



USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis

Youngjoon Kim and Sunguk Shin, *Korea University*; Hyounghick Kim,
Sungkyunkwan University; Jiwon Yoon, *Korea University*

<https://www.usenix.org/conference/usenixsecurity25/presentation/kim-youngjoon>

**This paper is included in the Proceedings of the
34th USENIX Security Symposium.**

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis

Youngjoon Kim
Korea University

Sunguk Shin
Korea University

Hyounghick Kim*
Sungkyunkwan University

Jiwon Yoon*
Korea University

Abstract

Research on automated vulnerability repair often requires extensive program analysis and expert input, making it challenging to deploy in practice. We propose SAN2PATCH, a system that generates patches using only sanitizer logs and source code, eliminating the need for costly program analysis or manual intervention. SAN2PATCH employs multi-stage reasoning with Large Language Models (LLMs) to decompose the patching process into four distinct tasks: vulnerability comprehension, fault localization, fix strategy formulation, and patch generation. Through tree-structured prompting and rigorous validation, SAN2PATCH can generate diverse, functionally-correct patches. Evaluations on the VulnLoc dataset show that SAN2PATCH successfully patches 79.5% of vulnerabilities, surpassing state-of-the-art tools like ExtractFix (43%) and VulnFix (51%) by significant margins. On our newly curated SAN2VULN dataset of 27 new vulnerabilities from various open-source projects, SAN2PATCH achieves a 63% success rate, demonstrating its effectiveness on modern security flaws. Notably, SAN2PATCH excels at patching complex memory-related vulnerabilities, successfully fixing 81.8% of buffer overflows while preserving program functionality. This high performance, combined with minimal deployment requirements and elimination of manual steps, makes SAN2PATCH a practical solution for real-world vulnerability remediation.

1 Introduction

Software projects contain various vulnerabilities that can lead to significant losses [4, 17]. Such damages have been increasing annually, with the total loss projected to reach \$10.5 trillion USD by 2025 [28]. Although automated tools such as fuzzing [24] have been developed to identify these issues, patching the discovered vulnerabilities still relies heavily on human experts. This reliance exists because, unlike vulnerability detection, patching requires a comprehensive understanding of the entire codebase, making automation difficult.

In addition, vulnerability patches must fix security flaws without changing the existing functionality of the project. Therefore, generating correct patches requires the sophisticated reasoning processes of domain experts, posing a significant challenge to automation.

Automated program repair (APR) has emerged as a promising solution for addressing software defects, with significant research advances in recent years [15, 46]. While early APR approaches focused on general bug fixing [19, 20, 29, 34], specialized techniques for vulnerability patching have demonstrated superior performance in addressing security flaws [13, 48]. The emergence of Large Language Models (LLMs) has further accelerated this field, with several studies exploring LLM-based approaches for vulnerability repair [12, 32, 42, 48, 51]. These studies leverage LLMs' advanced code understanding capabilities, showing promising results that suggest even greater potential as LLM technology continues to evolve [11, 43].

However, existing approaches face two critical limitations that hinder fully automated security patch generation. First, existing tools require substantial human intervention in the repair process. Many systems, such as VulnFix [48] and VulRepair [12], depend on manually provided information about patch locations or vulnerable code snippets. This dependency on human input creates a significant bottleneck in achieving complete automation. Additionally, most studies have focused solely on patch generation, neglecting the crucial step of fault localization [41], which is essential for a fully automated solution. Second, existing methods rely heavily on resource-intensive program analysis techniques. For instance, VulnLoc [38] and VulnFix [48] employ fuzzing-based methods for fault localization and patch generation, while ExtractFix [13] and CPR [37] utilize symbolic or concolic execution. These dynamic analysis approaches introduce significant challenges in practice. They necessitate complex environmental setups and instrumentation, which substantially increases the complexity of deployment and maintenance. Moreover, these methods consume considerable computational resources and processing time.

*Corresponding authors

To address these limitations, we propose SAN2PATCH, a novel approach to automated vulnerability repair that operates solely on sanitizer logs from vulnerability detection. By eliminating the need for complex program analyses or additional developer input, SAN2PATCH autonomously handles the entire repair pipeline—from fault localization to patch generation—addressing the key shortcomings of existing methods.

At the core of SAN2PATCH lies our utilization of LLMs for the efficient generation of patches. Specifically, we structure the process into four stages: *Vulnerability Comprehension*, *Where-to-Fix*, *How-to-Fix*, and *Candidate Generation*, and integrate Tree of Thought [45] prompting. We further devise an algorithm that extracts and provides code snippets to the LLM, balancing minimal snippet size with essential code context.

We evaluate SAN2PATCH on two comprehensive datasets to demonstrate its effectiveness in real-world vulnerability repair. First, on the VulnLoc dataset [38], SAN2PATCH achieves a 79% success rate (31 out of 39 vulnerabilities), significantly outperforming ExtractFix’s 43% success rate (9 out of 21) and VulnFix’s 51% success rate. Our detailed analysis shows that SAN2PATCH successfully patches 7 out of 9 vulnerabilities that ExtractFix could fix while also addressing 10 out of 12 vulnerabilities that ExtractFix failed to repair.

In addition, we conduct an assessment on our newly curated SAN2VULN dataset, comprising 27 new and unseen vulnerabilities from various open-source projects—including complex issues such as buffer overflows (e.g., CVE-2022-26981) and memory corruption bugs (e.g., CVE-2022-31783). SAN2PATCH successfully generates valid patches for 17 of these 27 vulnerabilities (63% success rate).

Particularly noteworthy is SAN2PATCH’s performance on memory-related vulnerabilities, achieving an 81.8% success rate on buffer overflow cases (18 out of 22). Furthermore, SAN2PATCH maintains program functionality in 81.8% of the patched buffer overflow cases, demonstrating its ability to generate precise fixes that preserve intended behavior.

The main contributions of this paper are as follows:

- We introduce SAN2PATCH, an LLM-based automated vulnerability repair system that relies only on sanitizer logs and source code, removing the need for additional developer input or sophisticated static analyses.
- We propose a four-step patch generation pipeline supplemented by Tree of Thoughts prompting, significantly enhancing the LLM’s reasoning capacity at each stage.
- We develop an optimized algorithm to extract and deliver concise yet context-rich code snippets and function context to the LLM, improving both performance and scalability.
- Through comprehensive evaluations across both the VulnLoc benchmark and our novel SAN2VULN dataset, we demonstrate SAN2PATCH’s superior performance

over existing solutions, with success rates of 79.5% and 63.0%, respectively, marking substantial improvements over state-of-the-art tools (ExtractFix: 43%, VulnFix: 51.3%).

We release SAN2PATCH as an open-source tool at <https://github.com/acorn421/san2patch>, and our benchmark, including SAN2VULN, is available at <https://github.com/acorn421/san2patch-benchmark>.

2 Motivating Example

In this section, we demonstrate the importance of SAN2PATCH by analyzing a real-world vulnerability, CVE-2016-1839, which was discovered in libxml2 [1]. This vulnerability is a heap-buffer overflow caused by an incorrect length parameter passed to `memcpy` at `dict.c:285`. Figure 1 shows the original code that contains the vulnerability.

```

283 found_pool:
284     ret = pool->free;
285     memcpy(pool->free, name, namelen); // bof
286     pool->free += namelen;
287     *(pool->free++) = 0;
288     pool->nbStrings++;
289     return(ret);

```

Figure 1: Vulnerable code in libxml2 (`dict.c:285`) where CVE-2016-1839 occurs.

When a buffer overflow vulnerability is triggered, the sanitizer generates a crash log. This log provides crucial information, including the call stack at the time of the crash and details about memory allocation and deallocation points. An example sanitizer log for CVE-2016-1839 is shown in Figure 2.

```

==2867725==ERROR: AddressSanitizer: heap-buffer-overflow
on address 0x** at pc 0x** bp 0x** sp 0x**
READ of size 73661 at 0x** thread T0
#0 0x** (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x79732)
#1 0x** in xmlDictAddString /src/dict.c:285
#2 0x** in xmlDictLookup /src/dict.c:926
#3 0x** in htmlParseNameComplex /src/HTMLparser.c:2517
... (Omitted)
#12 0x** in main /src/xmllint.c:3767
#13 0x** in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21c86)
#14 0x** in _start (/src/xmllint+0x774c9)
... (Omitted)
SUMMARY: AddressSanitizer: heap-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x79732)

```

Figure 2: Address sanitizer log of CVE-2016-1839.

The sanitizer log provides essential information for identifying the type and location of a vulnerability. This basic information serves as a foundation for patch generation. For example, if Figure 2 reveals that the vulnerability is a heap-buffer-overflow, developers can determine that one possible approach to fixing it is to add a boundary check at the point

```
Human: Based on the sanitizer output, determine which parts of the project code need to be fixed to patch the vulnerability.
```

```
<Sanitizer_Output>  
{sanitizer_output}  
</Sanitizer_Output>
```

```
LLM: To determine the appropriate fix locations for the heap-buffer-overflow vulnerability described in the AddressSanitizer output, we need to analyze the stack trace provided. ... Given this information, the most relevant fix location is in the 'dict.c' file at line 285, where the 'xmlDictAddString' function attempts to access the buffer. ...
```

(a) LLM's answer to identifying patch location.

```
diff --git a/dict.c b/dict.c  
--- a/dict.c  
+++ b/dict.c  
@@ -282,6 +282,9  
 }  
 found_pool:  
 ret = pool->free;  
+ if (pool->free + namelen >= pool->end) {  
+ return NULL; // Prevents buffer overflow  
+ }  
 memcpy(pool->free, name, namelen);  
 pool->free += namelen;  
 *(pool->free++) = 0;
```

(b) Final patch file generated by the LLM, based on the fix location in 3a.

Figure 3: LLM-generated patch without prompt engineering to fix CVE-2016-1839.

where the vulnerability is triggered. This solution is simple and direct, and ensures that out-of-bounds memory access is avoided by validating input variables. To this end, the first step is to identify the section of code that needs to be changed to fix the issue. If we input only the sanitizer output to an LLM without prompt engineering, and ask it to identify the patch location and generate a patch, the LLM will provide responses like Figure 3.

In Figure 3a, the LLM identifies `dict.c` at line 285 in the `xmlDictAddString` function as the fix location and generates a patch (Figure 3b) that adds a boundary check before the `memcpy` operation. The patch attempts to prevent buffer overflow by comparing `pool-> free + namelen` against `pool-> end`, returning `NULL` if the boundary is violated.

