice Bellard. 2021. QuickJS Javascript Engine. https://github.com/bellard/quickjs

[17] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM.

[18] 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)*. USENIX Association.

[19] Francesco Cozzuto. 2022. A lightweight HTTP server as a library. https://github.com/cozis/xHTTP

[20] Free Software Foundation. 2011. Debugging with GDB. https://sourceware.org/gdb/current/onlinedocs/gdb.html

[21] Free Software Foundation, Inc. 1991. A handy little utility to display a tree view of directories. https://github.com/execjosh/tree

[22] Free Software Foundation, Inc. 2007. GNU Recutils. https://www.gnu.org/software/recutils/

[23] Free Software Foundation, Inc. 2007. Radare2: Libre Reversing Framework for Unix Geeks. https://github.com/radareorg/radare2

[24] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 935–947. https://doi.org/10.1145/3540250.3549098

[25] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*.

[26] 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 Transactions on Software Engineering and Methodology* 30, 2 (2021). https://doi.org/10.1145/3418461

[27] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2019).

[28] Yaru Hao, Yutao Sun, Li Dong, Zhixiong Han, Yuxian Gu, and Furu Wei. 2022. Structured Prompting: Scaling In-Context Learning to 1,000 Examples.

[29] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM.

[30] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE.

[31] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. 2025. Template-Guided Program Repair in the Era of Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1895–1907. https://doi.org/10.1109/ICSE55347.2025.00030

[32] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '23)*. ACM.

[33] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *Thirty-Seventh AAAI Conference on Artificial*

Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press.

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

[35] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*.

[36] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* (2019).

[37] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS*.

[38] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '18)*. ACM.

[39] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM.

[40] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* (2024).

[41] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR '21)*. IEEE.

[42] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*.

[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701.

[44] Meta. 2021. A static analyzer for Java, C, C++, and Objective-C. https://fbinfer.com/

[45] MITRE. 2024. CWE-401: Missing Release of Memory after Effective Lifetime. https://cwe.mitre.org/data/definitions/401.html

[46] MITRE. 2024. CWE-415: Double Free. https://cwe.mitre.org/data/definitions/415.html

[47] MITRE. 2024. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html

[48] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* (2018).

[49] A. Murali, M. Alfadel, M. Nagappan, M. Xu, and C. Sun. 2024. AddressWatcher: Sanitizer based Localization of Memory Leak Fixes. *IEEE Transactions on Software Engineering* (2024).

[50] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM.

[51] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[52] Thanh-Toan Nguyen, Quang-Trung Ta, Ilya Sergey, and Wei-Ngan Chin. 2021. Automated Repair of Heap-Manipulating Programs Using Deductive Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.).

[53] NIST. 2019. CVE-2019-6455. https://nvd.nist.gov/vuln/detail/CVE-2019-6455

[54] NIST. 2023. CVE-2023-38434. https://nvd.nist.gov/vuln/detail/CVE-2023-38434

[55] Peter Johnson and other Yasm developers. 2014. Yasm Assembler mainline development tree. https://github.com/yasm/yasm

[56] Rui Ueyama. 2019. chibicc: A Small C Compiler. https://github.com/rui314/chibicc.git

[57] Sean Barrett. 2017. stb single-file public domain libraries for C/C++. https://github.com/nothings/stb

[58] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association.

[59] Murray Shanahan, Kyle McDonell, and Laria Reynolds. 2023. Role play with large language models. *Nature* 623, 7987 (2023), 493–498.

[60] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. 390–405. https://doi.org/10.1145/3453483.3454051

[61] Ridwan Shariffdeen, Christopher S. Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2025. Vulnerability Repair via Concolic Execution and Code Mutations. *ACM Transactions on Software Engineering and Methodology* 34, 4, Article 105 (2025), 27 pages. https://doi.org/10.1145/3707454

[62] Congzheng Song and Ananth Raghunathan. 2020. Information leakage in embedding models. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 377–390.

[63] Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. *Proc. ACM Softw. Eng.* FSE (2024).

[64] Benjamin Steenhoek, Kalpathy Sivaraman, Renata Saldivar Gonzalez, Yevhen Mohylevskyy, Roshanak Zilouchian Moghaddam, and Wei Le. 2025. Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 01–13. https://doi.org/10.1109/ICSE55347.2025.00126

[65] Steven Massey, Volodymyr Shymanskyy. 2019. A fast WebAssembly interpreter and the most universal WASM runtime. https://github.com/wasm3/wasm3

[66] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* (1986).

[67] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*.

[68] Tom Gilbert. 2000. SCReenshOT - command line screen capture utility. https://github.com/resurrecting-open-source-projects/scrot

[69] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM.

[70] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*.

[71] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] https://arxiv.org/abs/2407.16741

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

[73] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery.

[74] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*.

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

[76] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Haotian Zhang, and Yuqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? *arXiv preprint arXiv:2404.12833* (2024).

[77] Yichen Xie and Alex Aiken. 2005. Context-and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 115–125.

[78] Qi Xin, Haojun Wu, Jinran Tang, Xinyu Liu, Steven P. Reiss, and Jifeng Xuan. 2024. Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs. *Proc. ACM Softw. Eng.* FSE (2024).

[79] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 414–425.

[80] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 512–523. https://doi.org/10.1109/ICSE.2019.00063

[81] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM.

[82] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual*

Conference on Neural Information Processing Systems. https://arxiv.org/abs/2405.15793
[83] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22).* ACM.
[84] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering (ICSE '22).* ACM.
[85] 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 Conference on Security Symposium.* USENIX Association, Article 226.
[86] Mingyue Yuan, Jieshan Chen, Zhenchang Xing, Aaron Quigley, Yuyu Luo, Tianqi Luo, Gelareh Mohammadi, Qinghua Lu, and Liming Zhu. 2025. DesignRepair: Dual-Stream Design Guideline-Aware Frontend Repair with Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE).* 2483–2494. https://doi.org/10.1109/ICSE55347.2025.00109
[87] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE] https://arxiv.org/abs/2401.07339
[88] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23).* IEEE.
[89] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2025. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Transactions on Software Engineering and Methodology* (2025). https://doi.org/10.1145/3705310
[90] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *CoRR* (2024).
[91] Zhihao Guo. 2023. CVE-2025-25739. https://github.com/lexborisov/myhtml/issues/194
[92] Zhihao Guo. 2024. Alloc Bugs. https://github.com/radareorg/radare2/issues/22767
[93] Zhihao Guo. 2024. memory leak in ls_extended. https://github.com/Chirag-Khandelwal/ls_extended/issues/45
[94] 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 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024).* https://doi.org/10.1145/3597503.3639222