

LogicEval: A Systematic Framework for Evaluating Automated Repair Techniques for Logical Vulnerabilities in Real-World Software

Syed Md Mukit Rashid, Abdullah Al Ishtiaq, Kai Tu, Yilu Dong, Tianwei Wu, Ali Ranjbar, Tianchang Yang, Najrin Sultana, Shagufta Mehnaz, Syed Rafiul Hussain

The Pennsylvania State University

{sizr5848, abduallah.ishtiaq, kjt5562, yiludong, tvw5452, aranjbar, tzy5088, nks5814, smehnaz, hussain1}@psu.edu

Abstract

Logical vulnerabilities in software stem from flaws in program logic rather than memory safety, which can lead to critical security failures. Although existing automated program-repair techniques primarily focus on repairing memory-corruption vulnerabilities, they struggle with logical vulnerabilities because of their limited semantic understanding of the vulnerable code and its expected behavior. On the other hand, recent successes of large language models (LLMs) in understanding and repairing code are promising. However, no framework currently exists to analyze the capabilities and limitations of such techniques for logical vulnerabilities. We aim to systematically evaluate both traditional and LLM-based repair approaches for addressing real-world logical vulnerabilities. To facilitate our assessment, we created the first-ever dataset, LogicDS, comprising 122 logical vulnerabilities that reflect tangible security impact. We also developed a systematic framework, LogicEval, to evaluate patches for logical vulnerabilities. Evaluations suggest that compilation and testing failures are primarily driven by prompt sensitivity, loss of code context, and difficulty in patch localization.¹

1 Introduction

Logical vulnerabilities in software pose significant risks because they stem from incorrect implementation of the program’s logic/functionality, rather than violations of language safety measures, such as memory corruptions. These vulnerabilities can be exploited to induce critical security and privacy implications, including authentication bypass (Tu et al., 2024), sensitive data leakage (Ranjbar et al., 2025), or system operations disruptions (Dong et al., 2025), often without triggering traditional security defenses, e.g., address sanitizers (Song et al., 2019). Several works (Felmetsger et al., 2010;

Deepa et al., 2018) have focused on identifying logical vulnerabilities in complex software systems and evaluating LLM performance to identify them (Ullah et al., 2023). However, little attention has been given to automatically repairing them, leaving vulnerable software exposed to significant risk.

Automatically repairing logical vulnerabilities presents unique challenges compared to fixing memory-corruption vulnerabilities. First, logical vulnerabilities do not follow consistent, reusable repair templates/patterns (Li et al., 2025). Generating a correct patch often requires deep semantic understanding of the vulnerable code, its surrounding context, and the compromised functionality or property. They also do not necessarily lead to crashes or illegal memory access, thus conventional signals such as compilation logs, runtime logs, or memory sanitizers (Song et al., 2019) provide limited help for localization. Finally, generating effective test cases to assess their patch correctness is difficult because these traditional tools offer little guidance.

Since manual vulnerability repair is labor-intensive and time-consuming (Li and Paxson, 2017), prior work has proposed automated techniques for generating patches (Gao et al., 2021; Jiang et al., 2021; Le Goues et al., 2012). However, most of these approaches target memory-safety violations, making them ineffective or unreliable for addressing logical vulnerabilities. Template-based (Lin et al., 2007; Novark et al., 2007; Cheng et al., 2019) and search-based (Le Goues et al., 2012; van Tonder and Le Goues, 2018) patch generation techniques rely on recognizable repair patterns, which limit generalization to logical vulnerabilities whose patches are diverse and context-specific. Deep learning-based methods (Chen et al., 2019; Jiang et al., 2021) seem a plausible option, but their effectiveness degrades on previously unseen patterns.

Recently, Large Language Models (LLMs) trained on vast training data have demonstrated the

¹To appear in ACL 2026 Main Conference.

ability to generate high-quality outputs across tasks such as text summarization (Jin et al., 2024; Urlana et al., 2024) and question answering (Saito et al., 2024; Yixing et al., 2024). Due to similarities between code and natural language, researchers have explored LLMs for code generation (Ni et al., 2025; Liu et al., 2024a), analysis (Fang et al., 2024), and repair (Jin et al., 2023; Kulsum et al., 2024; Wang et al., 2024). LLMs can capture program syntax and semantics and reason about program behaviors. Recently, LLM-based approaches have attempted to localize vulnerabilities (Li et al., 2024), generate repair patches (Jin et al., 2023; Kulsum et al., 2024; Pearce et al., 2023), and assess patch correctness (Zhou et al., 2024), underscoring LLMs’ potential to repair logical vulnerabilities.

However, to our knowledge, there is no systematic framework for analyzing the capabilities and limitations of automated vulnerability repair (AVR) for logical vulnerabilities. A targeted evaluation is critical for advancing AVR beyond memory-safety issues and toward the more subtle domain of logical vulnerabilities. To this end, we develop a systematic framework for evaluating repair techniques for logical vulnerabilities. We examine the performance of state-of-the-art non-learning-based, learning-based, and off-the-shelf LLM approaches, and study how prompting strategies, repair complexity, and auxiliary vulnerability information influence repair outcomes. We also identify the key challenges LLM-based approaches must address to improve performance.

Unfortunately, existing vulnerability datasets (Just et al., 2014; Gao et al., 2021; Fan et al., 2020; Jimenez et al., 2023) primarily focus on memory-safety bugs and lack representative samples of real-world logical vulnerabilities, thereby limiting their usefulness for our purposes. To address this gap, we first construct LogicDS, the first-ever curated dataset containing 61 real-world logical vulnerabilities including vulnerable and fixed code, vulnerability descriptions, and, when available, behavioral specifications and developer-authored repair rationales.

Building on this dataset, we then design LogicEval, an end-to-end evaluation framework that takes localized vulnerable code and auxiliary inputs (e.g., context, specifications, repair descriptions) and evaluates AVR techniques, including both traditional tools and LLM-based methods, on their ability to synthesize correct patches. LogicEval supports diverse prompt

configurations and enforces structured patch outputs for seamless grafting into source code. It evaluates patches using compilation and testing pipelines and introduces automated reasoning to assess patch quality. Together, these capabilities enable LogicEval to systematically characterize traditional and LLM-based AVR approaches, as well as off-the-shelf LLMs, for patch generation and assessment on logical vulnerabilities. Through extensive empirical analysis, we identify key strengths, limitations, and failure modes of LLMs and offer insights to guide future AVR development for logical vulnerabilities.

2 Logical Vulnerabilities

A logical vulnerability ϑ is a logical flaw in a program P that leads the program to deviate from an expected behavior E and compromises its security in terms of confidentiality, integrity, or availability.

```

1 typedef struct {
2 ...
3 ++ #define CHACHA20_POLY1305_MAX_IVLEN 12
4 ...
5
6 static int chacha20_poly1305_ctrl(EVP_CIPHER_CTX
   ↪ *ctx, int type, int arg)
7 ...
8     return 1;
9
10 case EVP_CTRL_AEAD_SET_IVLEN:
11 - if (arg <= 0 || arg > CHACHA_CTR_SIZE)
12 ++ if (arg <= 0 || arg > CHACHA20_POLY1305_MAX_IVLEN)
13     return 0;
14     ctx->nonce_len = arg;
15     return 1;
16 ...

```

Listing 1: Fix for CVE-2019-1543 (cve, 2019a). Lines prefixed with ++ denote additions, whereas - indicate deletions.

Listing 1 shows a real-world logical vulnerability in OpenSSL’s `chacha20_poly1305_ctrl`, which handles ChaCha20-Poly1305 control operations. RFC 7539 (rfc) requires a unique 96-bit (12-byte) nonce. However, OpenSSL allows variable-length nonces, pads those shorter than 12 bytes with leading zeros, and even accepts lengths up to 16 bytes, using only the last 12 bytes and silently discarding the rest. This improper validation of the `arg` parameter risks nonce reuse and potentially enables severe cryptographic attacks (cve, 2019a).

The expected behavior of logical vulnerabilities is often described by design specifications (e.g., RFCs) or implementation documents. However, they can also stem from implicit knowledge (e.g., privilege escalation in CVE-2024-25420 (cve, 2024)). Logical vulnerabilities may compromise an implementation’s security assumptions even when the program is free from memory-safety issues.

Logical vulnerabilities usually stem from incorrect or missing control flow or validation logic (e.g., skipping required state transitions, omitting essential preconditions, placing checks in the wrong location, or accepting values outside the intended range). They do not exhibit consistent fix patterns. As such, resolving them requires a deep understanding of the program logic and its intended behavior.

3 Related Works

AVR approaches. Automatic Vulnerability Repair (AVR) techniques aim to propose a patch that (i) eliminates an identified vulnerability and (ii) preserves the original functionality after the fix. Existing AVR approaches can be broadly classified into *non-learning-based*, *learning-based*, and *LLM-based* approaches. Among non-learning-based AVR approaches, template-guided approaches (Shaw et al., 2014; Huang et al., 2019; Xing et al., 2024; Zhang et al., 2022) rely on patterns of vulnerability properties or historical patches to generate patches, and thus usually excel where known fix templates can be reliably applied. In contrast, fixes for logical vulnerabilities typically lack such common patterns. Constraint-based approaches extract program constraints through static analysis (Oh, 2018; Chida and Terauchi, 2022), symbolic execution (Shariffdeen et al., 2021), or dynamic analysis (Xuan et al., 2016; Agrawal and Horgan, 1990) against a set of test suites. However, each logical vulnerability typically has its own distinct expected behavior that does not necessarily map to the program structure and often provides only limited exploit traces. Search-based approaches (Le Goues et al., 2012; Marginean et al., 2019; Jiang et al., 2018; Le et al., 2016) explore a hypothetical search space for repair through mutations in existing code or by extracting similar code from other functions or source files within a project; such mutations perform poorly for real-world logical vulnerabilities. Deep learning-based AVR approaches (Chen et al., 2019; Jiang et al., 2021, 2023) treat program repair as a neural machine translation task. Although these approaches perform better than non-learning-based approaches, they require a dataset of vulnerabilities and their fixes for training and attempt to extract recurring repair patterns from source code. Recently, several LLM-based approaches have been proposed to generate repair patches (Jin et al., 2023; Kulsum et al., 2024; Pearce et al., 2023). LLMs appear promising for repairing logical vulnerabilities

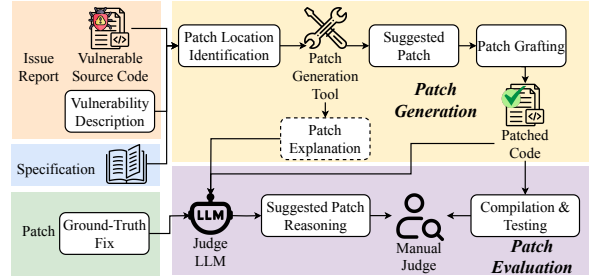


Figure 1: Overview of LogicEval

because they can understand security invariants, though they face limitations when evaluating complex constraints. However, to our knowledge, no prior research has systematically analyzed their capabilities and limitations in this specific context. Further details regarding existing AVR techniques and their limitations are provided in Appendix A. **AVR evaluation frameworks.** Existing evaluations on LLMs for security vulnerabilities assess the security of generated code (Pearce et al., 2025), study LLM assistance in writing code (Sandoval et al., 2023), or evaluate LLMs for vulnerability identification and related security tasks (Deng et al., 2024; Ullah et al., 2023). Prior work also provides taxonomies and benchmarks for AVR approaches (Li et al., 2025), but does not address logical vulnerabilities or the limitations of existing AVR approaches for them. Most related to our work is that of Pearce et al. (Pearce et al., 2023), which evaluates the performance of LLMs towards repairing security vulnerabilities with zero-shot prompts. However, their evaluation framework does not take auxiliary vulnerability information as input, lacks automated metrics to identify reasonable patches, and uses address sanitizers and CodeQL to test patches, which are not suitable for evaluating logical vulnerabilities.