However, this patch fails to address the underlying vulnerability in CVE-2016-1839, which stems from a read exception at the source address of the `memcpy`, not a write exception at the target address. The LLM, although correctly identifying the issue as a buffer overflow, misdiagnosed the type of overflow and implemented checks on the destination buffer (`pool-> free`) instead of the source buffer (`name`). This illustrates a limitation in the LLM's understanding of memory safety semantics. To properly fix CVE-2016-1839, the solution requires input validation at a lower level in the stack trace, as demonstrated in the developer's patch shown in Figure 4.

```
diff --git a/HTMLparser.c b/HTMLparser.c  
--- a/HTMLparser.c  
+++ b/HTMLparser.c  
@@ -2473,2 +2473,4  
     in++;  
+     if (in == ctxt->input->end)  
+         return(NULL);  
     if ((*in > 0) && (*in < 0x80)) {  
@@ -2515,5 +2519,7  
         c = CUR_CHAR(1);  
     }  
+     if (ctxt->input->base > ctxt->input->cur - len)  
+         return(NULL);  
     return(xmlDictLookup(ctxt->dict, ctxt->input->cur  
         - len, len));  
 }
```

Figure 4: Developer's patch to address CVE-2016-1839.

```
Vulnerability Comprehension: "The vulnerability occurs due to improper handling of buffer boundaries. When a buffer overflow happens, it can lead to an out-of-bounds read, where data is read beyond the allocated memory. ..."
```

```
Where-To-Fix: "The function htmlParseNameComplex at line 2517 in HTMLparser.c is identified as the crash point in the stack trace, indicating it is directly responsible for the out-of-bounds read. ... This location is chosen because it is the earliest point in the stack trace where the vulnerability manifests, allowing us to address the root cause effectively."
```

```
How-To-Fix: "The fix involves adding boundary checks before performing read operations on buffers. ..."
```

```
Patch Generation: "... The fix involves checking that operations involving buffer pointers do not exceed the allocated bounds. The candidates ensure that before accessing the buffer with 'ctxt->input->cur - len', a check is performed to ensure the operation doesn't go out of the allocated memory. ..."
```

(a) The rationale for the answers of SAN2PATCH.

```
diff --git a/HTMLparser.c b/HTMLparser.c  
--- a/HTMLparser.c  
+++ b/HTMLparser.c  
@@ -2511,6 +2511,9  
     GROW;  
     }  
     len += 1;  
+     if ((ctxt->input->cur - len) < ctxt->input->base) {  
+         // Bounds check added  
+         return(NULL);  
+     }  
     NEXTL(1);  
     c = CUR_CHAR(1);  
 }
```

(b) Successful patch generated by SAN2PATCH.

Figure 5: Final patch to fix CVE-2016-1839.

From the developer's patch, input validation is performed at a lower level of the stack trace, focusing on the `ctx->input` variable at `HTMLparser.c`. However, as shown in Figure 3a, the LLM does not have sufficient reasoning power to provide an appropriate patch location. Therefore, in order to generate appropriate patches by considering only the sanitizer output using LLM, it is necessary to maximize the reasoning capabilities of the LLM. Among various methods, the most straightforward and representative approach is to improve the prompt structure through prompt engineering. Accordingly, we propose SAN2PATCH, which subdivides the patching process and uses various prompt structures, such as Chain-of-Thought [40], Self-Consistency [39], and Tree of

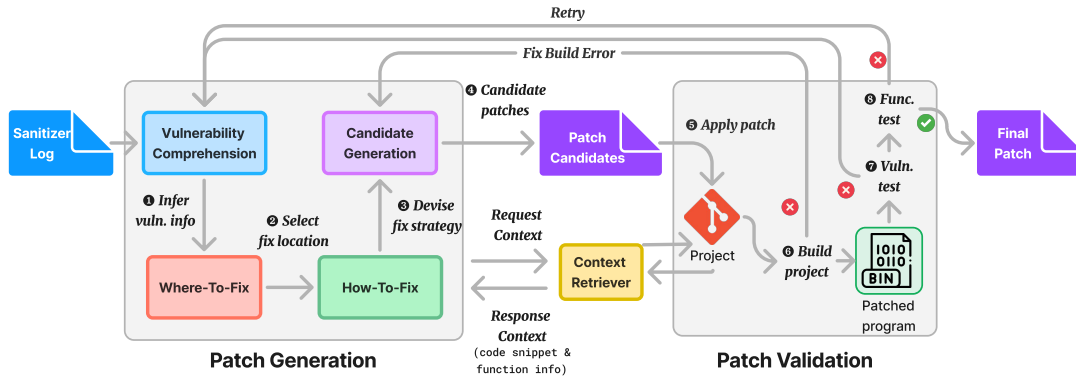


Figure 6: Overview of SAN2PATCH: Our system has two main phases — Patch Generation and Patch Validation. In Patch Generation, a sanitizer log undergoes four stages: ① *Vulnerability Comprehension* analyzes the vulnerability, ② *Where-to-Fix* identifies fix locations, ③ *How-to-Fix* formulates repair strategies, and ④ *Candidate Generation* creates candidate patches. The Context Retriever mediates between stages by providing relevant code snippets and function information. In Patch Validation, candidate patches are ⑤ first applied and then undergo rigorous testing, including ⑥ build verification, ⑦ vulnerability testing, and ⑧ functionality preservation checks, ultimately producing a validated patch. If validation fails, the system refines the patch until all criteria are met.

Thoughts [45], to perform patching. The results of running SAN2PATCH against CVE-2016-1839 are shown in Figure 5.

Figure 5a illustrates the rationale behind the LLM’s responses throughout the entire patching process implemented in SAN2PATCH. At each stage of generating a valid patch, the LLM can accurately understand the vulnerability (*Vulnerability Comprehension*), determine the location for the patch (*Where-To-Fix*) and fix strategy (*How-To-Fix*), and precisely derive the constraints required to fix the vulnerability (*Candidate Generation*). Also, the final patch generated by SAN2PATCH to fix CVE-2016-1839 is presented in Figure 5b. Although this patch slightly differs from the developer’s patch in terms of fix location, it successfully infers the necessary boundary conditions. After applying this patch and running the proof-of-concept (PoC), the vulnerability was no longer detected, and all of the project’s unit tests passed successfully.

3 SAN2PATCH Design

3.1 System Overview

SAN2PATCH is an end-to-end system for automated vulnerability repair that leverages LLMs to transform sanitizer logs into validated security patches. As illustrated in Figure 6, the system operates through two main phases: Patch Generation and Patch Validation, with several key components working in concert to enable reliable and efficient patch creation.

The Patch Generation phase implements a novel four-stage pipeline designed to mirror expert developers’ vulnerability repair workflow. In the *Vulnerability Comprehension* stage, LLMs analyze the sanitizer output to understand the root cause and characteristics of the vulnerability. This understanding informs the *Where-To-Fix* stage, where LLM reasoning is

employed to identify precise code locations requiring modification, similar to traditional fault localization approaches. The *How-To-Fix* stage then formulates repair strategies according to the type of vulnerability and fix location. Finally, the *Candidate Generation* stage combines these insights to create candidate patches that address the identified vulnerability.

To ensure robust patch generation, SAN2PATCH incorporates advanced techniques. The system employs Tree of Thoughts (ToT) [45] prompting, enabling the exploration of multiple reasoning paths simultaneously instead of a single line of thought. This is complemented by a self-evaluation [44] mechanism that evaluates and prunes reasoning paths, optimizing the exploration of solutions while maintaining patch quality. Additionally, a specialized Context Retriever module provides relevant code context to the LLM, balancing information completeness with token limitations.

The Patch Validation phase implements rigorous quality control through a series of automated checks. Each candidate patch undergoes sequential verification steps: build verification ensures syntactic correctness, vulnerability testing confirms the security fix, and functionality preservation checks validate that the patch maintains the program’s original behavior. This comprehensive validation ensures that only patches meeting all quality criteria are selected as final solutions.

These components create a robust framework that automatically generates and validates security patches, eliminating the need for manual intervention or advanced program analysis techniques. As shown in Figure 6, the Patch Generation phase uses only sanitizer logs and source code. The Patch Validation phase additionally requires a PoC input to trigger the vulnerability, along with the project’s build instructions, execution script, and functionality tests. These inputs are required to run SAN2PATCH end to end.

3.2 Multi-Stage Patch Generation

Generating vulnerability patches requires complex reasoning and domain expertise that can challenge even advanced LLMs. Drawing from prior research [49] on breaking down complex problems, SAN2PATCH implements a carefully structured four-stage pipeline that mirrors the systematic approach taken by expert developers. This decomposition aligns with established APR methodologies while incorporating critical aspects of real-world vulnerability remediation. Traditional APR primarily focuses on two phases: *Where-To-Fix* and *(Patch) Candidate Generation*. SAN2PATCH expands upon this foundation by adding two crucial stages that reflect actual developer workflows: *Vulnerability Comprehension* and *How-To-Fix*. These additional stages enable a more systematic and thorough approach to vulnerability repair.

Vulnerability Comprehension. In this initial stage, SAN2PATCH performs in-depth analysis of the sanitizer log to understand the vulnerability’s fundamental characteristics. Unlike typical scenarios where developers have access to detailed vulnerability reports, SAN2PATCH must infer critical information—including root cause, vulnerability type, and potential impact—solely from the sanitizer output. This understanding forms the foundation for subsequent stages, much like how developers begin their repair process by thoroughly analyzing the security flaw.

Where-To-Fix. With clear vulnerability information in place, SAN2PATCH identifies optimal code locations for implementing security fixes, corresponding to the fault localization phase in traditional APR methods [41]. The system primarily relies on crash stack traces from sanitizer logs, which contain crucial information about memory access patterns, heap allocations, and free operations. For example, heap-based buffer overflow traces typically include both memory access and heap allocation information, while UAF vulnerabilities present additional memory-free operation details. SAN2PATCH leverages an LLM-driven process to analyze these diverse stack trace patterns, supplemented by the context retrieval algorithm that provides the necessary code context for accurate analysis.

The system’s approach extends beyond simple stack trace analysis. Although stack traces often indicate the immediate crash location, LLM can identify alternative fix locations based on broader vulnerability context and code structure analysis. This flexibility is particularly important for complex vulnerabilities that require multi-hunk patches—modifications across multiple code locations. In such cases, SAN2PATCH can identify and manage multiple fix points simultaneously, ensuring comprehensive vulnerability remediation while maintaining code consistency.

How-To-Fix. Before generating patch candidates, SAN2PATCH formulates comprehensive repair strategies based on the vulnerability information gathered in the

previous stage. In this crucial step, the LLM generates specific fix guidelines by analyzing the vulnerability’s root cause, type, and characteristics identified during the *Vulnerability Comprehension* stage. This strategic planning stage is essential for avoiding one of the common pitfalls in automated repair: the tendency to generate patches that merely replicate patterns from training data.

The fix strategies developed in this stage serve as a foundation for generating diverse and effective patches. By establishing clear repair guidelines before attempting specific code modifications, SAN2PATCH can explore a broader range of potential solutions rather than defaulting to common fix patterns. This approach enables the system to generate more effective, targeted patches while ensuring they directly address the identified vulnerability characteristics. The resulting fix guidelines significantly influence subsequent stages, ultimately leading to more effective and varied patch candidates.

Candidate Generation. The final stage synthesizes insights from previous stages to produce concrete fixes. With the fix location already identified, SAN2PATCH provides the LLM with curated code context from the *Where-To-Fix* stage, along with vulnerability details and repair strategies from earlier stages. To ensure precise modifications, the system augments the code context with clear markers, including comments identifying the exact fix locations. Instead of using `diff` directly, which can complicate line numbering for LLMs, the system generates fully modified code files first. These files are then processed by dedicated tools to create accurate patch files. Each patch candidate undergoes thorough validation: build verification ensures syntactic correctness, PoC testing confirms the fix’s effectiveness, and functionality testing verifies that the original program behavior is preserved. By generating fully modified code instead of `diff` patches, SAN2PATCH avoids line-numbering issues common with LLMs. After modification, a patch generation tool creates the final accurate patch file.

Once the patch is generated, it is applied to the project, and the verification process begins. First, the modified project is built to check for any syntax or build errors. If any build errors occur, the error messages are fed back to the LLM, guiding it in generating corrected code. If the build errors cannot be resolved after several iterations, the patch is rejected. If the build succeeds, the patch is tested with the PoC input to ensure the vulnerability has been fixed. If the sanitizer still detects the crash, the patch is rejected. For patches that pass the security test, a final functionality test is conducted using the project’s unit tests to verify that the patch does not affect the original functionality. Only patches that successfully pass all tests are selected as the final patch.

3.3 LLM Reasoning Strategies

Each of the stages described earlier involves complex tasks that require advanced reasoning capabilities. Simple prompt-

ing is insufficient to achieve effective reasoning in such cases. Therefore, we employ various techniques to maximize the LLM’s reasoning performance.

Figure 7 illustrates the LLM prompting framework in SAN2PATCH. Each circle denotes a distinct prompt invocation. The *Vulnerability Comprehension* stage employs Chain-of-Thought with Self-Consistency (CoT-SC) to generate multiple reasoning paths and selects the most consistent answer. In contrast, the *Where-To-Fix*, *How-To-Fix*, and *Candidate Generation* stages apply ToT prompting [45] to explore diverse solutions. ToT generates multiple candidate paths at each step, assigns self-evaluated scores, and prunes low-scoring branches to focus on the most promising patch candidates.

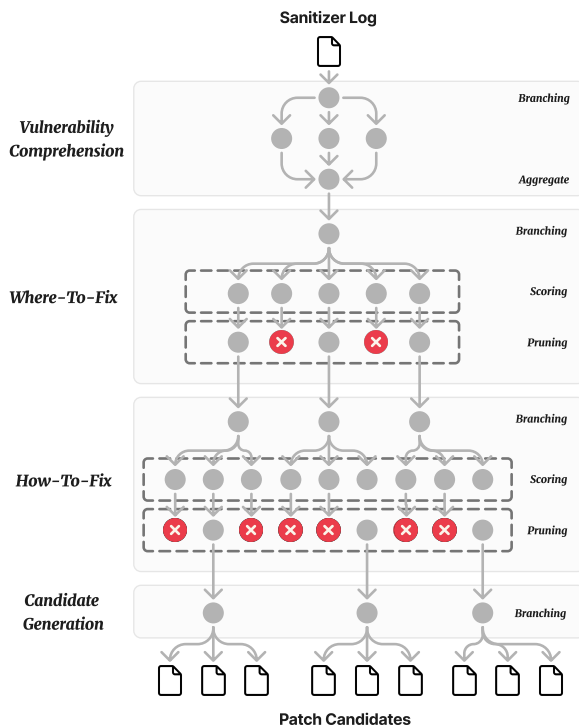


Figure 7: LLM Reasoning of SAN2PATCH: The *Vulnerability Comprehension* stage utilizes the CoT-SC to generate multiple answers and adopt the most frequent answer. *Where-to-Fix*, *How-to-Fix*, and *Candidate Generation* stages utilize the ToT to generate various locations and patches, and generate the final patch through various explorations. For the detailed prompt, see the supplementary material.¹

Rationale Engineering. First, we ensure that the LLM generates a corresponding rationale for each of its responses. This rationale provides an explanation for the answer, improving the LLM’s reasoning process. As demonstrated in previous research [21], generating a rationale along with the answer improves the reasoning performance of LLMs. Examples of the rationales generated at each stage can be found in Figure 5a.

¹<https://doi.org/10.5281/zenodo.15654492>

Furthermore, the generated rationale is essential for verifying whether the response is correct. This is particularly important in techniques such as Chain of Thought (CoT) prompting, where rationales guide the reasoning process. In CoT, the rationale for the previous step is utilized in the next step. Additionally, in SAN2PATCH, these rationales are passed to subsequent stages and serve as valuable extra information for further reasoning. For example, the vulnerability details inferred in the *Vulnerability Comprehension* stage are leveraged in later stages, where not only the results of the inference are passed but also the rationale for the inference. This helps to enhance the reasoning results of the LLM and increase the memory of previous steps.

CoT-SC. The results of each subtask in the patching process are used in subsequent stages. Therefore, if an incorrect result is generated at any stage, especially at an earlier stage, such as *Vulnerability Comprehension*, this may significantly reduce the overall quality of the patching process.

To address this issue, SAN2PATCH applies Chain of Thought (CoT) [40] and Self-Consistency [39] techniques to the first subproblem to improve the reliability of the LLM’s answers. CoT-SC involves prompting the LLM to generate multiple responses and then selecting the most consistent one. Initially, the LLM generates K different responses through the CoT process. These are then processed by an aggregation prompt, which evaluates each candidate and selects the most consistent one. This selected answer becomes the final output of the stage. This approach helps reduce the risk of the LLM providing probabilistically incorrect answers. The hyperparameter K , which specifies the number of answers generated simultaneously, is adjustable. In this work, we set K to 3.

ToT. Tree of Thoughts (ToT) prompts can provide better performance than simple Chain-of-Thought by generating diverse candidates of answers [45]. ToT reasoning works similarly to CoT by performing reasoning in sequential steps. However, unlike CoT, it explores multiple potential solutions at each step, allowing a diverse search for possible answers without aggregating them into a single answer before continuing to the next stage.

The ToT process consists of three main steps: *Branching*, *Scoring*, and *Pruning*. In the *Branching* step, the LLM generates K new candidate results based on the outcomes of the previous inference step. This process allows the model to explore diverse reasoning paths, ensuring a comprehensive search for potential solutions. However, as the inference progresses, the number of nodes can grow exponentially, making it crucial to retain only the most optimal nodes for further reasoning. To achieve this, in *Scoring* step, a self-evaluation [44] mechanism using the same LLM is employed to assess and assign scores to each reasoning node. Based on these scores, ToT retains the top N nodes through selection criteria. This *Pruning* step helps focus on the most promising reasoning paths while optimizing computational efficiency.

In SAN2PATCH, ToT prompting is applied across the *Where-To-Fix*, *How-To-Fix*, and *Patch Generation* stages. During the *Branching* step, the LLM generates K new reasoning nodes based on the results of the previous stage. For example, in the *Where-To-Fix* stage, the LLM selects five potential fix location candidates by analyzing the inferred root cause and type of vulnerability from the earlier stage. To maximize diversity in the generated responses, the LLM is prompted to produce all K candidates in a single query, ensuring a broader exploration of potential solutions.

In *Scoring* step, the generated nodes undergo self-evaluation using evaluation prompts. During this process, the evaluation LLM not only assigns a score to each node but also provides a confidence level for the given score, enhancing the reliability of the evaluation. The final score of a node is determined using the formula: $reliability_score = score \times (1 + 0.1 \times confidence)$. Both the score and confidence values are restricted to integers within the range of 1 to 10.

Finally, once the self-evaluation process determines each node’s score, only the optimal nodes are selected for further reasoning. Two approaches are used to select optimal nodes: a “greedy” method that retains the top N nodes with the highest scores and a “sampling” technique that performs weighted sampling based on the assigned scores.

In this study, the values of K and N were set to 5 and 3 for the *Where-To-Fix* stage, and 3 and 1 for the *How-To-Fix* stage. *Scoring* and *Pruning* steps were not applied in the *Patch Generation* stage. Instead, three patch candidates were generated per node and evaluated through validation. The selection of the optimal nodes was performed using a “sampling” technique.

3.4 Code Context Management

Managing code context is crucial for LLM-based vulnerability repair, as it directly impacts the quality of generated patches. SAN2PATCH implements two complementary approaches to provide comprehensive yet focused code context: AST-based code segment extraction and function-level context analysis.

Code Segment Extraction LLMs face inherent token limitations that make it impractical to process entire codebases. However, the balance of context is critical—too much code can dilute the LLM’s attention, while too little can lead to incorrect patches. For example, missing variable declarations can prevent proper boundary check implementation during buffer overflow repairs.

To address this challenge, we present an AST-based context retrieval algorithm described in Algorithm 1. Rather than extracting arbitrary line ranges, this algorithm identifies compound statements through AST parsing, focusing on syntactically meaningful units that include the target fix location. These blocks naturally capture essential programming constructs like conditional statements, loops, and function declarations. The algorithm recursively examines up to $M(= 5)$ com-

Algorithm 1 AST-based Code Context Retrieval Algorithm

```

1: Input: Source File ( $F$ ), Target Line ( $L$ ),  $MIN\_LINE$ ,  $MAX\_LINE$ ,  $M$ 
2: Output: Extracted Code Block ( $B_{code}$ )
3:  $S_{code} \leftarrow$  Read entire source code from  $F$ 
4:  $AST_{code} \leftarrow$  Parse  $S_{code}$  into an Abstract Syntax Tree (AST)
5:  $N_{root} \leftarrow$  Root node of  $AST_{code}$ 
6: function FINDTARGETNODE( $N, L$ )
7:   for  $child \in N.children$  do
8:      $result \leftarrow$  FINDTARGETNODE( $child, L$ )
9:     if  $result \neq \emptyset$  then
10:      return  $result$ 
11:   if  $L$  is within the range of lines for node  $N$  then
12:     return  $N$ 
13:   return  $\emptyset$ 
14: function FINDCOMPOUNDBLOCKS( $N, k$ )
15:    $blocks \leftarrow []$ 
16:   while  $N \neq \emptyset$  and  $k > 0$  do
17:     if  $N.type ==$  “compound_statement” then
18:        $blocks.append(N.parent)$ 
19:        $k \leftarrow k - 1$ 
20:      $N \leftarrow N.parent$ 
21:   return  $blocks$ 
22:  $N_{target} \leftarrow$  FINDTARGETNODE( $N_{root}, L$ )
23:  $blocks \leftarrow$  FINDCOMPOUNDBLOCKS( $N_{target}, M$ )
24: for  $block \in blocks$  do
25:    $B_{code} \leftarrow$  Extract lines from  $S_{code}$  for  $block$ 
26:   if  $MIN\_LINE \leq$  number of lines in  $B_{code} \leq MAX\_LINE$  then
27:     return  $B_{code}$ 
28: return  $\emptyset$  ▷ Return empty if no suitable block found

```

ound statements that encompass the target line, selecting appropriately sized blocks while preserving their parent contexts. Through extensive experimentation, we established optimal parameters with $MIN_LINE = 20$ and $MAX_LINE = 100$. However, these values are adjusted based on specific stage requirements (e.g., smaller contexts for *Where-To-Fix* stages versus *Candidate Generation*). The rationale for choosing AST-based code context extraction is discussed in detail in Section 5.3. Examples of the code context provided at each stage are explained with illustrations in the supplementary material.