4 LogicEval: Evaluation Framework

To address limitations of existing evaluation approaches, we develop LogicEval, a systematic end-to-end framework that automatically generates and evaluates patches for logical vulnerabilities. We also define metrics to assess patch correctness and quality. Using the generated patches and evaluation results, LogicEval enables analysis of the capabilities and limitations of different AVR approaches. Figure 1 outlines the LogicEval workflow.

4.1 Inputs for LogicEval

We assume the following inputs to LogicEval:

Vulnerable (S) and Fixed (F) Source Code. We provide code from the vulnerable implementation source code S as input to the repair framework under evaluation. After patch generation, we assess patch quality by comparing generated patches against a ground-truth patch F .

Vulnerability Description (D). We assume a natural text description of the vulnerability is available to clarify how it leads to deviations from expected behavior and violates security guarantees.

Behavioral Specification (V_S). In some experimental settings, we additionally provide natural-text behavioral specifications (e.g., extracted from RFCs or formal implementation documentation). Since such specifications are not always available for all logical vulnerabilities, this input is optional and is only provided in certain experiments.

Context (V_{ctx}). We define context V_{ctx} as local variable declarations at the start of the vulnerable function and global/static definitions at the top of the file/class. We optionally provide V_{ctx} in some experiments, as it may help repair techniques generate more plausible fixes.

Compilation (C) and Testing (T) Scripts. We provide a script to compile the full project with any suggested patch. If available, we also provide a testing script T that runs existing testcases against the project, often including a proof-of-concept (PoC) exploit to check whether the vulnerability persists.

4.2 Patch Localization

To generate a patch for a logical vulnerability, a developer must first *locate* the code segment that requires modification (i.e., patch localization). Most existing AVR approaches (Jiang et al., 2023; Kulsum et al., 2024; Jiang et al., 2021; Lutellier et al., 2020) either assume perfect localization or rely on automated fault localization. Prior work on fault localization (Campos et al., 2012; Jones and Harrold, 2005; Abreu et al., 2007) mainly targets general bugs or memory-safety issues and is not designed for logical vulnerabilities, which are harder to localize, as automated techniques (e.g., address sanitizers) cannot identify code that triggers them. Moreover, no existing method achieves perfect accuracy, and localization errors frequently lead to incorrect or ineffective AVR patches (Jiang et al., 2018, 2023). Furthermore, both existing AVR techniques and off-the-shelf LLMs (§6.2) already struggle with logical vulnerabilities. To isolate patch-generation performance, we restrict our evaluation to single-hunk fixes within a single function and as-

sume *perfect patch localization* by manually identifying the precise vulnerable region.

However, real-world patches for security-relevant logical vulnerabilities often span multiple locations. We observe that most such patches contain a *core fix*: a contiguous block of statements that implements the essential logic to eliminate the vulnerability, while other changes (e.g., variable declarations, helper functions, or supporting assignments) primarily support this core fix (examples provided in Appendix H). Hence, we treat repairing the core fix as addressing a single-hunk vulnerability, motivated by (i) evidence that AVR performs better on single-hunk patches (Li et al., 2025) and (ii) the intuition that multi-hunk vulnerabilities can often be handled by generating patches independently at each location. We provide additional examples and our identification procedure in the Appendix B.

4.3 Patch Generation for LogicEval

LogicEval leverages the provided inputs and localized vulnerability to generate patches using the AVR technique under evaluation M . For baseline approaches (Table 7), we provide vulnerable source code V and the required localization (either the vulnerable block V_b or function V_s). When evaluating off-the-shelf LLMs, we construct prompts along the following dimensions using the inputs in §4.1: **Portion of source code.** We provide either the vulnerable block V_b or vulnerable function V_f ; in some experiments, we also include context V_{ctx} .

Auxiliary information. To assist LLM-based repair, we provide one or more of the vulnerability description D , behavioral specification S , and repair description R . We vary these inputs across experiments to measure how auxiliary information affects patch correctness and quality.

LLM Configuration. We evaluate prompting strategies, including zero-shot, few-shot, and chain-of-thought. We also configure temperature and explore task-oriented and role-oriented prompting styles.

Patch grafting. While constructing prompts, we instruct the LLM to place the suggested repair within tags and to output code that can directly replace the provided vulnerable function or block. After receiving the output, we extract the tagged code and replace the annotated vulnerable block or function. **Obtaining explanation for suggested patch.** After each patch-generation, we ask the LLM in a follow-up prompt to provide a brief natural-language ex-

planation of the suggested patch. The resulting explanation E is used for reasoning-based evaluation.

4.4 Patch Evaluation for LogicEval

Evaluating patches for logical vulnerabilities poses unique challenges. Although automated patch validation techniques exist (Tan et al., 2016; Xin and Reiss, 2017; Le-Cong et al., 2023), most rely on static code features or dynamic testing with pre-defined testcases or known exploits. Since logical vulnerabilities do not follow consistent patterns for identification or repair, extracting features that capture their essence is difficult, limiting the applicability of existing automated patch evaluation techniques. Learning-based approaches have also been proposed (Tian et al., 2022, 2023), but the lack of large, high-quality datasets of logical vulnerability patches constrains their effectiveness (Li et al., 2025). More recently, LLM-based validation techniques (Zhou et al., 2024) have emerged, yet their reliability remains questionable due to difficulty in handling complex constraints (Li et al., 2025; Liu et al., 2024b). Several works employ CodeQL queries (cod, 2021) for testing (Sandoval et al., 2023; Pearce et al., 2025, 2023), but crafting queries that capture logical vulnerabilities is non-trivial. Consequently, manual inspection remains the most dependable way to confirm that a proposed patch addresses the underlying logical vulnerability without introducing side effects (Li et al., 2025; Pearce et al., 2025; Sandoval et al., 2023). Based on these considerations, LogicEval evaluates *compilation and testing* and *reasoning*.

4.4.1 Compilation and Testing

LogicEval first checks whether a suggested patch is *compilable* using the compilation script C . Because we focus on the core fix, we graft the suggested patch Z onto the localized function F_f or block F_b over the ground-truth *fixed* implementation F , ensuring that auxiliary modifications needed for compilation and testing are present while focusing only on the core fix. If compilation succeeds, the patch is *compilable*. If a testing script T is available, LogicEval runs it on compilable patches; a patch that passes all tests is *plausible*. However, a plausible patch may still be incorrect due to overfitting. Therefore, we manually inspect all plausible patches to determine whether they are *correct*, i.e., whether they eliminate the vulnerability and introduce no unintended side effects. A

patch that is not plausible cannot be correct.

4.4.2 Reasoning

Aside from compilation and testing, analyzing the steps a patch performs to fix the vulnerability is essential for logical vulnerabilities, especially to assess how *reasonable* the patch is as a suggestion to the developer. To address the lack of an objective automated technique, we use LLM-assisted reasoning metrics similar to prior work on vulnerability identification (Ullah et al., 2023). Since LLMs understand program semantics well, they can identify similarity in rationale between a suggested patch and a ground-truth patch. A high similarity indicates that the candidate patch’s rationale aligns closely with the ground-truth, which we use as a proxy for *reasonable* patches.

Concretely, LogicEval compares the natural-language explanation E of each candidate patch against a reference explanation E_g of the ground-truth fix. We first prompt a “judge” LLM in a zero-shot setting with the vulnerability description D , vulnerable block V_b , and ground-truth fix block F_b to derive E_g . When the patch-generating LLM outputs a suggested fix, it also provides E (§4.3). We then compare E and E_g using cosine similarity (CS) (Ullah et al., 2023), and use a judge LLM (Ullah et al., 2023) to provide a binary verdict (J) on whether they are similar. We also evaluate these reasoning metrics in Section 6.4.

5 LogicDS: Logical Vulnerability Dataset

Previous AVR approaches curated different benchmark datasets for different programming languages such as Java (Just et al., 2014; Bui et al., 2022; Lin et al., 2017), C/C++ (Gao et al., 2021; Nikitopoulos et al., 2021; Fan et al., 2020) and Python (Jimenez et al., 2023). However, we observe that these datasets contain very few logical flaws, and most have no clear security impact (e.g., the changing-offset issue in the Time-3 Defects4J sample (tim)).

We, therefore, focus on security-relevant logical vulnerabilities and build LogicDS, a new dataset containing 61 real-world cases. We identify vulnerabilities caused by deviations from expected behavior that undermine security guarantees by examining CVEs across 28 popular open-source implementations. For each sample, we use the CVE description as D , the fix commit as the fixed version F , and its parent commit as the vulnerable version V . We manually localize the core-fix vulnerable function and block (V_f, V_b), context V_{ctx}

(§4.1), and the corresponding fixed function and block (F_f, F_b). We create a compilation script C , extract relevant specification text S when available, and collect testcases to build a test script T where applicable. All these steps were performed manually to ensure the correctness and reliability of the ground truth dataset. Constructing each vulnerability data point took ~ 10 person-hours. To obtain a repair description R , we use LLMs by providing D, V_b , and F_b , asking for a natural-language description of the repair steps, and then manually refining it.

Synthetic Java Samples. Many state-of-the-art repair techniques (Jiang et al., 2018, 2023; Campos et al., 2012) are tailored to Java constructs and adopt Java-based fault localization, driven by the popularity of Defects4J (tim) and Vul4J (Bui et al., 2022). This limits their direct applicability to other languages (e.g., C/C++). To facilitate logical vulnerability repair research and benchmark Java-focused approaches (Jiang et al., 2018, 2023; Kulsum et al., 2024), we also construct 61 synthetic Java samples derived from our real-world cases, each including V, F , and testcases. Curation details are provided in Appendix C.

6 Evaluations & Observations

In this Section, we present the results and key findings from our LogicEval evaluation. We analyze baseline AVR approaches and also off-the-shelf LLM prompting by varying *source code*, *auxiliary information*, and *LLM configuration* on both real-world and synthetic (LogicDS) vulnerabilities. Note that, for each claim presented in this Section, we also performed statistical significance tests for their justification, which are provided in Appendix G. The code for LogicEval, LogicDS, and the generated patch suggestions are publicly available in (log, 2026).