Function Context Extraction To support semantically accurate patch generation, SAN2PATCH augments code blocks with additional function-level context. This includes the function signature, which provides the name, return type, and argument list, as well as all return statements within the function. These elements help the LLM infer the intended behavior by revealing how outputs are computed and which variables are emphasized.

While code segments provide the immediate structure for edits, function-level context offers a broader view of intent and usage. Together, they guide the model toward generating patches that preserve not only syntactic correctness but also the expected program semantics.

4 Evaluation

In this section, we evaluate the performance of the proposed SAN2PATCH framework through various experiments. To guide the evaluation, we set five key research questions:

- RQ1: How does SAN2PATCH’s performance compare to other vulnerability repair tools?
- RQ2: Can SAN2PATCH practically patch recent or new vulnerabilities?
- RQ3: How does patching performance vary across different LLM models?
- RQ4: How effective is SAN2PATCH in generating functionally correct patches?
- RQ5: Can SAN2PATCH successfully patch a variety of vulnerability types?

VulnLoc Benchmark. We evaluated SAN2PATCH using the VulnLoc dataset [38], which contains 43 real-world vulnerabilities from nine major open-source projects, including binutils, ffmpeg, libxml2, and libtiff. The dataset encompasses five vulnerability types, ranging from buffer overflows and integer overflows to more complex use-after-free vulnerabilities.

For our evaluation, we compiled all projects with appropriate sanitizer instrumentation and executed them with PoC inputs to generate crash logs. Four vulnerabilities (bugchrome-1404, CVE-2017-9992, CVE-2016-3186, CVE-2016-5314) were excluded as they could not be reproduced with sanitizer instrumentation, resulting in a final evaluation set of 39 vulnerabilities.

SAN2VULN Benchmark. To evaluate SAN2PATCH’s ability to handle novel vulnerabilities, we created SAN2VULN, a benchmark focusing on recent and previously unseen security flaws. This addresses a key limitation of the VulnLoc dataset, which contains vulnerabilities only from 2018 and earlier that could exist in LLM training data.

We constructed SAN2VULN by systematically collecting and validating recent vulnerability reports from public GitHub issues, identified through sanitizer log keywords. Each vulnerability was reproduced in a controlled environment matching the VulnLoc setup. The final dataset contains 27 vulnerabilities from four open-source projects. Of these, 11 vulnerabilities were published after August 2024, ensuring they fall outside the training period of the LLM used in our experiments. Furthermore, 12 vulnerabilities in the dataset currently lack official fixes, providing an authentic test of SAN2PATCH’s ability to generate novel patches. For consistency in evaluation, we standardized the dataset format to align with VulnLoc. The Appendix provides detailed information about SAN2VULN.

Implementation We implemented SAN2PATCH in Python 3.10, leveraging several key frameworks. The LangChain package [7] enabled seamless integration with LLM APIs,

including GPT, Claude, and Gemini. For structured prompt engineering, we used LangGraph [2] to organize our four-stage patch generation pipeline as interconnected subgraphs, forming a unified processing workflow. LangSmith [3] provided comprehensive logging and debugging capabilities, allowing us to optimize prompts and handle edge cases effectively.

Validation We validated each generated patch through a three-stage process. First, we confirmed that the patched project compiled successfully. Second, we reran the original proof-of-concept input to ensure that the vulnerability no longer triggered a crash. Third, we conducted functionality testing using project-specific unit tests where available. Functional tests are detailed in Appendix B.

After conducting automated tests, we manually validated every patch that passed the automated tests. Two security experts, each with over eight years of experience in vulnerability detection and patch validation, independently reviewed each patch that passed automated testing. Each patch was categorized into one of four labels: *Correct (developer-matched)*, when the patch was semantically identical to the developer’s fix; *Correct (alternative)*, when the patch differed from the developer’s fix but resolved the general cause of the vulnerability without changing the program’s functionality; *Incorrect (failed some test cases)*, when the patch did not address the underlying cause and only prevented specific PoCs; and *Incorrect (failed all test cases)*, when the patch was entirely incorrect, such as simply removing vulnerable code. Disagreements were resolved through structured discussion. Details and illustrative examples of manual validation are included in the supplementary material.

Evaluation Environment. All experiments were conducted in a 64-bit Ubuntu 22.04 environment. The hardware used for the experiments comprised an Intel Xeon Gold 5218 CPU with 32 cores (2.30 GHz) and 64 GB of RAM. Throughout the experiments, only OpenAI, Anthropic and Google LLM models were utilized, and no GPU resources from the server were employed. Unless otherwise stated, the default LLM model used was OpenAI’s “gpt-4o-2024-08-06” model.

4.1 RQ1. Superiority of SAN2PATCH

To evaluate the performance of SAN2PATCH, we compared it with existing vulnerability repair tools for C language programs. SAN2PATCH was tested against VulnFix [48] and ExtractFix [13], both focused on patching vulnerabilities in C programs. VulnFix performs only patch generation, providing the patch location without fault localization, whereas ExtractFix handles both fault localization and patch generation. The results for VulnFix and ExtractFix were taken directly from their respective papers. For ExtractFix, only a subset of the Vulnloc benchmark was tested in the original study, so we included results for those specific datasets. Neither study conducted functionality tests; patches were considered successful

if they matched or were semantically equivalent to the developer's patch. We applied the same criteria in our comparison and considered VulnFix successful if either of its two backends generated a valid patch.

We also conducted a comparative analysis with recent LLM-based patch generation studies, using VulRepair [12] and VulMaster [51] as baselines. Unlike SAN2PATCH, these approaches do not perform fault localization but instead generate patches based on the vulnerable code snippet and its CWE-formatted vulnerability type. The comparison focused solely on patch generation performance. Developer patches from our dataset were used to extract relevant code snippets for testing, with the ground truth CWE ID provided as input. Special tokens were inserted around the fix location in the code, and the generated patches were manually evaluated against the developer patches as ground truth.

To assess the impact of our contributions, we conducted experiments under different settings by adjusting prompting techniques and context retriever configurations. In the *Zero-shot* configuration, the process involved two steps similar to [32]: selecting a single fix location based on the sanitizer output and extracting the corresponding code segment for the LLM. In the *CoT* configuration, our problem decomposition approach was applied without branching, with prompts structured in a sequential CoT format.

In the *No Context* setting, SAN2PATCH's ToT reasoning was applied while disabling the context retriever. Without the context retriever, the LLM received limited function-related information, as the code snippet was extracted by including $(MAX_LINE + MIN_LINE)/2$ lines both before and after the target line, ensuring a consistent number of surrounding lines instead of relying on AST-based parsing.

To ensure fairness, the context retriever was also disabled in the *Zero-shot* and *CoT* configurations. The *No Context* and original SAN2PATCH settings were tested up to five times to evaluate their patch generation success. Since SAN2PATCH generates up to nine patch candidates in a single execution via ToT prompting, the *Zero-shot* and *CoT* experiments were conducted nine times more than the SAN2PATCH experiments. The results of all experiments for RQ1 are summarized in Table 1.

First, the performance comparison between VulnFix and ExtractFix is as follows. As shown in Table 1, SAN2PATCH outperformed the other tools. Compared to VulnFix, SAN2PATCH successfully patched eleven more vulnerabilities in total. Considering that VulnFix operates with the patch location already provided, this demonstrates a notable difference in performance. In particular, SAN2PATCH performed better in addressing buffer overflow vulnerabilities, successfully patching 18 out of 22 cases, while VulnFix patched only 10. This is because the LLM can accurately infer the buffer size and generate appropriate constraints.

SAN2PATCH achieved a higher success rate than ExtractFix, although the datasets are not identical. SAN2PATCH

patched 31 out of 39 vulnerabilities for a success rate of 79.5%. Considering only the 21 vulnerabilities in ExtractFix's dataset, SAN2PATCH successfully patched 17 out of 21 vulnerabilities for a success rate of 81%, compared to ExtractFix's 42.9% (9 out of 21). A detailed comparison revealed that SAN2PATCH successfully patched 7 out of the 9 vulnerabilities that ExtractFix patched while also patching 10 of the 12 vulnerabilities that ExtractFix failed to patch. This demonstrates SAN2PATCH's ability to address vulnerabilities that traditional approaches cannot.

Compared to VulRepair and VulMaster, which are LLM-based patch generation tools, SAN2PATCH demonstrated significantly better performance. Both VulMaster and VulRepair require the fix locations for code patches. Three items from the VulnLoc benchmark were excluded due to the difficulty of extracting fix locations from the corresponding developer patches. As a result, the evaluation was conducted on the remaining 36 items. VulMaster and VulRepair successfully patched only 2 and 1 vulnerabilities, respectively. However, VulMaster successfully patched CVE-2016-8691, which SAN2PATCH failed to address. Further investigation revealed that VulMaster's training data contained closely related solutions, suggesting that its success may have resulted from data leakage rather than generalization ability.

The performance of VulRepair and VulMaster has declined in terms of exact match accuracy compared to the results reported in their paper. This decline is attributed to the issue of label leakage. Although the original VulMaster removed identical samples between the training and test datasets, we found that they did not completely eliminate similar data samples. In the VulMaster test set, 7.8% of the answers were identical to those in the training set. Additionally, when comparing code samples based on Jaccard Similarity of words, 15.8% of the samples exceeded a similarity score of 0.7.

In the performance comparison across various configurations, SAN2PATCH achieved the best results. The *Zero-shot*, *CoT*, and *No Context* configurations patched 21, 22, and 24 vulnerabilities, while SAN2PATCH successfully patched 31. Notably, SAN2PATCH outperformed the *Zero-shot* approach, which relies solely on the LLM's inherent reasoning, by patching 10 more vulnerabilities. These results underscore the significant impact of the proposed ToT prompting and Context Retriever on improving LLM-based vulnerability patching.

SAN2PATCH achieved a 79.5% patch success rate (31/39 vulnerabilities), significantly outperforming ExtractFix (43%) and VulnFix (51%). Our approach with ToT prompting and context retrieval showed particular strength in handling buffer overflows (18/22 cases), demonstrating superior performance over both traditional APR tools and recent LLM-based approaches.

Table 1: Comparative Performance of SAN2PATCH and Existing Patch Generation Tools. Note that VulnFix and ExtractFix did not perform functionality tests. VulRepair and VulMaster were provided with the vulnerable code and CWE information.

Package	Bug ID	Bug Type*	Zero-shot		CoT		No Context		SAN2PATCH		VulnFix [48]	ExtractFix [13]	VulRepair [12]	VulMaster [51]
			A†	M‡	A	M	A	M	A	M				
binutils	CVE-2017-14745	IO	✗	✗	✗	✗	✗	✗	✓	✓	✓	–	✗	✗
	CVE-2017-15020	BO	✓	▲	✓	▲	✓	✓(A)	✓	✓(A)	✓	–	✗	✗
	CVE-2017-15025	DZ	✓	✓	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓	✗	✗
	CVE-2017-6965	BO	✗	✗	✗	✗	✗	✗	✓	▲	✓	–	✗	✗
coreutils	gnubug-19784	BO	✗	✗	✓	▲	✓	✓(A)	✓	✓	✓	✗	✗	✗
	gnubug-25003	IO	✓	✓(A)	✓	✓(A)	✓	✓	✓	✓	✓	✗	✗	✗
	gnubug-25023	BO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✗	✗	✗
	gnubug-26545	IO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✗	✗	✗	✗
jasper	CVE-2016-8691	DZ	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
	CVE-2016-9557	IO	✗	✗	✓	✓(A)	✓	▲	✓	✓(A)	✗	–	✗	✗
libarchive	CVE-2016-5844	IO	✓	✓	✓	✓(A)	✓	✓	✓	✓(A)	✓	–	✗	✗
libjpeg	CVE-2012-2806	BO	✓	✓	✓	✓(A)	✓	✓	✓	✓(A)	✓	✓	✗	✗
	CVE-2017-15232	NPD	✓	✓(A)	✓	✓(A)	✗	✗	✗	✗	✓	✓	–	–
	CVE-2018-14498	BO	✗	✗	✗	✗	✗	✗	✗	✓(A)	✗	✗	✗	✗
	CVE-2018-19664	BO	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
libming	CVE-2016-9264	BO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	–	✗	✗
	CVE-2018-8806	UAF	✗	✗	✗	✗	✗	✗	✗	✗	✗	–	✗	✗
	CVE-2018-8964	UAF	✗	✗	✗	✗	✗	✗	✗	✗	✗	–	✗	✗
libtiff	bugzilla-2611	DZ	✓	✓(A)	✓	✓(A)	✗	✗	✓	✓(A)	✗	✓	✗	✗
	bugzilla-2633	BO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✗	✗	✗	✗
	CVE-2016-10092	BO	✗	✗	✗	✗	✓	✓(A)	✓	✓(A)	✗	–	✗	✗
	CVE-2016-10094	BO	✓	✓(A)	✓	▲	✓	▲	✓	✓(A)	✗	✗	✗	✓
	CVE-2016-10272	BO	✗	✗	✗	✗	✓	✓(A)	✓	✓(A)	✗	–	✗	✗
	CVE-2016-5321	BO	✓	✓	✓	✓(A)	✓	✓	✓	✓	✓	✓	✗	✗
	CVE-2016-9273	BO	✗	✗	✗	✗	✗	✗	✓	✓(A)	✗	✗	✗	✗
	CVE-2016-9532	BO	✗	✗	✗	✗	✓	✓(A)	✓	✓(A)	✗	–	✗	✗
	CVE-2017-5225	BO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✗	–	✗	✗
	CVE-2017-7595	DZ	✓	✓(A)	✓	✓	✓	✓(A)	✓	✓(A)	✓	✓	✗	✗
	CVE-2017-7599	DT	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✗	–	–	–
	CVE-2017-7600	DT	✓	✓	✓	✓	✓	✓(A)	✓	✓(A)	✗	–	–	–
	CVE-2017-7601	IO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✗	✗	✗
	libxml2	CVE-2012-5134	BO	✓	✓	✓	✓	✗	✗	✓	✓(A)	✓	✓	✗
CVE-2016-1838		BO	✓	✓(A)	✓	✓(A)	✓	✓(A)	✓	✓	✗	✗	✗	✗
CVE-2016-1839		BO	✗	✗	✓	✓(A)	✓	✓(A)	✓	▲	✓	✗	✗	✗
CVE-2017-5969		NPD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
potrace	CVE-2013-7437	BO	✗	✗	✗	✗	✓	▲	✓	▲	✓	–	✗	✗
zziplib	CVE-2017-5974	BO	✗	✗	✗	✗	✓	✓(A)	✓	✓(A)	✓	–	✗	✗
	CVE-2017-5975	NPD	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✗	✗
	CVE-2017-5976	BO	✗	✗	✓	▲	✓	▲	✓	✓(A)	✗	–	✗	✗
Total			22/39	21/39	26/39	22/39	28/39	24/39	34/39	31/39	20/39	9/21	1/36	2/36

* IO: Integer Overflow, DT: Data Type Overflow, DZ: Divide by Zero, NPD: Null Pointer Dereference, BO: Buffer Overflow, UAF: Use-After-Free.
† Automated validation. ✓: Both vulnerability and functionality tests passed; ✗: Vulnerability test or functionality test failed.
‡ Manual validation. ✓: Correct (developer-matched); ✓(A): Correct (alternative); ▲: Incorrect (failed some test cases); ✗: Incorrect (failed all test cases). For detailed examples of each determination, see the supplementary material.

4.2 RQ2: Effectiveness on New Vulnerabilities

In this research question, we evaluate whether SAN2PATCH can effectively patch recent and previously unseen vulnerabilities. Specifically, we focus on its performance for vulnerabilities not included in the LLM’s training data. To test this, we used our newly constructed SAN2VULN benchmark, which primarily consists of recent vulnerabilities, in contrast to the outdated VulnLoc benchmark. Additionally, we measured the time and cost required for patch generation to assess the practical feasibility of SAN2PATCH in real-world scenarios.

As shown in Table 2, SAN2PATCH successfully patched 17 out of 27 vulnerabilities in the SAN2VULN benchmark. Notably, it patched 6 out of 10 vulnerabilities that do not yet have an official fix (prefixed with NOFIX), demonstrating its ability to handle entirely new vulnerabilities. Furthermore, among the 11 vulnerabilities disclosed after September 2024, which were not included in the LLM’s training data,

SAN2PATCH successfully patched 7. These results confirm that SAN2PATCH can effectively patch vulnerabilities the LLM has not been trained on.

The evaluation also shows that SAN2PATCH is practical in both time and cost. The average time required per vulnerability was 8 minutes and 55 seconds, with a patch generation cost of \$0.48 per attempt, regardless of success or failure. These results demonstrate that SAN2PATCH can be reasonably applied in real-world environments.

SAN2PATCH effectively patches recent and unseen vulnerabilities, including those without official fixes, at a low cost (\$0.48 per attempt) and within practical timeframes (under 9 minutes per patch).

Table 2: Evaluation of SAN2PATCH on SAN2VULN.

Package	Bug ID	SAN2PATCH		Pass@	Total Cost(\$)	Total Time (min:sec)
		A	M			
libming	CVE-2024-24148	✓	✓(A)	1	0.26	08:55
	CVE-2024-24146	✓	✓(A)	1	0.24	08:47
Bento4	NOFIX-2024-002	✓	▲	1	0.31	03:27
	NOFIX-2024-003	✓	✓(A)	1	0.25	03:19
	NOFIX-2024-004	✓	✓(A)	1	0.26	05:36
	NOFIX-2024-005	✓	▲	1	0.4	05:00
	NOFIX-2024-006	✓	▲	1	0.3	03:08
	NOFIX-2024-007	✓	✓(A)	1	0.27	04:02
	NOFIX-2024-008	✓	▲	1	0.26	06:16
	NOFIX-2024-009	✓	✓(A)	1	0.21	03:09
	NOFIX-2024-010	✓	✓(A)	1	0.29	04:06
	NOFIX-2024-001	✓	✓(A)	1	0.29	11:39
Liblouis	CVE-2022-26981	✓	✓(A)	1	0.27	08:01
	CVE-2022-31783	✗	✗	5	1.26	47:31
	GIT-2024-1530	✗	✗	5	1.05	46:34
	GIT-2024-1531	✗	✗	5	1.13	53:15
	GIT-2024-1532	✓	✓(A)	1	0.39	09:53
	GIT-2024-1533	✓	✓(A)	2	0.53	18:29
	GIT-2024-1534	✓	✓(A)	1	0.27	09:03
	GIT-2024-1535	✓	✓(A)	2	0.45	14:49
	GIT-2024-1536	✗	✗	5	1.15	71:34
	GIT-2024-1537	✓	✓(A)	1	0.27	09:11
Kamilio	OSV-2024-1206	✓	✓(A)	1	0.33	09:53
	OSV-2024-1210	✓	✓(A)	1	0.28	06:15
	OSV-2024-1230	✓	✓(A)	5	1.16	42:04
	OSV-2024-1244	✓	▲	1	0.37	09:48
		23 / 27	17 / 27		0.48 (Avg.)	08:55 (Avg.)

4.3 RQ3. Impact of LLM Models

To evaluate how the performance of the LLM affects SAN2PATCH, we compared its performance using different LLM models. LLM performance can be measured with various benchmarks, depending on the domain and purpose. Among these, we identified MMLU and HumanEval as key benchmarks likely to impact SAN2PATCH’s performance. MMLU (Massive Multitask Language Understanding) [8] measures code generation capabilities through a set of programming tasks. These benchmarks are widely used and considered essential because SAN2PATCH’s reasoning process requires both general knowledge and language understanding as well as code generation skills. To analyze how LLM performance affects SAN2PATCH, we selected models with varying MMLU and HumanEval scores for our experiments. The models tested included GPT-4o, GPT-4o-mini, GPT-3.5-turbo, Claude 3.5 Sonnet, Gemini 1.5 Pro, and Gemini 1.5 Flash.

Table 3 presents the patching results for each model. The evaluation criteria were consistent with those used in RQ1 and RQ2, assessing whether vulnerabilities were fixed and functionality remained intact. To enhance reliability, patches that passed automated validation were manually inspected.

As shown in Table 3, GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro delivered sequentially higher performance, while smaller models performed less effectively. GPT-4o, the best-performing model, successfully patched 31 out of 39 vulnerabilities. Claude 3.5 Sonnet and Gemini 1.5 Pro patched 28 and 18 vulnerabilities, respectively. Notably, Gemini 1.5 Pro fixed 2 of the 8 vulnerabilities that GPT-4o failed to patch.

Table 3: Patching results of SAN2PATCH using various LLMs.