6.1 Performance of Baseline Approaches

This experiment evaluates the performance, general capabilities, and limitations of existing automatic vulnerability repair (AVR) approaches. We use three state-of-the-art approaches to generate patches for logical vulnerabilities in LogicDS: SimFix (Jiang et al., 2018), a non-learning-based method; KNOD (Jiang et al., 2023), a deep learning-based technique; and VRPilot (Kulsum et al., 2024), an LLM-based program repair approach. Since SimFix and KNOD are Java-focused

Dataset	Tool	C	T	CSL	CSQ	JL	JQ
Real	VRPilot	0.82	0.08	0.65	0.66	0.02	0.07
Synthetic	SimFix	0.01	0.00	0.62	0.64	0.01	0.00
	KNOD	0.35	0.00	0.64	0.65	0.02	0.00
	VRPilot	0.56	0.09	0.65	0.65	0.15	0.03

Table 1: Baseline tool results. “C” and “T” represent the % of generated patches successfully compiled and passed all tests, respectively. “CS” represents average cosine similarity among suggested patches and ground-truth explanations, “J” represents the % of patches a judging LLM determined similar to ground-truth fix. Suffixes “L”, “Q” after each “CS” and “J” represent judging LLM, Llama, and Qwen, respectively. Color heat is relative to all results for the same metric across all our experiments.

repair approaches, we only evaluate them on synthetic Java samples. On the other hand, we evaluate VRPilot on both real-world and synthetic samples of LogicDS. The detailed adoption process for the baselines is discussed in Appendix D.

We present the compilation and testing results, and automated reasoning scores of the baseline approaches in both real-world and synthetic examples in Table 1. VRPilot, being an LLM-based approach, performs better compared to other baseline approaches. However, its CoT-based approach has a lower reasoning score than zero-shot prompting techniques, as we discuss in Section 6.2. Both SimFix and KNOD mostly generate unreasonable patches and use incorrect repair logic. E.g., patches from SimFix (Jiang et al., 2018) and from KNOD (Jiang et al., 2023) generate totally unreasonable logic.

6.2 Performance of Off-the-shelf LLMs

This experiment aims to evaluate the performance of off-the-shelf LLMs to repair logical vulnerabilities using LogicEval. Specifically, we evaluate the performance of LLMs along three dimensions when prompting for logical vulnerabilities: *LLM configuration*, *source code*, and *auxiliary information*. Examples for manual inspection are provided in Appendix H.

LLMs considered and default configurations. We leverage three popular off-the-shelf LLMs: Meta Llama 3.1 (Grattafiori et al., 2024), Qwen 2.5 (Hui et al., 2024), and OpenAI o3-mini (OpenAI, 2025). For each experiment, each LLM is used to generate and evaluate patches using LogicEval for each vulnerability. Note that our work aims not to benchmark all LLMs to repair logical vulnerabilities, but rather to identify the capabilities and

limitations of LLMs in general towards repairing logical vulnerabilities.

In each experiment, we adjust one dimension and set the others to their defaults. We use a temperature of 0.2, role orientation, and zero-shot prompts as the default LLM configuration, as they are the best-performing parameters (or at least equally performing compared to other choices) in their respective dimension as per our experiments described in detail below. Also, we use the vulnerable block V_b as the default provided vulnerable source code and the vulnerability information D as the default provided auxiliary information while varying configurations of other dimensions. We provide the prompt templates used in our experiments in Appendix E. For brevity, in this section, we display only the results for real-world samples in LogicDS across our experiments. The corresponding results for synthetic examples in LogicDS are shown in Appendix F.

For each LLM, we leverage the other two LLMs as judge LLMs, and obtain the cosine similarity scores among explanations, and also judging verdicts (Section 4.4). We display the average cosine similarity scores and percentage of patches whose explanation aligned with the ground-truth fix according to each judge-LLM.

P	Llama-3.1					Qwen-2.5					OpenAI-o3-mini							
	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P1	0.50	0.06	0.81	0.76	0.10	0.35	0.66	0.04	0.81	0.73	0.60	0.53	0.58	0.07	0.77	0.77	0.79	0.32
P2	0.48	0.07	0.81	0.76	0.13	0.33	0.70	0.04	0.81	0.74	0.54	0.54	-	-	-	-	-	-
P3	0.54	0.06	0.81	0.77	0.13	0.36	0.64	0.04	0.80	0.74	0.54	0.52	-	-	-	-	-	-
P4	0.50	0.06	0.81	0.75	0.11	0.49	0.66	0.04	0.80	0.70	0.49	0.51	0.66	0.07	0.79	0.79	0.64	0.41
P5	0.53	0.03	0.82	0.77	0.13	0.44	0.57	0.03	0.80	0.74	0.51	0.46	0.68	0.08	0.77	0.77	0.66	0.28
P6	0.54	0.00	0.82	0.74	0.16	0.36	0.64	0.04	0.80	0.73	0.51	0.54	0.60	0.03	0.47	0.46	0.30	0.11
P7	0.45	0.00	0.73	0.75	0.11	0.34	0.60	0.04	0.75	0.77	0.59	0.48	0.51	0.09	0.74	0.72	0.72	0.38
P8	0.68	0.04	0.68	0.68	0.03	0.10	0.75	0.05	0.64	0.63	0.07	0.07	0.66	0.08	0.63	0.62	0.25	0.10

Table 2: Compilation, testing, and reasoning score results for different LLM configurations in real-world vulnerabilities. Column meanings are the same as in Table 1. ‘‘P’’ represents the prompt IDs. Suffixes ‘‘L’’, ‘‘Q’’, and ‘‘O’’ after each ‘‘CS’’ and ‘‘J’’ represent judging LLM Llama, Qwen, and OpenAI-o3-mini, respectively.

6.2.1 Impact of Adjusting LLM configurations

To assess the impact of LLM configurations on patch generation performance, we adjust the temperature, orientation, and prompting strategy of LLMs while keeping the prompt contents the same. We provide the vulnerable block V_b , the vulnerability description D , and ask the LLM to suggest a patch using LogicEval. To assess temperature, we run patch generation with temperatures 0.2 (low),

0.5 (medium), and 0.9 (high) using prompts **P1**, **P2**, and **P3**, respectively. To assess orientation, we adopt a task-oriented prompt in **P4**. Finally to assess prompting strategy, we consider: (i) zero-shot prompt **P5** with V_b and D ; (ii) few-shot prompt **P6**, where in addition to V_b and D , we provide a real-world sample from LogicEval with its D , V_b , and ground-truth F_b as an example fix; (iii) chain-of-thought (CoT) prompt **P7**, where we first provide V_b and D and ask for steps $R_{\text{suggested}}$ to repair the vulnerability, then provide V_b and $R_{\text{suggested}}$ to obtain a patch; and (iv) CoT prompt **P8**, similar to **P7** except we do not include $R_{\text{suggested}}$ in the second prompt.

We present the compilation and testing results, along with the average reasoning scores, in Table 2 for real-world examples in LogicDS. We observe that the compilation, testing, and reasoning scores for **P1**, **P2**, and **P3** are similar (we cannot adjust the temperature of openAI-o3-mini), indicating *there is minimal effect of logical vulnerability repair performance based on adjusting temperature*. Similarly, the scores for prompt **P4** are also similar to **P1**, indicating *there is minimal effect of logical vulnerability repair performance based on orientation as well*.

When evaluating different prompting strategies, we observe that *usually zero-shot prompt P5 obtains higher compilation percentage and reasoning scores than CoT prompts, especially on real-world examples*. Several compilation errors occur for CoT prompt **P7** because it creates extra undefined variables based on the reasoning suggestion from its first prompt (Listing 15). Prompt **P8** omits all auxiliary information and relies only on the vulnerable source code (V_b). As a result, it generates more compilable patches but achieves significantly lower reasoning scores than both **P5** and **P7**. This indicates *auxiliary information is required alongside the source code to achieve reasonable patches*. Scores for few-shot prompt **P6** are usually marginally lower than zero-shot prompts across the three LLMs; we manually did not observe any contrasting behavior pattern between the two strategies.

6.2.2 Impacts of Adjusting Source Code

We evaluate the advantages and disadvantages of providing different portions of vulnerable source code that are manually localized (See Section 4.2). We either provide V_b , where the core fix is required, or V_f . Providing V_b allows more fine-grained lo-

P	Llama-3.1					Qwen-2.5					OpenAI-o3-mini							
	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P9	0.53	0.07	0.82	0.76	0.08	0.41	0.60	0.07	0.81	0.74	0.54	0.49	0.64	0.14	0.78	0.78	0.70	0.31
P10	0.55	0.07	0.78	0.76	0.02	0.34	0.65	0.08	0.80	0.74	0.54	0.58	0.71	0.16	0.75	0.75	0.61	0.36
P11	0.49	0.06	0.81	0.76	0.10	0.43	0.62	0.11	0.82	0.75	0.49	0.49	0.68	0.14	0.77	0.77	0.72	0.36

Table 3: Compilation, testing, and reasoning score results for varying portions of the source code provided in real-world vulnerabilities. Column meanings are the same as in Table 1 and 2.

calization, whereas V_f provides broader context. We first compare two prompting strategies: **P9**, which supplies only the vulnerable block V_b , and **P10**, which supplies the entire vulnerable function V_f . Across both real-world and synthetic datasets—and for all LLMs tested, **P10** achieves marginally higher compilation and testing success rates, while **P9** has marginally better reasoning scores (the J score differences are not statistically significant, however, the CS scores are, see Appendix G).

Manual inspection reveals that when only the vulnerable block is provided, LLMs sometimes fail to resolve the correct variable names and instead introduce placeholder variables, leading to compilation failures. Supplying the full function context may mitigate the need for placeholder variables. Conversely, when functions are very large, giving the entire function can make it harder for the LLMs to locate the exact block that needs to be modified, occasionally resulting in unreasonable patch suggestions (as in Listing 17), suggesting that *providing only a vulnerable block without context sometimes leads LLMs to create placeholder variables. In contrast, when the entire vulnerable function is provided, LLMs sometimes fail to locate the block to modify to fix the vulnerability.*

We also consider another prompting strategy **P11**, which provides *context* (Section 4.1) and compare the performance with **P9**. We observe a marginal increase in compilation pass rate across all LLMs and sample types (both real-world and synthetic) except for Llama patches over real-world samples. In contrast, reasoning scores do not improve much. This suggests that *providing context somewhat facilitates the compilation process.*

6.2.3 Impacts of Adjusting Auxiliary Information

The goal of this experiment is to observe to what extent different auxiliary information facilitates repairing logical vulnerabilities with LLMs. Hence, we consider different combinations of auxiliary

P	Llama-3.1					Qwen-2.5					OpenAI-o3-mini							
	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P12	0.66	0.04	0.67	0.63	0.02	0.10	0.75	0.05	0.62	0.61	0.11	0.13	0.60	0.07	0.64	0.64	0.20	0.07
P13	0.55	0.03	0.81	0.76	0.11	0.41	0.60	0.04	0.80	0.73	0.59	0.48	0.64	0.14	0.77	0.77	0.70	0.38
P14	0.49	0.00	0.81	0.78	0.18	0.51	0.55	0.00	0.80	0.74	0.69	0.64	0.51	0.09	0.78	0.77	0.72	0.39
P15	0.62	0.07	0.81	0.77	0.46	0.72	0.70	0.08	0.81	0.75	0.79	0.77	0.55	0.13	0.76	0.76	0.84	0.57
P16	0.63	0.00	0.77	0.74	0.33	0.70	0.63	0.00	0.74	0.72	0.67	0.67	0.51	0.08	0.72	0.71	0.72	0.44

Table 4: Compilation, testing, and reasoning scores for varying auxiliary information provided in prompts for real-world vulnerabilities. Column meanings are the same as in Table 1 and 2.

information in each of the five different prompts: prompt **P12** having no auxiliary information (i.e., it only provides the vulnerable block V_b to LLMs and ask for a patch), prompt **P13** provides only the vulnerability information D , prompt **P14** additionally provides a specification text (if available, if not this defaults to **P13**), prompt **P15** provides the repair steps R alongside the vulnerability information D , and prompt **P16** only provides the repair steps R , without the vulnerability information.

The compilation, testing, and reasoning results are shown in Table 4 for real-world samples. Here, first, we observe that prompt **P12** has a much higher compilation rate, but much lower reasoning scores compared to other prompts. Upon manual inspection, we find that without any auxiliary information, LLM tends to treat the vulnerable source code as a possible memory vulnerability issue (as in this use-after-free example provided in Listing 19) and suggests patches to fix those vulnerabilities. These patches pass compilation, but do not meaningfully attempt to repair the logical vulnerability. This suggests that *Auxiliary information about the vulnerability (e.g., vulnerability description, specification text, or repair steps) is essential for generating accurate patches for logical vulnerabilities.*

Furthermore, we observed that in some cases, since specification texts usually describe expected behavior in a different abstraction from the implementation, LLMs tend to create undeclared variables adopted from the text (example provided in Table 20), which is a similar phenomenon we observed while evaluating prompt **P7**. Consequently, Prompt **P13** has much higher reasoning scores compared to prompt **P12**, and prompt **P14** has similar reasoning scores but marginally less compilation rate compared to prompt **P13**.

6.2.4 Overall Observations

Off-the-shelf LLMs perform significantly better when provided auxiliary information, outperforming other baseline approaches. However, the overall

performance leaves room for improvement, especially in a real-world environment where the LLMs generated a correct patch for only 5 out of 61 samples. However, the observations suggest that repair prompts should incorporate localization cues and more relevant context about the vulnerability. To reduce hallucinations, any text included in the prompt should be highly specific and aligned with the actual implementation.

	Llama-3.1					Qwen-2.5					OpenAI-o3-mini												
	P	C	T	CSQ	CSO	JQ	JO	P	C	T	CSL	CSO	JL	JO	P	C	T	CSL	CSQ	JL	JQ		
P17	0.51	0.03	0.56	0.52	0.02	0.07	0.38	0.00	0.60	0.58	0.02	0.08	0.30	0.03	0.56	0.55	0.02	0.00					
P18	0.43	0.00	0.63	0.59	0.00	0.07	0.38	0.00	0.60	0.57	0.05	0.07	0.23	0.05	0.61	0.61	0.05	0.02					
P19	0.43	0.00	0.62	0.59	0.03	0.08	0.38	0.00	0.61	0.60	0.05	0.10	0.17	0.00	0.61	0.60	0.10	0.02					
P20	0.42	0.00	0.70	0.66	0.07	0.16	0.38	0.00	0.64	0.63	0.28	0.20	0.30	0.00	0.69	0.70	0.34	0.05					
P21	0.36	0.00	0.80	0.75	0.10	0.48	0.38	0.00	0.74	0.69	0.67	0.64	0.21	0.00	0.75	0.75	0.75	0.23					

Table 5: Compilation, testing, and reasoning score results using templates provided in (Pearce et al., 2023) in real-world vulnerabilities. Column meanings are the same as in Table 1 and 2.

6.3 Comparison with Other Evaluation Works

Among existing evaluation frameworks for vulnerability repair, Pearce et al. (Pearce et al., 2023) propose a framework for evaluating LLM-generated patches for security vulnerability repair. We use their prompt templates (Table IV in (Pearce et al., 2023)) to generate patches for logical vulnerabilities, namely prompts P17–P21 in Table 9, providing source code as prescribed in (Pearce et al., 2023). Except for P21, the other prompts (P17–P20) do not include vulnerability information and often fail to generate reasonable patches, as reflected by their lower reasoning scores in Table 5 for real-world samples in LogicDS. Prompt P21, which uses the vulnerability description as the message, achieves a reasoning score similar to P1. This further proves that by failing to incorporate auxiliary information, the prompt templates proposed by Pearce et al. are unable to generate more reasonable patches for logical vulnerabilities.

6.4 Effectiveness of Evaluation Metrics for Reasoning

In this experiment, we assess how much our automatically calculated reasoning metrics align with human assessments of patch quality and, in turn, their reliability. We randomly selected 200 off-the-shelf LLM-generated patches from our off-the-shelf evaluation and manually annotated them as *reasonable* or *unreasonable* using the criteria in

Metric	Precision	Recall	F1 Score	Accuracy
Compilation	0.424	0.6	0.497	0.571
J-Qwen	0.773	0.667	0.716	0.8
J-OpenAI	0.5	0.721	0.59	0.701
J-LLaMA	0.547	0.839	0.662	0.677
CodeEmbed 70th (0.944)	0.47	0.508	0.488	0.622
CodeEmbed 90th (0.981)	0.615	0.262	0.368	0.68
RougeL 70th (0.822)	0.452	0.4	0.424	0.62
RougeL 90th (0.957)	0.682	0.214	0.326	0.69
CodeBLEU 70th (0.708)	0.352	0.357	0.355	0.545
CodeBLEU 90th (0.906)	0.579	0.157	0.247	0.665

Table 6: Precision, recall, F1 score, and accuracy across metrics.

Section 4.4.2. Two security-knowledgeable annotators independently labeled each patch as reasonable or unreasonable, blinded to all automated scores. Disagreements were rare (<5%) and resolved through discussion. We evaluate (i) compilation success, (ii) the LLM-as-a-judge binary verdict (J), (iii) cosine similarity between patch code embeddings computed with *Microsoft/unixcoder-base*, (iv) ROUGE-L over patch, and (v) codeBLEU over patch. For each continuous metric, we treat patches above a percentile threshold as inferred reasonable, and report both the 70th and 90th percentile thresholds, including the corresponding threshold values. Table 6 reports precision, recall, F1 score, and accuracy for all metrics.

We observe that compilation is the weakest indicator of patch reasonableness, having the lowest accuracy (0.571). In contrast, the LLM-as-a-judge metric using Qwen achieves the highest accuracy (0.8). While several metrics achieve higher accuracy at the 90th percentile threshold, this is largely driven by class imbalance, as the dataset is naturally skewed toward the unreasonable patch class (130/200). Conservative thresholds tend to label most patches as unreasonable, thereby inflating accuracy. Overall, these results suggest that LLM-as-a-judge is a reasonably reliable metric that aligns most closely with human judgments, although there remains room for improvement.

7 Conclusion

We introduced LogicEval, a systematic evaluation framework that leverages LLMs for patch validation. We also curated LogicDS, a benchmark dataset of logical vulnerabilities. Using LogicEval, we evaluated traditional AVR tools, LLM-based systems, and off-the-shelf LLMs across multiple dimensions and identified some key takeaways that would be helpful for future research.

Limitations

We discuss some of the limitations of `LogicEval`, and our constructed dataset `LogicDS` below.

Size of `LogicDS`. Our benchmark dataset, `LogicDS`, consists of 61 logical vulnerabilities drawn from real-world open-source projects. We focus on real vulnerabilities to reflect the conditions developers actually face and to measure how well existing techniques perform outside of synthetic settings. For our work, since we require providing full source code for evaluation, end-to-end validation of patches through compilation and, when available, executable-based testing, we are forced to limit ourselves to open-source projects to construct `LogicDS`. We curate samples by prioritizing (i) popular and widely used projects (e.g., `OpenSSL`, `GnuTLS`), (ii) logical vulnerabilities with direct security impact (e.g., privilege escalation, location tracking, DoS), and (iii) publicly available commits for both the vulnerable version and its patch, along with build instructions and, if available, testing scripts.

Constructing each sample requires substantial manual effort. For each selected project, we scan CVEs, identify issues that match our definition of logical vulnerabilities, confirm that a corresponding patch is publicly available, and then manually locate the core fix, as discussed in Section B. We additionally ensure the codebase compiles correctly (and run available tests when possible) so that each sample supports reliable compilation checking and practical evaluation. This high curation cost per sample make it challenging to scale `LogicDS` to much larger sizes (e.g., thousands of samples).

Assuming Perfect Localization and its Implications. A real-world security vulnerability mitigation workflow consists of four primary stages: (i) vulnerability detection, (ii) root-cause analysis and localization, (iii) patch generation, and (iv) validation. Each stage introduces distinct technical challenges, and extensive prior work exists across these phases, including automated detection (Ullah et al., 2023), localization (Yu et al., 2024), patch generation (Li et al., 2025), and regression and security testing for validation (Zhou et al., 2024). However, none of these works produces the expected output (i.e., 100% sound and complete). As a result, all prior approaches focus on only one stage (e.g., localization) of the vulnerability mitigation workflow and assume a perfect solution for prior stages (e.g., vulnerability detection).

Consistent with prior work, this work aims to independently evaluate the *patch generation* stage for logical vulnerabilities, isolating it from the effects of vulnerability detection and localization. Thus, we assume perfect vulnerability detection and localization, which can be performed manually or semi-automatically by first applying existing automated techniques and then refining them manually. Without this assumption, localization errors would propagate into the patch-generation stage, and failures could stem from incorrect/partial localization rather than limitations of the patch generator itself, making it difficult to identify the true strengths and weaknesses of repair tools. This is why we adopted perfect localization rather than an end-to-end testing approach that also involves error-prone automated localization.

Adoption of samples from existing repair dataset.

Existing dataset curation efforts (Bhandari et al., 2021; Mei et al., 2024) collect CVEs and public commits based on CWE categories. However, several challenges limit their applicability for our purposes: (i) they do not explicitly target logical vulnerabilities, which often do not map cleanly to specific CWEs, (ii) they do not localize the patch in a commit, which often modifies several files at once where not all of the modifications are related to the vulnerability itself, (iii) these datasets also typically do not directly provide test scripts, which are often provided in commits separate from the patch itself, and (iv) most of their collected samples are memory-corruption vulnerabilities. Consequently, we adopted a manual curation approach to construct `LogicDS`. However, evaluation result of samples in `LogicDS` was still enough to confirm the patterns observed in our evaluations as several patches demonstrated such behavior to confirm our observation. We plan to elaborate `LogicDS` in the future and also make the curation scripts and guidelines publicly available for future elaboration and research.

Adoption of older CVEs. A common concern for generating patches for known CVEs is that LLMs might simply reproduce the fixes they saw during training. However, studies (Jimenez et al., 2023) demonstrated that even if an LLM has encountered earlier versions of the code while training, it cannot memorize and regenerate those patches. Our patch validation analysis and manual observation also support this claim.

Adopting ground-truth fix to reason patches. We measure reasoning by similarity to the ground-truth

patch, which could undervalue correct fixes that implement the same logic differently. However, in our experiments, every plausible patch (i.e., a patch that passed both compilation and testing) scored above the 80th percentile in cosine similarity and was judged similar to the ground truth by the LLM, so we observed no such misclassifications.