Bug ID	GPT-4o	GPT-4o-mini	GPT-3.5	Claude 3.5 Sonnet	Gemini 1.5 Pro	Gemini 1.5 Flash
CVE-2017-14745	✓	✗	✓	✗	✗	✓
CVE-2017-15020	✓	✗	✗	✓	✓	✓
CVE-2017-15025	✓	✗	✗	✓	✗	✓
CVE-2017-6965	✗	✗	✗	✗	✗	✗
gndbug-19784	✓	✗	✗	✓	✓	✓
gndbug-25003	✓	✓	✓	✓	✓	✓
gndbug-25023	✓	✓	✗	✓	✓	✓
gndbug-26545	✓	✗	✗	✓	✓	✗
CVE-2016-8691	✗	✗	✗	✗	✗	✗
CVE-2016-9557	✓	✗	✗	✓	✗	✗
CVE-2016-5844	✓	✗	✗	✓	✗	✗
CVE-2012-2806	✓	✗	✗	✓	✓	✗
CVE-2016-95232	✗	✓	✗	✓	✓	✗
CVE-2018-14498	✓	✗	✗	✗	✗	✗
CVE-2018-19664	✗	✗	✗	✗	✗	✗
CVE-2016-9264	✓	✗	✓	✓	✓	✗
CVE-2018-8806	✗	✗	✗	✗	✗	✗
CVE-2018-8964	✗	✗	✗	✗	✗	✗
bugzilla-2611	✓	✓	✗	✓	✓	✗
bugzilla-2633	✓	✓	✗	✓	✓	✗
CVE-2016-10092	✓	✗	✗	✓	✗	✗
CVE-2016-10094	✓	✗	✗	✓	✗	✗
CVE-2016-10272	✓	✗	✗	✓	✗	✓
CVE-2016-5321	✓	✓	✗	✓	✓	✓
CVE-2016-9273	✓	✗	✗	✓	✗	✗
CVE-2016-9532	✓	✓	✗	✗	✓	✗
CVE-2017-5225	✓	✓	✗	✓	✗	✗
CVE-2017-7595	✓	✓	✓	✓	✓	✓
CVE-2017-7599	✓	✓	✓	✓	✗	✗
CVE-2017-7600	✓	✓	✗	✓	✗	✓
CVE-2017-7601	✓	✓	✗	✓	✗	✗
CVE-2012-5134	✓	✗	✗	✗	✓	✗
CVE-2016-1838	✓	✗	✗	✓	✓	✗
CVE-2016-1839	✗	✗	✗	✗	✓	✓
CVE-2017-5969	✓	✓	✓	✓	✗	✓
CVE-2013-7437	✗	✗	✗	✗	✗	✗
CVE-2017-5974	✓	✓	✗	✓	✓	✗
CVE-2017-5975	✓	✗	✗	✓	✗	✗
CVE-2017-5976	✓	✗	✗	✓	✓	✓
Total	31 / 39	17 / 39	8 / 39	28 / 39	18 / 39	14 / 39

GPT-4o-mini achieved a similar performance to Gemini 1.5 Pro, successfully patching 17 vulnerabilities.

In contrast, GPT-3.5-turbo and Gemini 1.5 Flash exhibited significantly lower performance. They patched only 8 and 14 vulnerabilities, respectively, which was below the success rate of the larger models. This reduced performance can be attributed to the smaller size of these models, which limits their ability to understand and process code effectively. Smaller LLMs, such as GPT-3.5-turbo and Gemini 1.5 Flash, often failed to follow the required output format, leading to patching failures.

SAN2PATCH’s performance improves with more advanced LLMs, as its effectiveness relies heavily on the LLM’s capabilities. Among the tested models, GPT-4o achieved the highest performance, demonstrating the potential of advanced LLMs in vulnerability patching.

4.4 RQ4. Functional Correctness

In this research question, we investigate whether the patches generated by SAN2PATCH preserve the original functionality of the program. In vulnerability patching, it is critical not only to fix the vulnerability but also to ensure that the program’s intended behavior remains unchanged. Unlike vulnerability tests that primarily detect crashes, functionality tests require skilled developers to manually create test cases, making them difficult to implement. Consequently, many projects lack comprehensive functionality tests, highlighting the importance of generating patches that do not alter existing behavior. To address this issue, we conducted experiments to evaluate whether SAN2PATCH can produce functionally correct patches. We compared the validation results of SAN2PATCH’s patch candidates with those from alternative experimental settings, using the same prompting and context retriever configurations as in RQ1, including zero-shot, CoT, and No Context. The results are summarized in Table 4.

Table 4: Comparison of functionality and patching success by prompt strategies.

	Zero-shot	CoT	No Context	SAN2PATCH
# Vuln. test passed*	77	93	50	103
# Func. test passed	22	26	28	34
# Patch success†	21	22	24	31
success / vuln. test passed	27%	24%	48%	30%

* Number of intermediate patch candidates that removed the vulnerability; only those passing all functional tests are selected as final patches for each prompt strategy.

† Number of patch candidates that successfully passed manual validation.

As shown in Table 4, experiments using ToT demonstrated a higher likelihood of producing functionally correct patches that resolved vulnerabilities. In SAN2PATCH, 30% of the patches that fixed vulnerabilities were verified to maintain the original program functionality. Similarly, in the No Context setting, which also uses ToT, 48% of the vulnerability-fixing patches preserved functionality. In contrast, simpler reasoning techniques like zero-shot and CoT showed lower success rates, with only 27% and 24% of the patches, respectively, maintaining the original program behavior. These results highlight that the ToT-based patch generation approach in SAN2PATCH is more effective at producing functionally correct patches.

The ToT-based patch generation methods, No Context and SAN2PATCH, produce a higher proportion of successful patches — achieving success rates of 48% and 30% respectively — outperforming zero-shot (27%) and CoT (24%).

4.5 RQ5: Patching Various Types

We evaluate whether SAN2PATCH can effectively patch various types of vulnerabilities. Using the GPT-4o model, the

patching performance of SAN2PATCH is categorized based on different vulnerability types. The results are summarized in Table 5.

Table 5: Patching success rates and vulnerability handling by SAN2PATCH for different vulnerability types.

Vulnerability Type	SAN2PATCH			
	Vuln. Passed	Success Rate	Patch Success	Success Rate
IO	6/6	100%	6/6	100%
BO	22/22	100%	18/22	82%
DZ	3/4	75%	3/4	75%
NPD	2/3	67%	2/3	67%
UAF	0/2	0%	0/2	0%
DT	2/2	100%	2/2	100%

As shown in Table 5, SAN2PATCH successfully patched a wide range of vulnerabilities. It was particularly effective in handling data type overflow and integer overflow vulnerabilities, achieving a 100% success rate for both types. Other arithmetic errors, such as ‘divide by zero,’ showed a success rate of 75%. For null pointer dereference vulnerabilities, SAN2PATCH successfully patched two out of three instances. For memory-related vulnerabilities, SAN2PATCH achieved an 81.8% success rate in addressing buffer overflow vulnerabilities. This high success rate can be attributed to the simplicity of patches, such as adding boundary checks near the crash stack trace, which effectively resolved the vulnerabilities. However, in four cases, the patches altered the original program’s functionality, causing them to fail functionality tests. These failures likely occurred because the patched function was widely used in other parts of the program. To pass functionality tests in such cases, more precise patches are required, involving an analysis of how the patched function is utilized in other parts of the code.

SAN2PATCH effectively patches various vulnerabilities, achieving 82% success for buffer overflow, and 100% for data type overflow, and integer overflow, while addressing divide by zero (75%) and null pointer dereference vulnerabilities.

5 Discussion

5.1 Limitation of SAN2PATCH

SAN2PATCH is limited to generating patches for vulnerabilities that are detected by a sanitizer. Vulnerabilities that cannot be detected by any sanitizer (e.g., the use of weak cryptographic algorithms [26] or authentication bypasses [25]) are fundamentally unpatchable. Moreover, in this work, we focus on AddressSanitizer [36] and UndefinedBehaviorSanitizer [23]. Therefore, vulnerabilities detectable only by other sanitizers, such as race conditions [27], which can be identified by ThreadSanitizer [22], are outside the scope of our

evaluation and require further study.

SAN2PATCH's patch validation process also has some limitations. First, in the vulnerability test, only a single PoC is used to test each vulnerability, which means that new vulnerabilities introduced by a patch may go undetected. Although fuzzing could help mitigate this issue, achieving sufficient coverage and reliable results requires running fuzzers for an extended period, which is often not practical. Additionally, in the functionality test, we rely on the project's existing unit tests. If the unit test coverage is incomplete, the accuracy of the functionality test may be significantly reduced. To address these limitations, we performed manual validation in this work; however, a more accurate and efficient functional testing process is still needed.

5.2 Analysis of Failure Cases

We carefully analyzed the cases where patching failed. As a result, we identified the following reasons for failure. Examples of failed patches can be found in the supplementary material.

Inaccurate Vulnerability Classification In several cases, we observed that SAN2PATCH incorrectly identified the vulnerability type during the first stage, *Vulnerability Comprehension*. Most sanitizer logs provided sufficient descriptions of the vulnerabilities, enabling the LLM to accurately determine the types of vulnerabilities. However, a few logs reported bug types that differed from the actual root cause, causing the LLM to misunderstand the vulnerability. Since this incorrect information propagates through subsequent stages, it leads to incorrect patch generation, ultimately preventing successful patches.

Challenges in Fault Localization SAN2PATCH's fault localization relies on stack traces. However, some vulnerabilities exhibit distinct characteristics, where the root cause lies far from the stack trace. Therefore, in such cases, reasoning in the *Where-To-Fix* stage often failed. For example, in CVE-2018-19664, which all models failed to patch, the vulnerability is triggered by a function pointer call. Developers fixed this by modifying the function pointer's assignment, which is far from the bug point. As a result, SAN2PATCH was unable to generate a successful patch. Similarly, for CVE-2016-8691, another vulnerability that the model failed to patch, simply fixing the location where the divide-by-zero occurs is insufficient; addressing the underlying root cause is necessary to prevent further divide-by-zero vulnerabilities. Similarly, for two UAF vulnerabilities — CVE-2018-8806 and CVE-2018-8964 — where SAN2PATCH also failed to generate patches, it is necessary to address their underlying root causes rather than simply patching the crash location indicated by the stack traces.

Limitations of Single-Location Fixes In SAN2PATCH's *Where-To-Fix* stage, multiple fix locations can be selected at once, but in most cases, the LLM tended to choose only a

single vulnerable code region. Unfortunately, when dealing with more intricate vulnerabilities, modifying only a single location is not sufficient.