Variations in reasoning scores among LLMs. We observe that the reasoning scores differ when we use different judging LLMs. For example, Qwen, as a judge, provides lower reasoning scores across all prompts compared to Llama and OpenAI. However, across different prompt approaches, the scores display the same trend, justifying our assumptions based on the reasoning scores.

Evaluating more baselines and LLMs. We primarily focused on repair baselines that can potentially produce patches that are plausible. Several categories of AVR approaches e.g., template-guided (Shaw et al., 2014; Huang et al., 2019), constraint-based (Chida and Terauchi, 2022; Xuan et al., 2016), and several search-based approaches (Marginean et al., 2019) are inapplicable to apply for repairing logical vulnerabilities primarily due to focusing on specific templates, struggling with path explosion when analyzing source code, or only being able to reuse existing source code. A detailed analysis of several categories of vulnerability repair approaches and their limitations in repairing logical vulnerabilities is presented Appendix A.

Circularity Risk of Overlapping LLM Judges. In our automated reasoning metrics, we used LLMs as judges for many of the metrics (e.g., LLM as binary judge J). However, potentially overlapping LLMs for both patch generation and evaluation may introduce circularity. This can cause the same LLM to generate a patch and then evaluate its own patch suggestion, leading to bias (Panickssery et al., 2024; Wataoka et al., 2024). To avoid this, we explicitly excluded any model from judging patches it generated. Concretely, a patch was evaluated only by Qwen3 and OpenAI models; a Qwen3-generated patch was judged only by Llama and OpenAI, and so on. Since patch generation and validation are performed independently, no LLM ever evaluates its own output in our experiments, mitigating the circularity concern.

Manual efforts. We primarily require manual effort to (i) localize vulnerability and (ii) evaluate patches for reasoning. To reduce the manual effort of inspecting all prompts, we use reasoning metrics to identify patches that score significantly higher un-

der one prompt than the other. To uncover common failure patterns, we manually inspect a fixed subset of patches with low scores. Prioritizing patches according to their reasoning metrics allows us to significantly reduce the number of manually reviewed patches.

Acknowledgements

We thank the anonymous reviewers for their feedback and suggestions. Also, our use of any AI assistant during writing was limited to purely language-related assistance and short-form input.

This work has been supported by the NSF under grants 2145631, 2215017, and 2442825, and the Public Wireless Supply Chain Innovation Fund (PWSCIF) under Federal Award ID Number 51-60-IF007.

References

- <https://github.com/rjust/defects4j/blob/master/framework/projects/Time/patches/3.src.patch>. [link].
- Defects4J Time Patch 3. <https://github.com/rjust/defects4j/blob/master/framework/projects/Time/patches/3.src.patch>.
- Fix for cve-2014-0224. <https://github.com/openssl/openssl/commit/bc8923b1ec9c467755cd86f7848c50ee8812e441>.
- National Vulnerability Database. <https://nvd.nist.gov/>.
- RFC 7539. <https://datatracker.ietf.org/doc/html/rfc7539>.
- 2015. CVE-2015-1793. <https://nvd.nist.gov/vuln/detail/CVE-2015-1793>.
- 2017. CVE-2017-3142. <https://nvd.nist.gov/vuln/detail/CVE-2017-3142>.
- 2019a. <https://nvd.nist.gov/vuln/detail/CVE-2019-1543?cpeVersion=2.2>. [link].
- 2019b. <https://github.com/openssl/openssl/commit/c62896c2c0cbd47ab01693d403e37fe5fe15aab8>. [link].
- 2019c. <https://github.com/openssl/openssl/commit/ee22257b1418438ebaf54df98af4e24f494d1809>. [link].
- 2019d. <https://github.com/openssl/openssl/commit/a4f0b50eafb256bb802f2724fc7f7580fb0fbabc>. [link].

- 2019e. https://github.com/openssl/openssl/blob/c62896c2c0cbd47ab01693d403e37fe5fe15aab8/crypto/evp/e_chacha20_poly1305.c. [link].
- 2019f. https://github.com/openssl/openssl/blob/ee22257b1418438ebaf54df98af4e24f494d1809/crypto/evp/e_chacha20_poly1305.c. [link].
- 2019g. CVE-2019-1543. <https://nvd.nist.gov/vuln/detail/CVE-2019-1543>.
2021. Codeql for research. <https://securitylab.github.com/tools/codeql/>.
2022. CVE-2022-1434. <https://nvd.nist.gov/vuln/detail/CVE-2022-1434>.
- 2023a. [link].
- 2023b. CVE-2023-48795. <https://nvd.nist.gov/vuln/detail/CVE-2023-48795>.
2024. CVE-2024-25420. <https://nvd.nist.gov/vuln/detail/CVE-2024-25420>.
2026. Logiceval repository. <https://github.com/SyNSec-den/LogicEval>.
- Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.
- Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256.
- Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39.
- Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 464–468.
- José Campos, André Ribeiro, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, pages 378–381.
- Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.
- Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiguang Sun. 2019. Automatic integer error repair by proper-type inference. *IEEE Transactions on Dependable and Secure Computing*, 18(2):918–935.
- Nariyoshi Chida and Tachio Terauchi. 2022. Repairing dos vulnerability of real-world regexes. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2060–2077. IEEE.
- Ganesan Deepa, P Santhi Thilagam, Amit Praseed, and Alwyn R Pais. 2018. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109.
- Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864.
- Yilu Dong, Tianchang Yang, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Ali Ranjbar, Kai Tu, Tianwei Wu, Md Sultan Mahmud, and Syed Rafiul Hussain. 2025. Corecrisis: Threat-guided and context-aware iterative learning and fuzzing of 5g core networks. In *34th USENIX Security Symposium (USENIX Security 25)*.
- Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512.
- Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, and 1 others. 2024. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846.
- Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2010. Toward automated detection of logic vulnerabilities in web applications. In *19th USENIX Security Symposium (USENIX Security 10)*.
- Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

- Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE symposium on security and privacy (SP)*, pages 539–554. IEEE.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309.
- Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1251–1263. IEEE.
- Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Hanlei Jin, Yang Zhang, Dan Meng, Jun Wang, and Jinghua Tan. 2024. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods. *arXiv preprint arXiv:2403.02901*.
- Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656.
- James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282.
- René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.
- Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d’Amorim. 2024. 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.
- Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE.
- Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Transactions on Software Engineering*, 49(6):3411–3429.
- Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *2012 34th international conference on software engineering (ICSE)*, pages 3–13. IEEE.
- Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215.
- Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Difix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 602–614.
- Ying Li, Gang Wang, Yuan Tian, and 1 others. 2025. Sok: Towards effective automated vulnerability repair. *arXiv preprint arXiv:2501.18820*.
- Yue Li, Xiao Li, Hao Wu, Yue Zhang, Xiuzhen Cheng, Sheng Zhong, and Fengyuan Xu. 2024. Attention is all you need for llm-based code vulnerability localization. *arXiv preprint arXiv:2410.15288*.
- Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56.
- Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 329–340.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

- Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. 2024b. Exploring {ChatGPT's} capabilities on vulnerability management. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 811–828.
- Yu Liu, Sergey Mehtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program repair guided by datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1216–1228.
- Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114.
- Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE.
- Sergey Mehtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701.
- Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupe, Hammond Pearce, Brendan Dolan-Gavitt, and 1 others. 2024. Arvo: Atlas of reproducible vulnerabilities for open source software. *arXiv preprint arXiv:2408.02153*.
- Ziyi Ni, Yifan Li, Ning Yang, Dou Shen, Pin Lyu, and Daxiang Dong. 2025. Tree-of-code: A self-growing tree framework for end-to-end code generation and execution in complex tasks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 9804–9819.
- Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1565–1569.
- Gene Novark, Emery D Berger, and Benjamin G Zorn. 2007. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11.
- Hakjoo Oh. 2018. Memfix: Static analysis-based repair of memory deallocation errors for c. In *FSE 2018: ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM.
- OpenAI. 2025. [Openai o3 and o4-mini system card](#).
- Arjun Panickssery, Samuel R Bowman, and Shi Feng. 2024. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802.
- Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105.
- Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.
- Ali Ranjbar, Tianchang Yang, Kai Tu, Saaman Khalilolahi, and Syed Rafiul Hussain. 2025. [Stateful Analysis and Fuzzing of Commercial Baseband Firmware](#). In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1120–1139, Los Alamitos, CA, USA. IEEE Computer Society.
- Kuniaki Saito, Kihyuk Sohn, Chen-Yu Lee, and Yoshitaka Ushiku. 2024. Unsupervised llm adaptation for question answering. *arXiv preprint arXiv:2402.12170*.
- Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2205–2222.
- Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 390–405.
- Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically fixing c buffer overflows using program transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 124–135. IEEE.
- Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE.
- Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738.

- Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2023. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–34.
- Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13.
- Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain. 2024. Logic gone astray: A security analysis framework for the control plane protocols of 5g basebands. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3063–3080.
- Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2023. Can large language models identify and reason about security vulnerabilities? not yet. *arXiv preprint arXiv:2312.12575*.
- Ashok Urlana, Pruthwik Mishra, Tathagato Roy, and Rahul Mishra. 2024. Controllable text summarization: unraveling challenges, approaches, and prospects—a survey. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 1603–1623.
- Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, pages 151–162.
- Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2081–2107.
- Koki Wataoka, Tsubasa Takahashi, and Ryokan Ri. 2024. Self-preference bias in llm-as-a-judge. *arXiv preprint arXiv:2410.21819*.
- Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE.
- Yunlong Xing, Shu Wang, Shiyu Sun, Xu He, Kun Sun, and Qi Li. 2024. What {IF} is not enough? fixing null pointer dereference with contextual check. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1367–1382.
- Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2397–2414.
- Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55.
- Peng Yixing, Quan Wang, Licheng Zhang, Yi Liu, and Zhendong Mao. 2024. Chain-of-question: A progressive question decomposition approach for complex knowledge base question answering. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 4763–4776.
- Jinhong Yu, Yi Chen, Di Tang, Xiaozhong Liu, XiaoFeng Wang, Chen Wu, and Haixu Tang. 2024. Llm-enhanced software patch localization. *arXiv preprint arXiv:2409.06816*.
- Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. 2022. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 190–201.
- Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883.
- Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353.

Appendices

A Characterizing AVR Approaches and Evaluation Frameworks for Logical Vulnerabilities

A.1 Automatic Vulnerability Repair (AVR)

Automatic Vulnerability Repair (AVR) techniques aim to automatically fix software vulnerabilities. Their primary objective is to propose a patch that (i) eliminates the identified vulnerability and (ii) preserves the original functionality after the fix. In practice, vulnerability repair comprises three main stages: *vulnerability localization*, *patch generation*, and *patch validation*, which mirror the steps developers naturally follow when repairing vulnerable code.

Category	Optimal Scenario (When it Works)	Adaptable for Logical Vulnerability Repair	Limitations When Applying for Logical Vulnerabilities
Template guided	When the vulnerability fix follows a template/pattern	No	Logical vulnerabilities does not follow any general fix pattern.
Constraint based	When the vulnerability can be fixed through enforcing constraints easily extractable from source code	Yes, but difficult	Constraint extraction for logical vulnerabilities struggle with path explosion in large codebases and often lacks the necessary specifications or testcases to derive required constraints.
Search based	When the vulnerability can be fixed by reusing or slightly adapting a similar code segment elsewhere in the source project	Yes, but achieves poor performance	Usually logical vulnerabilities cannot be fixed by reusing existing code, as each one has distinct expected behavior and requires its own custom logic.
Deep learning based	When the vulnerability fix follows a pattern learnable by deep learning models	Yes, but achieves poor performance	Usually logical vulnerabilities do not follow such recognizable pattern

Table 7: Analysis of existing AVR techniques and limitations towards repairing logical vulnerabilities

In this work, evaluate the *patch generation* phase of logical vulnerabilities. We identify the primary challenges for automatically generating fixes for such vulnerabilities, design frameworks to systematically evaluate the correctness of the generated fixes, and discuss the strengths and limitations of existing non-LLM and LLM approaches.

Existing AVR approaches primarily focus on repairing memory corruption vulnerabilities and can be broadly classified into two categories: *non-learning-based* and *learning-based* approaches. A recent study by Li et al. (Li et al., 2025) provides a comprehensive analysis of various categories of AVR frameworks in addressing those vulnerabilities. In this section, we build upon previous studies by examining each category of AVR approaches when applied to logical vulnerability repair, highlighting the unique challenges and limitations they face in this context, which are summarized in Table 7.

A.1.1 Non-Learning Based Approaches

These non-machine learning based methods fall into three categories based on their repairing strategy: *template-guided*, *search-based*, and *constraint-based* approaches.

Template-guided approaches. Template-guided approaches rely on patterns of vulnerability properties (Shaw et al., 2014; Huang et al., 2019) or historical

patches (Xing et al., 2024; Zhang et al., 2022) to generate patches. Due to their reliance on fixed patterns, these methods usually focus on vulnerabilities exhibiting recurring structures (e.g., buffer overflow (Shaw et al., 2014), null pointer dereference (Xing et al., 2024), API misuses (Zhang et al., 2022)) where known fix templates can be reliably applied. In contrast, logical vulnerabilities typically involve highly context-specific behaviors and lack such common patterns, rendering these approaches inapplicable.

Constraint-based approaches. These methods extract program constraints that need to be satisfied to eliminate a vulnerability. These constraints are either extracted through (i) static analysis (Oh, 2018; Chida and Terauchi, 2022), (ii) symbolic execution (Shariffdeen et al., 2021), or (iii) dynamic analysis through running against a set of test suites (Xuan et al., 2016; Agrawal and Horgan, 1990). (i) Among *static analysis-based* techniques, Lee et al. (Oh, 2018) leverage tpestate checking to detect memory vulnerabilities. Liu et al. (Liu et al., 2023) extract and modify Datalog facts to repair Java null-pointer exceptions, Python data leaks, and Solidity smart contract flaws, whereas Xu et al. (Xu et al., 2020) build a mathematical model from historical patches and use automated reasoning to generate hotpatches. However, these

approaches rely on carefully defined properties to drive the static analysis and can only target specific vulnerability classes. Each logical vulnerability has its own distinct expected behavior, and formulating a corresponding property would require domain knowledge and significant manual effort comparable to writing the repair manually. (ii) Symbolic execution-based methods derive the constraints needed to satisfy given testcases (Shariffdeen et al., 2021; Mechtaev et al., 2016). Consequently, they rely on a comprehensive, vulnerability-specific test suite to extract precise constraints. However, real-world logical vulnerabilities often provide only a single or no exploit trace, which is insufficient to recover the necessary constraints (Li et al., 2025). (iii) Dynamic testing-based approaches (Xuan et al., 2016; Agrawal and Horgan, 1990) also require a comprehensive test suite focusing on the vulnerability. Gao et al. (Gao et al., 2021) eliminate this requirement by using address sanitizers to extract crash-free constraints. However, logical vulnerabilities rarely cause program crashes and cannot be captured using address sanitizers, rendering this approach inapplicable.

Search-based approaches. Search-based approaches explore a hypothetical search space for repair through mutations in existing code (Le Goues et al., 2012; Marginean et al., 2019) or by extracting similar code from other functions or source files within a project (Jiang et al., 2018; Le et al., 2016). However, their ability to repair a vulnerability depends on the availability of similar code in their search base that can either be directly placed or adopted with minimal changes to fix a vulnerability. In contrast, it is usually challenging to find similar code that can be used to fix a real-world logical vulnerability, as each logical vulnerability has a distinct expected behavior and thus distinct logic that can be used to fix it. As a result, although we can technically apply search-based mechanisms to repair logical vulnerabilities, their performance is poor, which we demonstrate in this paper using SimFix (Jiang et al., 2018) in §6.1.

A.1.2 Learning Based Approaches

Learning-based approaches pivot on machine learning techniques and can be broadly divided into two categories: deep learning-based and large language model (LLM)-based approaches.

Deep learning-based approaches. Deep learning-based approaches adopt deep learning models and treat the program repair problem as a neural ma-

chine translation (NMT) problem, providing the faulty code as input and obtaining the fixed code as output. In this direction, SequenceR (Chen et al., 2019) is the first to adopt an encoder-decoder based supervised recurrent neural network (RNN) machine translation model to generate patches. Later on, CURE (Jiang et al., 2021) improves on previous techniques on NMT-based program repair through subword tokenization and a code-aware token search strategy to provide more accurate patches. A recent work, KNOD (Jiang et al., 2023), further increases the accuracy on Java program vulnerabilities through leveraging a novel tree-based decoder and emitting an abstract syntax tree (AST) to repair programs.

Although these approaches perform better than non-learning-based approaches (Lutellier et al., 2020; Jiang et al., 2023) they require a dataset of vulnerabilities and their fixes to train the model. Furthermore, while training, the model attempts to extract patterns in source code that can be used to generate patches for specific vulnerabilities. Intuitively, these models work well when the underlying vulnerability can be fixed using a language safety pattern (e.g., bounds checking for buffer overflow vulnerabilities). In contrast, logical vulnerabilities do not follow any patterns. As we show in §6.1, the state-of-the-art repair framework KNOD with a deep learning approach perform poor in fixing logical vulnerabilities.

A.1.3 Large Language Models for AVR

Recently, Large Language Models (LLMs) trained on vast training data have shown the ability to generate high-quality natural language outputs that are widely applicable across diverse areas such as text summarization (Jin et al., 2024) and question answering (Saito et al., 2024). Due to similarities between code and natural language, researchers have explored LLMs in programming-related tasks such as code generation (Liu et al., 2024a) and analysis (Fang et al., 2024), and have been shown to be effective in understanding program syntax and semantics and analyzing program behaviors.

Several LLM-based approaches have been recently proposed to generate repair patches (Jin et al., 2023; Kulsum et al., 2024; Pearce et al., 2023). However, their capabilities and limitations are unknown for logical vulnerabilities. Also, LLMs have the potential to understand security invariants, even though they face limitations when computing complex constraints (Li et al., 2025).

As such, LLMs and LLM-based approaches appear promising for repairing logical vulnerabilities. However, to our knowledge, no prior research has systematically analyzed their capabilities and limitations in this specific context.

A.2 Limitations of AVR Evaluation Frameworks

Among existing evaluations on LLMs for security vulnerabilities, Pearce et al. (Pearce et al., 2025) assess the security of code generated by GitHub Copilot. Sandoval et al. (Sandoval et al., 2023) conducted a user study in which students used LLM assistance to write code, and the resulting programs were evaluated for security vulnerabilities. Deng et al. (Deng et al., 2024) propose and evaluate the use of LLMs for penetration testing. Ullah et al. (Ullah et al., 2023) automatically evaluates LLMs with various prompts to identify security vulnerability in source codes. However, none of them particularly addresses repairing vulnerabilities. Jimenez et al. (Jimenez et al., 2023) evaluate how LLMs generate fixes for GitHub issues; however, most of these issues are not labeled to indicate whether they address security vulnerabilities. Among evaluation works that focus on automatically repairing security vulnerabilities, Li et al. (Li et al., 2025) systematically study existing AVR approaches, and provide a taxonomy, discussing pros and cons of each category of approaches. They also benchmark several existing works based on existing datasets. However, they do not focus on logical vulnerabilities and the limitations of using existing AVR approaches when applied to logical vulnerabilities.

Most related to our work is that of Pearce et al. (Pearce et al., 2023), which evaluates the performance of LLMs towards repairing security vulnerabilities with zero-shot prompts. However, their evaluation framework have several limitations towards evaluating logical vulnerabilities: (i) Their evaluation approach do not take auxiliary information related to the vulnerability, such as vulnerability description, specification, or repair steps, etc., as input, which facilitates generation of reasonable patches for logical vulnerabilities (§6.2.3), (ii) They lack any automated metrics for evaluation to indicate reasoning / identify reasonable patches. They only manually evaluate some patches of the Extractfix (Gao et al., 2021) dataset and label them as identical, semantically equivalent, or reasonable, and (iii) They use address sanitizers and CodeQL (cod, 2021) to test patches, which are not suitable

for logical vulnerabilities.

B Locating The Core Fix of a Patch

```

1 int ssl3_accept(SSL *s)
2 ...
3 ++ #define SSL3_FLAGS_CCS_OK 0x0080;
4 ...
5     case SSL3_ST_SR_FINISHED_B:
6 ++ s->s3->flags |= SSL3_FLAGS_CCS_OK;
7 ...
8
9 int ssl3_read_bytes(SSL *s, int type, unsigned
    ↪ char *buf, int len, int peek)
10 ...
11 ++ if (!(s->s3->flags & SSL3_FLAGS_CCS_OK))
12 ++ {
13 ++     al = SSL_AD_UNEXPECTED_MESSAGE;
14 ++     SSLerr(SSL_F_SSL3_READ_BYTES,
15 ++         SSL_R_CCS_RECEIVED_EARLY);
16 ++     goto f_err;
17 ++ }
18 ++
19 ++ s->s3->flags &= SSL3_FLAGS_CCS_OK;
20 ...
21 }

```

Listing 2: Fix for CVE-2014-0224 (hen).

We define the core fix as the portion of the patch that implements the decision logic needed to eliminate a vulnerability (e.g., the key authorization/validation/state-transition guard). As an example of a core fix, Listing 2 shows the fix for CVE-2014-0224 (hen), where we observe a macro definition at Line 3, assignment of the macro to a variable at lines 6 and 19, and a check based on the variable and macro at lines 11–17. However, the primary fix is to add the check at lines 11–17, which enforces a check based on the flag and ensures that a ChangeCipherSpec message is received when expected. We refer to the *if* block at lines 11–17 as the *core fix*. In C/C++ and Java, the core fix typically corresponds to a compound statement enclosed in braces, with the function body as the top-level block.

During identification and annotation, two security-knowledgeable human annotators selected the hunk(s) they believed contained this decision logic. Disagreements during annotation were rare (< 5%) and were resolved through discussion to reach a consensus. We manually identify and localize the core fix by observing the vulnerable and fix commits, the vulnerability description D , and, if available, the behavioral specification S .

During manual localization, we identify both the *enclosing function* and the *enclosing block*. The enclosing block provides fine-grained context for patch generation, while the enclosing function supplies the broader program context needed to understand the fix. To localize the enclosing function, we extract its full body and signature. To local-

ize the enclosing block, we identify the smallest brace-enclosed region that contains the core fix. If the block appears within a conditional or loop, we include the associated control structure (e.g., the condition). In general, this block may range from a small basic block to the entire function body.