Incorrect Code Modifications Even when the LLM fully understood the vulnerability and correctly identified the fix location, it sometimes failed during *Candidate Generation*. Most of these failures involved generating completely invalid code, such as introducing incorrect constraints. In other cases, the generated code was insufficient, addressing only specific scenarios. For example, in CVE-2017-6965, the sanitizer log showed a buffer overflow on a buffer of size 157 bytes. The LLM used this detail to add a check for buffers smaller than 157 bytes, but the actual buffer size is determined at runtime. Similarly, for CVE-2013-7437, the LLM added an arbitrary large upper bound, resulting in a patch that only addressed the specific PoC. On the other hand, some patches introduced too many boundary checks, making the code overly restrictive and causing functional test failures.

Syntactic and Structural Errors in Patch Code In some cases, the LLM generated patches with basic syntax errors, most often mismatched curly braces, especially in lower-performing models. Post-processing resolved some errors, but incomplete code blocks and unexpected formatting still caused build failures. The LLM sometimes returned only partial edits despite instructions to generate complete code, or defined new functions within provided blocks, which often led to failures in vulnerability or functional tests.

Superficial Fixes That Bypass the Real Issue Another reason for patch failures was that the LLM sometimes attempted to bypass vulnerabilities through superficial fixes. For example, it occasionally removed entire functions or important logic, thereby avoiding the vulnerability trigger without addressing the root cause.

5.3 Code Context Trade-off

In SAN2PATCH, we primarily provide code surrounding the stack trace entries identified in the sanitizer log, parsed using an AST-based approach. This decision was based on experimental results indicating that supplying excessive context can actually hinder the LLM's ability to accurately select the correct fix location. When we provided program slicing or inter-procedural results to the LLM, it frequently chose irrelevant locations during fault localization, leading to lower-quality patch candidates. By restricting the input to code near the stack trace — where the root cause is most likely to reside — SAN2PATCH was able to improve overall patch success rates. Our design aligns with studies showing that relevant, focused information is more useful for patch generation [31]. Furthermore, research highlights the importance of local context for effective reasoning [33], indicating that targeted code context outperforms excessive code context.

Nonetheless, there are cases where the root cause does

not appear in the stack trace. For example, in CVE-2016-8691 and CVE-2018-19664, the functions modified by the developer's patch are not included in the stack trace. Focusing on code regions close to the stack trace generally improves efficiency and relevance, but may fail to capture vulnerabilities needing more thorough root cause analysis and expanded context. This underscores the balance between targeted focus and broader coverage.

5.4 Factors Influencing Performance

The effectiveness of SAN2PATCH is shaped by several key factors. One primary consideration is the potential influence of training data, as LLM-based approaches may perform better on vulnerabilities similar to those in their training sets. Previous research, such as VulMaster, demonstrated this effect, showing higher accuracy for known vulnerability patterns. We addressed this concern by carefully excluding vulnerability identifiers and temporal information from our experiments, forcing SAN2PATCH to rely on fundamental reasoning rather than pattern matching. The quality of crash analysis from sanitizer tools emerged as another crucial factor. Our use of GCC AddressSanitizer logs provided detailed crash information that enhanced SAN2PATCH's patch generation capabilities. This observation suggests that advances in crash analysis tools could further improve automated patch generation, mirroring how human analysts benefit from detailed crash information.

5.5 Vulnerability-Specific Performance

SAN2PATCH showed varying effectiveness across vulnerability types, excelling in buffer overflows and integer-related issues but struggling with UAF vulnerabilities. This highlights the need for more advanced methods to address complex memory-related issues, particularly those requiring a deeper understanding of memory lifecycle patterns. These results suggest promising directions for future research, including specialized strategies for challenging vulnerabilities.

6 Related Work

6.1 Automated Program Repair (APR)

Several studies in APR have specifically focused on patching security vulnerabilities [15]. One such approach is SenX [16], which generates patches for vulnerabilities by leveraging human-specified, vulnerability-specific safety properties. Another notable method is CPR [37], which co-explores both patch and input spaces through concolic path exploration, effectively reducing overfitting patches and improving repair correctness. ExtraFix [13], on the other hand, fixes vulnerabilities by extracting crash constraints, using these as proof obligations for patch synthesis without relying on extensive

test suites [13]. Finally, VulnFix [48] employs an inductive inference approach, constructing patch invariants from state mutations to generate patches, starting with a single failing input.

6.2 LLM-based Vulnerability Repair

Research has explored machine learning models for vulnerability repair [9, 10, 12], with LLMs emerging as a promising approach [50]. Pearce et al. [32] examines zero-shot vulnerability repair using LLMs, highlighting challenges in prompt design for generating secure code fixes. While LLMs perform well in synthetic scenarios, real-world applications reveal difficulties in creating functionally correct patches. Wu et al. [42] compares LLMs with deep learning techniques for Java vulnerability repair, emphasizing the need for larger datasets and better tuning methods. Similarly, Zhang et al. [47] demonstrates the superiority of pre-trained models over traditional methods, proposing enhancements through transfer learning and practical strategies. Zhou et al. [51] trains a CodeT5-based model using AST and CWE knowledge, while recent work [18, 30] explore CoT prompting for patch generation.

However, most LLM-based studies rely on explicitly provided modification locations and, in some cases, precise CWE information [51]. In contrast, SAN2PATCH requires only a sanitizer log, allowing the LLM to autonomously analyze the crash log, locate the issue, and generate patches without human intervention. Furthermore, while previous research primarily employs basic prompt engineering techniques like CoT, more advanced methods such as ToT, which explore diverse reasoning paths to enhance patch diversity, have remained unexplored. Our research demonstrates that ToT-based prompting not only improves patch success rates but also ensures vulnerability fixes while maintaining functional correctness.

6.3 Prompt Engineering

Efforts to maximize the reasoning capabilities of LLMs have been ongoing [14]. Among these approaches, prompt engineering [35] is a technique aimed at enhancing LLM performance without requiring fine-tuning. Unlike methods that alter the model itself, which demand substantial computational resources, prompt engineering optimizes input prompts to elicit better responses. Few-shot prompting [6] involves providing relevant QA pairs within the prompt to guide the model. CoT [40], a widely adopted method, improves LLM performance by breaking down problems into sub-steps and enabling sequential reasoning. Building on CoT, Self-Consistency [39] enhances reliability by generating multiple solutions and selecting the most consistent one. Advanced methods such as ToT [45] and Graph of Thoughts (GoT) [5] extend CoT by using tree or graph structures to explore diverse reasoning paths and identify the best solution. Additionally, Least-to-Most Prompting [49] addresses CoT's limitations

by decomposing complex tasks into simpler subproblems, improving generalization to harder challenges.

7 Conclusion

We presented SAN2PATCH, a system based on LLMs that generates security patches directly from sanitizer logs using ToT prompting. Our evaluation demonstrated that SAN2PATCH successfully patches 79% of vulnerabilities in the VulnLoc benchmark, significantly outperforming existing tools such as ExtractFix (43%) and VulnFix (51%). The system also proved effective on our newly curated SAN2VULN dataset, which consists of previously unseen vulnerabilities, achieving a success rate of 63% and showcasing its ability to generalize beyond the training data. Most notably, SAN2PATCH excelled at addressing complex memory-related vulnerabilities, successfully fixing 81.8% of buffer overflows while preserving program functionality.

Acknowledgments

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00460321 and No.RS-2024-00439819).

Ethics considerations

Our research adheres to strict ethical guidelines, ensuring responsible practices throughout. SAN2PATCH is designed to operate within the boundaries of responsible AI usage, aiming to automate vulnerability patching while minimizing risks such as unintended modifications or overfitting in repairs. The datasets utilized in this research—SAN2VULN—were collected with a focus on reproducibility and transparency, avoiding exposure to sensitive or proprietary code. SAN2VULN includes publicly available vulnerabilities and logs from open-source projects, as well as near zero-day vulnerabilities discovered during the dataset curation process. These vulnerabilities were responsibly disclosed to the developers, along with proposed patching strategies. To further enhance ethical adherence, we employed rigorous validation to confirm that the generated patches preserve functionality while addressing security flaws.

Open Science

Upon acceptance of this work, all research artifacts, including the SAN2PATCH framework, its underlying implementation, and relevant evaluation datasets, will be made publicly available. This aligns with the principles of open science, promoting transparency and reproducibility in automated vulnerability repair research. Our contributions will be shared via

an artifact evaluation process, ensuring they are accessible to the broader research community. The provided resources will include detailed documentation to support independent validation and further development. Access will be restricted to legitimate research purposes to prevent misuse of the system for malicious activities.

References

- [1] libxml2, 2018. URL <https://gitlab.gnome.org/GNOME/libxml2>.
- [2] LangGraph, 2024. URL <https://github.com/langchain-ai/langgraph>.
- [3] LangSmith, 2024. URL <https://www.langchain.com/langsmith>.
- [4] A. Anwar, A. Khormali, J. Choi, H. Alasmarty, S. Choi, S. Salem, D. Nyang, and D. Mohaisen. Measuring the cost of software vulnerabilities. *EAI Endorsed Transactions on Security and Safety*, 7(23), 2020.
- [5] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.
- [6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- [7] H. Chase. LangChain, Oct. 2022. URL <https://github.com/langchain-ai/langchain>.
- [8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [9] Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.
- [10] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering*, 49(2):564–585, 2022.
- [11] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large lan-

- guage models. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 1469–1481. IEEE, 2023.
- [12] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung. Vulrepair: a t5-based automated software vulnerability repair. In Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, pages 935–947, 2022.
- [13] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. ACM Transactions on Software Engineering and Methodology (TOSEM), 30(2):1–27, 2021.
- [14] J. Huang and K. C.-C. Chang. Towards reasoning in large language models: A survey. In 61st Annual Meeting of the Association for Computational Linguistics, ACL 2023, pages 1049–1065. Association for Computational Linguistics (ACL), 2023.
- [15] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang. Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities. ACM Computing Surveys, 2024.
- [16] Z. Huang, D. Lie, G. Tan, and T. Jaeger. Using safety properties to generate vulnerability patches. In 2019 IEEE symposium on security and privacy (SP), pages 539–554. IEEE, 2019.
- [17] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba. The secret life of software vulnerabilities: A large-scale empirical study. IEEE Transactions on Software Engineering, 49(1):44–63, 2022.
- [18] U. Kulsum, H. Zhu, B. Xu, and M. d’Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In Proceedings of the 1st ACM International Conference on AI-Powered Software, pages 103–111, 2024.
- [19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 38(1):54–72, 2011.
- [20] J. Lee, S. Hong, and H. Oh. 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, pages 95–106, 2018.
- [21] W. Ling, D. Yogatama, C. Dyer, and P. Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 158–167, 2017.
- [22] LLVM Project. ThreadSanitizer. <https://clang.llvm.org/docs/ThreadSanitizer.html>, . LLVM Compiler Infrastructure Project.
- [23] LLVM Project. UndefinedBehaviorSanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>, . LLVM Compiler Infrastructure Project.
- [24] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering, 47(11):2312–2331, 2019.
- [25] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>, . Common Weakness Enumeration, MITRE.
- [26] MITRE. CWE-327: Use of a Broken or Risky Cryptographic Algorithm. <https://cwe.mitre.org/data/definitions/327.html>, . Common Weakness Enumeration, MITRE.
- [27] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ("Race Condition") (Version 4.17). <https://cwe.mitre.org/data/definitions/362.html>, . Common Weakness Enumeration, MITRE.
- [28] S. Morgan. Cybercrime to cost the world \$10.5 trillion annually by 2025. Cybercrime Magazine, Nov 2020. URL <https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/>.
- [29] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In 2013 35th International Conference on Software Engineering (ICSE), pages 772–781. IEEE, 2013.
- [30] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. arXiv preprint arXiv:2402.17230, 2024.
- [31] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. Barr, and S. Mechtaev. The fact selection problem in llm-based program repair. arXiv preprint arXiv:2404.05520, 2024.
- [32] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In 2023 IEEE Symposium on Security and Privacy (SP), pages 2339–2356. IEEE, 2023.
- [33] J. A. Prenner and R. Robbes. Out of context: How important is local context in neural program repair? In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13, 2024.
- [34] S. Saha et al. Harnessing evolution for multi-hunk program repair. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 13–24. IEEE, 2019.