When the fix logic is split across multiple hunks that are jointly required (e.g., a flag assigned in one place and enforced in another), in cases where possible, we expand the core-fix span to include all complementary hunks necessary to realize the intended security behavior, and treat that expanded region as the single-hunk core fix target for our evaluation. For example, in the fix presented in Listing 3 (representative of CVE-2023-0465, simplified for exposition), we place the entire block within the for loop as the core fix.

```

1 - int i;
2 + int i, saw_err = 0;
3
4 // - - -start of hunk - - -
5 - for (i = 1; i < N; i++) \{
6 + for (i = 0; i < N; i++) \{
7   item = chain[i];
8 + if (item.has_invalid_policy) saw_err = 1;
9   FAIL_IF(item.has_invalid_policy);
10  \}
11
12 + if (!saw_err) return internal_error();
13 // - - - end of hunk - - -
14 return OK;

```

Listing 3: Simplified fix from CVE-2023-0465

In cases where two similar hunks in different code locations implement essentially the same decision logic for different modes/configurations, we select one of them as the representative fix, since both reflect the same underlying repair logic. Finally, if the main logic resides in a newly added helper function, we treat the helper function body as part of the core-fix span (i.e., include it within the single localized region).

We perform this localization on both the vulnerable source code S and the fixed code F , resulting in the corresponding localized blocks and functions: V_b , V_f , F_b , and F_f . The repair framework receives V_b or V_f as input, while F_b and F_f are used solely for evaluation.

In some cases, such as the fix for CVE-2023-5363 (cve, 2023a), the patch involves inserting logic between two basic blocks within a much larger compound statement. Including the entire enclosing block in such cases would dilute the relevance of the context. To preserve focus, if the enclosing block exceeds a token threshold (2048 in our case), we instead extract a focused span

centered on the patch location, including the immediate predecessor and successor basic blocks surrounding the patch location (e.g., lines 227–250 in the vulnerable code for CVE-2023-5363). This threshold was intended to avoid context dilution while staying within a practical prompt token budget for some LLMs. This ensures precise and focused block-level localization.

C Curation of LogicDS

For dataset curation, we first search CVEs of the selected targets from the years 2010 to 2024, and based on the CVE description and publicly available fix, we determine whether the vulnerability is a logical vulnerability and has security implications. While selecting vulnerability samples, we exclude memory-corruption vulnerabilities (e.g., buffer overflow) and vulnerabilities that are more related to safe programming practices rather than a specific logical issue (e.g., SQL injection). Then, we manually searched the internet to obtain the developer-provided fix commit for the open-source implementation, by examining commit logs and open source vulnerability websites (e.g., (bug; nvd)). A breakdown of the open-source projects used to construct LogicDS is given in (log, 2026).

An example of the different portions of a sample is demonstrated in Table 8.

C.1 Constructing Synthetic Java Samples.

Manually creating such samples would require significant time and efforts. Thus, to alleviate the manual work, we develop an LLM-assisted semi-automated approach for synthetic sample creation, which will also help future research works in creating synthetic Java examples from real-world logical vulnerabilities written in other programming languages.

In this approach, we first manually create one synthetic sample based on a real-world vulnerability by replicating the vulnerable and fixed functions V_f & F_f , with an intent to preserve original variable and function names where possible. We also write testcases that validate the created patch. Once we create one such example, we leverage LLMs to create subsequent examples, by adopting a few-shot technique. We do that by providing the manually created sample, and prompting it to generate similar synthetic examples given a vulnerable real-world function and vulnerability description. In the subsequent prompt, we provide the gener-

Input	Mandatory	Example
Vulnerable Source Code	✓	Commit (cve, 2019b) which is the parent commit of the fix in (cve, 2019c)
Fixed Source Code	✓	Commit (cve, 2019c) available in national vulnerability database (cve, 2019a)
Vulnerability Description	✓	CVE description for CVE-2019-1543 found in (cve, 2019a)
Behavioral Specification	✗	Extracted from RFC 7539 (rfc) Section 2.8
Context Lines	✗	Lines 21-31 and 140-147 in <i>crypto/x509/x509_vfy.c</i> in (cve, 2019b)
Repair Description	✗	A text description generated by observing the fixed source code (cve, 2019c).
Compilation Scripts	✓	Compilation script generated manually following implementation documents
Testing Scripts	✗	Test found (cve, 2019d) by manually checking the fix commit (cve, 2019c)
Faulty Function and block	✓	Lines 323-429 and 359-363 respectively in <i>crypto/x509/x509_vfy.c</i> in (cve, 2019e)
Fixed Function and block	✓	Lines 500-607 and 537-541 respectively in <i>crypto/x509/x509_vfy.c</i> in (cve, 2019f)

Table 8: Example input and vulnerability localization (CVE-2019-1543 (cve, 2019a)) for LogicEval.

ated vulnerable Java function and the real-world V_b and F_b . Finally, we prompt the LLM to generate testcases for validation. We also assist the LLM iteratively generate syntactically correct Java code by providing compilation logs in case of any errors. This few-shot technique assists the LLM in creating precise synthetic examples when provided with a real-world sample by clearly demonstrating the task at hand. Note that, this semi-automated technique only intended to alleviate the manual work. Although the generated samples needed to be manually verified after generation, this approach is significantly less time-consuming than creating synthetic samples from scratch.

D Adopting Baselines for LogicEval

Existing work on program repair can be broadly categorized into three main paradigms: (i) non learning-based approaches, (ii) deep learning-based approaches, and (iii) large language model (LLM)-based approaches. To evaluate the capabilities and limitations of each, we select representative state-of-the-art methods from each category. We aim to analyze their capabilities in repairing logical vulnerabilities and derive insights that can be leveraged in future automated repair frameworks, focusing on them.

Non Learning-Based Program Repair. We chose a state-of-the-art search-based method, SimFix (Jiang et al., 2018), among non-learning-based approaches. Template-based methods focus on specific classes of bugs and are unsuitable due to the diverse nature of logical vulnerabilities, whereas constraint-based methods struggle with path explosion in large projects and are often inapplicable due to the lack of necessary specifications or test cases.

SimFix operates by mining AST-level modifications from a corpus of buggy programs and their corresponding patches. During inference, it analyzes the buggy project and attempts to replace faulty code snippets with similar code snippets from previously mined patches. Our experiments allow SimFix to generate up to 100 patches per bug. Also, since SimFix works with line-level localization, LogicEval provides a manually located line within the core fix most suitable as the fault location. Furthermore, we allow SimFix to collect candidate patches from the largest project in Defects4J (Just et al., 2014), Closure, providing a sufficiently large search space for collecting candidate patches.

Deep Learning-Based Program Repair. For deep learning-based repair, we evaluate *KNOD* (Jiang et al., 2023), a state-of-the-art deep learning-based framework, due to its superior performance com-

ID	Description
P1-P3	Temperature set to 0.2, 0.5, and 0.9, respectively
P4	Task-oriented prompt
P5	Zero-shot prompt (same as P1)
P6	Few-shot prompt including an example real-world vulnerability description, and vulnerable and fixed blocks
P7	Chain-of-thought prompting, first prompt obtains a repair suggestion $R_{suggestion}$, which is placed in the second prompt that asks for patch
P8	Same as P7 , but $R_{suggestion}$ is not added to the second prompt
P9	Provides vulnerable block V_b as input source code (same as P1)
P10	Provides entire vulnerable function V_f as input source code
P11	Provides vulnerable block along with additional context lines $V_{context}$
P12	Provides no auxiliary information, only the vulnerable block V_b
P13	Provides the vulnerability description D (same as P1)
P14	Provides the vulnerability description D and the specification S
P15	Provides repair steps R and vulnerability description D
P16	Provides the repair steps R only, does not provide D
P17	Obtained from (Pearce et al., 2023), vulnerable code provided with no help (n.h.)
P18	Obtained from (Pearce et al., 2023), deletes V_f and adds a comment “bugfix: fixed logical vulnerability” (s.1).
P19	Obtained from (Pearce et al., 2023), deletes V_f and adds comment “fixed logical vulnerability” (s.2).
P20	Obtained from (Pearce et al., 2023), after a “// BUG: logical vulnerability” comment, V_b is shown in commented-out form, immediately followed by a “// FIXED:” section. (c.)
P21	Obtained from (Pearce et al., 2023), first include a “// BUG: logical vulnerability” comment, then a “// MESSAGE: D ” line. Next, show V_b as commented-out lines, and finally introduce the corrected code under a “// FIXED VERSION:” header. (c.m.)

Table 9: Prompt Descriptions for Different Prompts Tested With Off-the-shelf LLMs. For prompts **P1** to **P16**, by default, the vulnerable block V_b is provided as source code,

pared to others (Jiang et al., 2021; Li et al., 2020; Zhu et al., 2021), on standard benchmarks (Just et al., 2014; Lin et al., 2017). KNOD uses a three-stage tree decoder to generate repaired Java AST patches, and incorporates domain-specific rules to enhance the accuracy of AST node predictions. KNOD defaults to an output token limit of 512; therefore, we only provide the vulnerable block V_b as input.

LLM-Based Program Repair. We consider *VRPilot* (Kulsum et al., 2024) as our baseline for LLM-based repair methods because of its robust performance and widely adoption in recent program repair evaluations (Li et al., 2025). *VRPilot* leverages chain-of-thought (CoT) prompting first to elicit reasoning about a required fix and then generate candidate patches. It refines patch quality through iterative feedback using logs from compilation errors, functional test failures, and security tests until a plausible patch is identified or a pre-

defined iteration budget is reached. Since we are testing for logical vulnerabilities, and through our experiments (§6.2.3) we observed that providing D significantly improves performance, we add D in *VRPilot*’s initial reasoning prompt.

In addition to baseline-specific adaptations, when evaluating a suggested patch from an adapted baseline for reasoning, `LogicEval` prompts the judge LLM with the vulnerability description D , the vulnerable block V_b , and the suggested patch to extract an explanation E for the patch.

E Prompts Used for LLM-based Experiments

The prompts used in our experiments are summarized in Table 9.

As an example of a concrete prompt, we provide the template of prompt **P7**:

...
Here is a portion of a code:

```

<code>
(Vulnerable source code block or function)
</code>
Here is a description of the specification telling
what to do:
<specification>
(Natural-text specification)
</specification> Here is a description of the
vulnerability:
<vulnerability>
(Vulnerability description)
</vulnerability>
Provide a repair for the mentioned code snippet
to fix the vulnerability.
Provide repaired code exactly to replace the
original buggy code mentioned between <code> and
</code>.
Provide repaired code between <repair> and
</repair> tags.
The code between <repair> and </repair> will be
directly copied to replace the code between <code>
and </code>.
...

```

Here, we also test with prompts **P15** and **P16** with a repair description. From the results of the suggested patches of these prompts shown in Table 4, we observe that they have much higher scores in all metrics, indicating LLMs can generate more reasonable patches if concrete repair steps can be provided. We do not focus on the reasoning scores for these two prompts, since the repair steps of ground-truth fixes were already provided to the LLM under testing.

F Results of Experiments in Synthetic Examples in LogicDS

In this Section, we present the results from the synthetic examples in LogicDS for our experiments in Section 6.