- [35] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. arXiv preprint arXiv:2402.07927, 2024.
- [36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. {AddressSanitizer}: A fast address sanity checker. In 2012 USENIX annual technical conference (USENIX ATC 12), pages 309–318, 2012.
- [37] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury. Concolic program repair. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 390–405, 2021.
- [38] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury. Localizing vulnerabilities statistically from one exploit. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pages 537–549, 2021.
- [39] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. In The Eleventh International Conference on Learning Representations.
- [40] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35: 24824–24837, 2022.
- [41] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. IEEE Transactions on Software Engineering, 42 (8):707–740, 2016.
- [42] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah. How effective are neural networks for fixing security vulnerabilities. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 1282–1294, 2023.
- [43] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 1482–1494. IEEE, 2023.
- [44] Y. Xie, K. Kawaguchi, Y. Zhao, J. X. Zhao, M.-Y. Kan, J. He, and M. Xie. Self-evaluation guided beam search for reasoning. Advances in Neural Information Processing Systems, 36, 2024.
- [45] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. Advances in Neural Information Processing Systems, 36, 2024.
- [46] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen. A survey of learning-based automated program repair. ACM Transactions on Software Engineering and Methodology, 33(2):1–69, 2023.
- [47] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen. Pre-trained model-based automated software vulnerability repair: How far are we? IEEE Transactions on Dependable and Secure Computing, 2023.
- [48] Y. Zhang, X. Gao, G. J. Duck, and A. Roychoudhury. Program vulnerability repair via inductive inference. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 691–702, 2022.
- [49] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. V. Le, et al. Least-to-most prompting enables complex reasoning in large language models. In The Eleventh International Conference on Learning Representations.
- [50] X. Zhou, S. Cao, X. Sun, and D. Lo. Large language model for vulnerability detection and repair: Literature review and roadmap. arXiv preprint arXiv:2404.02525, 2024.
- [51] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13, 2024.

A Ablation Study

We conducted an ablation study to verify whether the four stages of SAN2PATCH (*Vulnerability Comprehension*, *Where-To-Fix*, *How-To-Fix*, and *Candidate Generation*) effectively contribute to its overall performance. Since the *Where-To-Fix* and *Candidate Generation* stages are essential for patch generation, we focused on assessing the impact of removing the *Vulnerability Comprehension* and *How-To-Fix* stages. When these stages were omitted, the information typically produced by them (e.g., vulnerability type, vulnerability root cause, and fix strategy) was provided as empty values to subsequent stages. The detailed results of this experiment are presented in Table 6.

Table 6: SAN2PATCH’s performance with specific stage removed.

Package	Bug ID	No Comprehend		No howtofix	
		A	M	A	M
binutils	CVE-2017-14745	✓	✓(A)	✓	✓(A)
	CVE-2017-15020	✓	✓(A)	✓	✓(A)
	CVE-2017-15025	✓	✓	✓	✓(A)
	CVE-2017-6965	✓	▲	✓	▲
coreutils	gndbug-19784	✓	✗	✓	✓
	gndbug-25003	✓	▲	✓	✓(A)
	gndbug-25023	✓	✓(A)	✓	✓(A)
	gndbug-26545	✓	✓(A)	✓	✓(A)
jasper	CVE-2016-8691	✗	✗	✗	✗
	CVE-2016-9557	✗	✗	✓	✓(A)
libarchive	CVE-2016-5844	✓	✓	✓	✓
libjpeg	CVE-2012-2806	✓	✓	✗	✗
	CVE-2017-15232	✓	✓(A)	✗	✗
	CVE-2018-14498	✓	✓(A)	✗	✗
	CVE-2018-19664	✗	✗	✗	✗
libming	CVE-2016-9264	✓	✓	✓	✓(A)
	CVE-2018-8806	✗	✗	✗	✗
	CVE-2018-8964	✗	✗	✗	✗
libtiff	bugzilla-2611	✓	✓(A)	✓	✓(A)
	bugzilla-2633	✓	✗	✓	✗
	CVE-2016-10092	✗	✗	✓	✓(A)
	CVE-2016-10094	✓	▲	✓	✓(A)
	CVE-2016-10272	✗	✗	✓	✓(A)
	CVE-2016-5321	✓	✓	✓	✓
	CVE-2016-9273	✗	✗	✓	✓(A)
	CVE-2016-9532	✗	✗	✓	▲
	CVE-2017-5225	✓	✓(A)	✓	✓(A)
	CVE-2017-7595	✓	✓(A)	✓	✓
	CVE-2017-7599	✓	✓(A)	✓	✓(A)
	CVE-2017-7600	✓	✓(A)	✓	▲
	CVE-2017-7601	✓	✓(A)	✗	✗
libxml2	CVE-2012-5134	✓	✓	✓	✓
	CVE-2016-1838	✓	✓	✓	✓(A)
	CVE-2016-1839	✓	✓(A)	✓	✓(A)
	CVE-2017-5969	✓	▲	✓	✓
potrace	CVE-2013-7437	✓	▲	✗	✗
zziplib	CVE-2017-5974	✓	✓(A)	✗	✗
	CVE-2017-5975	✓	✓	✓	✓
	CVE-2017-5976	✓	▲	✗	✗
Total		30 / 39	22 / 39	28 / 39	24 / 39

Comparing the original experimental results shown in Ta-

ble 1 with the ablation results in Table 6, we clearly observe improved performance when all components are included. Specifically, omitting the *Vulnerability Comprehension* stage resulted in successfully patching only 22 vulnerabilities, whereas skipping the *How-To-Fix* stage led to successful patching of 24 vulnerabilities. These outcomes demonstrate that the multi-stage approach adopted by SAN2PATCH is effective, and confirm that each individual component provides a meaningful contribution.

B Benchmark Details

B.1 VulnLoc Benchmark

In this study, we added automated functionality tests to the VulnLoc benchmark to improve patch validation. These tests use unit tests provided by project developers. If any unit tests failed in the pre-patch project, those tests were excluded from the functional test evaluation. However, for projects without unit tests, functionality tests could not be applied. In such cases, we performed a more detailed patch analysis during manual validation. The following CVEs do not have functional tests: gndbug-19784, CVE-2016-8691, CVE-2016-9557, CVE-2016-5844, CVE-2017-5974, CVE-2017-5975, and CVE-2017-5976.

B.2 SAN2VULN Benchmark

In this study, we constructed the SAN2VULN benchmark to evaluate whether SAN2PATCH can realistically handle modern and emerging vulnerabilities. The SAN2VULN benchmark consists of a total of 27 recent vulnerabilities. Further details on the San2Vuln benchmark are provided in Table 7.

Table 7: Detailed information of our SAN2VULN benchmark.

Package	Bug ID	Functional Test	Published Date	Possibly learned by LLM? [†]	Fixed	Reference
libming	CVE-2024-24148	make check -j -k	Jan 13, 2024	✓	✗	https://github.com/libming/libming/issues/308
	CVE-2024-24146	make check -j -k	Jan 13, 2024	✓	✗	https://github.com/libming/libming/issues/307
bento4	NOFIX-2024-002	✗	Dec, 24, 2024	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/990#Bug1
	NOFIX-2024-003	✗	Dec, 24, 2024	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/990#Bug2
	NOFIX-2024-004	✗	Sep 9, 2024	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/980#Bug1
	NOFIX-2024-005	✗	Sep 9, 2024	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/980#Bug2
	NOFIX-2024-006	✗	Dec, 24, 2024	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/989#Bug2
	NOFIX-2024-007	✗	Mar 29, 2024	✓	✗	https://github.com/axiomatic-systems/Bento4/issues/946
	NOFIX-2024-008	✗	May 27, 2024	✓	✗	https://github.com/axiomatic-systems/Bento4/issues/964#Bug2
	NOFIX-2024-009	✗	May 27, 2024	✓	✗	https://github.com/axiomatic-systems/Bento4/issues/964#Bug3
	NOFIX-2024-010	✗	Jan 16, 2025	✗	✗	https://github.com/axiomatic-systems/Bento4/issues/991#Bug2
	NOFIX-2024-001	make check -j'nproc'	Sep 11, 2024	✗	✗	https://github.com/liblouis/liblouis/issues/1632
liblouis	CVE-2022-26981	make check -j'nproc'	Mar 4, 2022	✓	✓	https://github.com/liblouis/liblouis/issues/1171
	CVE-2022-31783	make check -j'nproc'	May 21, 2022	✓	✓	https://github.com/liblouis/liblouis/issues/1214
	GIT-2024-1530	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1530
	GIT-2024-1531	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1531
	GIT-2024-1532	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1532
	GIT-2024-1533	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1533
	GIT-2024-1534	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1534
	GIT-2024-1535	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1535
	GIT-2024-1536	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1536
	GIT-2024-1537	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1537
GIT-2024-1539	make check -j'nproc'	Mar 16, 2024	✓	✓	https://github.com/liblouis/liblouis/issues/1539	
kamilio [‡]	OSV-2024-1206	running unit test	Oct 11, 2024	✗	✓	https://osv.dev/vulnerability/OSV-2024-1206
	OSV-2024-1210	running unit test	Oct 11, 2024	✗	✓	https://osv.dev/vulnerability/OSV-2024-1210
	OSV-2024-1230	running unit test	Oct 19, 2024	✗	✓	https://osv.dev/vulnerability/OSV-2024-1230
	OSV-2024-1244	running unit test	Oct 27, 2024	✗	✓	https://osv.dev/vulnerability/OSV-2024-1244

[†] The training data of the LLM is assumed to be collected as of August 2024, based on the GPT-4o-2024-08-06 model.

[‡] These vulnerabilities were discovered through the OSS-Fuzz project. Therefore, they are triggered in the fuzzing harness.