	Llama-3.1					Qwen-2.5					OpenAI-o3-mini								
	P	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P1	0.27	0.05	0.81	0.76	0.12	0.42	0.17	0.07	0.80	0.76	0.48	0.50	0.32	0.13	0.78	0.76	0.68	0.34	
P2	0.25	0.05	0.80	0.75	0.12	0.44	0.19	0.08	0.79	0.74	0.47	0.51	-	-	-	-	-	-	-
P3	0.17	0.03	0.79	0.75	0.08	0.38	0.15	0.08	0.78	0.74	0.47	0.49	-	-	-	-	-	-	-
P4	0.23	0.07	0.81	0.74	0.08	0.42	0.25	0.10	0.77	0.71	0.49	0.54	0.32	0.13	0.80	0.78	0.67	0.30	
P5	0.27	0.05	0.82	0.75	0.08	0.45	0.17	0.10	0.80	0.73	0.57	0.48	0.32	0.12	0.77	0.76	0.62	0.28	
P6	0.25	0.07	0.80	0.73	0.12	0.38	0.28	0.10	0.79	0.73	0.52	0.45	0.37	0.17	0.47	0.46	0.28	0.08	
P7	0.10	0.02	0.73	0.76	0.05	0.30	0.17	0.07	0.74	0.76	0.35	0.45	0.25	0.12	0.73	0.73	0.63	0.30	
P8	0.45	0.02	0.65	0.66	0.00	0.03	0.50	0.00	0.60	0.59	0.02	0.05	0.47	0.07	0.62	0.61	0.15	0.03	

Table 10: Compilation, testing, and reasoning score results for different LLM configurations for synthetic vulnerabilities in LogicDS. Column meanings are the same as in Table 1. “P” represents the prompt IDs. Suffixes “L”, “Q”, and “O” after each “CS” and “J” represent judging LLM Llama, Qwen, and OpenAI-o3-mini, respectively.

	Llama-3.1					Qwen-2.5					OpenAI-o3-mini								
	P	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P9	0.23	0.05	0.81	0.75	0.10	0.38	0.20	0.07	0.81	0.75	0.50	0.55	0.30	0.12	0.78	0.76	0.63	0.27	
P10	0.40	0.07	0.76	0.75	0.08	0.35	0.40	0.17	0.78	0.73	0.45	0.52	0.48	0.22	0.76	0.74	0.72	0.30	
P11	0.23	0.08	0.80	0.76	0.07	0.42	0.30	0.13	0.80	0.75	0.48	0.53	0.43	0.18	0.76	0.75	0.72	0.30	

Table 11: Compilation, testing, and reasoning score results for varying portions of the source code provided for synthetic vulnerabilities in LogicDS. Column meanings are the same as in Table 1 and 2.

	Llama-3.1					Qwen-2.5					OpenAI-o3-mini								
	P	C	T	CSQ	CSO	JQ	JO	C	T	CSL	CSO	JL	JO	C	T	CSL	CSQ	JL	JQ
P12	0.48	0.02	0.65	0.62	0.00	0.03	0.47	0.00	0.59	0.57	0.05	0.03	0.42	0.05	0.61	0.60	0.13	0.00	
P13	0.22	0.07	0.81	0.75	0.08	0.37	0.18	0.05	0.81	0.75	0.48	0.48	0.32	0.12	0.77	0.75	0.63	0.25	
P14	0.12	0.05	0.77	0.78	0.17	0.52	0.13	0.05	0.80	0.76	0.68	0.65	0.17	0.07	0.78	0.76	0.75	0.30	
P15	0.43	0.33	0.79	0.82	0.72	0.97	0.58	0.53	0.80	0.78	0.97	0.98	0.38	0.32	0.77	0.75	0.95	0.73	
P16	0.42	0.28	0.77	0.82	0.79	0.93	0.53	0.49	0.77	0.79	0.93	1.00	0.33	0.26	0.75	0.73	0.95	0.63	

Table 12: Compilation, testing, and reasoning scores for varying auxiliary information provided in prompts for synthetic vulnerabilities in LogicDS. Column meanings are the same as in Table 1 and 2.

G Statistical Significance Analysis of Primary Claims

To evaluate whether the observed claims in our Experiments Section (§6) are statistically significant, we performed paired significance tests for each claim. For binary outcomes, compilation success/failure (**C**), test-case success/failure (**T**), and the binary LLM-as-a-judge verdict (**J**): we used an exact McNemar’s test and report the discordant counts, p-value, and odds ratio (OR). For continuous outcomes, such as the LLM-as-a-judge cosine similarity score (CS) we used the Wilcoxon signed-rank test and report the p-value, Cliff’s δ (effect size), the mean paired difference, and the bootstrap confidence interval (CI).

Our primary claims are as follows:

1. Adjusting temperature has minimal impact on repair performance (i.e., compilation, testing and reasoning scores for output patches from prompts **P1–P3** are identical). We test this by comparing **P1** vs. **P2**.
2. Orientation has minimal impact (i.e., output patches from prompt templates **P1** and **P4** yield similar results).
3. The zero-shot prompt template (**P5**) achieves higher compilation success and slightly better reasoning than the CoT prompt template (**P7**).
4. Including reasoning text in the patch-generation prompt improves patch quality.

ID	Claim	C (McNemar)	CS (Wilcoxon p, Cliff δ , Mean Diff., CI)	J (McNemar)
1	P1 and P2 show similar results	86/219 vs 85/219 ($p=1.0000$, OR=1.14)	0.7801 vs 0.7740; $p=0.1516$ (<i>ns</i>); $\delta=+0.080$; $\Delta=+0.0061$ [-0.0018, +0.0140]	169/438 vs 175/438 ($p=0.5446$, OR=0.84)
2	P1 and P4 show similar results	136/332 vs 141/332 ($p=0.5114$, OR=0.76)	0.7753 vs 0.7696; $p=0.1634$ (<i>ns</i>); $\delta=+0.062$; $\Delta=+0.0057$ [-0.0028, +0.0141]	289/664 vs 283/664 ($p=0.6137$, OR=1.13)
3	P5 shows better C , CS , J than P7	139/339 vs 114/339 ($p=0.0015$, OR=2.47)	0.7757 vs 0.7420; $p \leq 0.001$ (<i>sig</i>); $\delta=+0.537$; $\Delta=+0.0337$ [+0.0260, +0.0413]	300/726 vs 285/726 ($p=0.2513$, OR=1.22)
4	P8 shows more C but less CS , J than P7	196/339 vs 114/339 ($p \leq 0.001$, OR=5.56)	0.6344 vs 0.7420; $p \leq 0.001$ (<i>sig</i>); $\delta=-$ 0.884; $\Delta=-0.1076$ [-0.1218, -0.0939]	54/726 vs 285/726 ($p \leq 0.001$, OR=0.08)
5a	P12 has higher C but lower CS , J than P13	189/339 vs 138/339 ($p \leq 0.001$, OR=3.12)	0.6206 vs 0.7731; $p \leq 0.001$ (<i>sig</i>); $\delta=-$ 0.967; $\Delta=-0.1525$ [-0.1662, -0.1389]	53/726 vs 301/726 ($p \leq 0.001$, OR=0.04)
5b	P12 has higher C but lower CS , J than P14	189/339 vs 107/339 ($p \leq 0.001$, OR=6.47)	0.6206 vs 0.7769; $p \leq 0.001$ (<i>sig</i>); $\delta=-$ 0.950; $\Delta=-0.1564$ [-0.1715, -0.1411]	53/726 vs 375/726 ($p \leq 0.001$, OR=0.04)
6	P11 has slightly better C than P9	153/339 vs 138/339 ($p=0.0444$, OR=1.88)	0.7758 vs 0.7789; $p=0.6332$ (<i>ns</i>); $\delta=+0.058$; $\Delta=-0.0031$ [-0.0093, +0.0032]	309/726 vs 301/726 ($p=0.5264$, OR=1.14)
7	P9 has lower C but higher CS , J than P10	136/333 vs 174/333 ($p \leq 0.001$, OR=0.36)	0.7784 vs 0.7579; $p \leq 0.001$ (<i>sig</i>); $\delta=+0.345$; $\Delta=+0.0205$ [+0.0127, +0.0284]	296/714 vs 289/714 ($p=0.6110$, OR=1.11)
8a	P17 has lower C , CS , J than P5	85/339 vs 139/339 ($p \leq$ 0.001, OR=0.31)	0.5683 vs 0.7757; $p \leq 0.001$ (<i>sig</i>); $\delta=-$ 0.967; $\Delta=-0.2074$ [-0.2250, -0.1900]	21/726 vs 300/726 ($p \leq 0.001$, OR=0.04)
8b	P20 has lower C , CS , J than P5	70/339 vs 139/339 ($p \leq$ 0.001, OR=0.17)	0.6656 vs 0.7757; $p \leq 0.001$ (<i>sig</i>); $\delta=-$ 0.934; $\Delta=-0.1100$ [-0.1233, -0.0974]	143/726 vs 300/726 ($p \leq 0.001$, OR=0.27)

Table 13: Paired significance test results for the primary claims, aggregated across all evaluated LLMs for each prompt type.

P	Llama-3.1				Qwen-2.5				OpenAI-o3-mini									
	C	T	CS	CSO	JQ	JO	C	T	CS	CSO	JL	JO	C	T	CS	CSQ	JL	JQ
P17	0.18	0.00	0.58	0.55	0.00	0.02	0.00	0.00	0.59	0.57	0.05	0.07	0.18	0.02	0.58	0.57	0.02	0.00
P18	0.10	0.00	0.62	0.61	0.00	0.05	0.00	0.00	0.60	0.57	0.03	0.05	0.08	0.00	0.59	0.58	0.07	0.03
P19	0.18	0.00	0.60	0.61	0.00	0.03	0.00	0.00	0.62	0.61	0.07	0.08	0.10	0.02	0.60	0.59	0.08	0.02
P20	0.10	0.00	0.69	0.66	0.07	0.25	0.02	0.00	0.65	0.62	0.18	0.25	0.08	0.02	0.68	0.68	0.38	0.13
P21	0.07	0.02	0.77	0.74	0.18	0.42	0.00	0.00	0.74	0.70	0.60	0.57	0.07	0.00	0.76	0.75	0.73	0.23

Table 14: Compilation, testing, and reasoning score results using templates provided in (Pearce et al., 2023) for synthetic vulnerabilities in Logi cDS. Column means are the same as in Table 1 and 2.

Specifically, **P8** increases compilation success but lowers reasoning scores relative to **P7**, which includes vulnerability text.

- Auxiliary information is essential for generating reasonable patches for logical vulnerabilities. Specifically, **P12** (no auxiliary info) performs worse than **P13** (vulnerability text) and **P14** (specification info), even though **P12** produces patches with a higher compilation success rate.
- Adding context (**P11**) to a vulnerable source-code block (**P9**) improves compilation, while reasoning scores remain similar.
- Providing only the vulnerable block (**P9**) yields more compilable patches and slightly

better reasoning than providing the entire function (**P10**).

- Prompts from Pearce et al. (**P17–P20**) are less effective than our baseline prompt configuration (**P5**). We demonstrate this by comparing **P17** vs. **P5** and **P20** vs. **P5**.

For each claim, we report results aggregated across all evaluated LLMs for each prompt type. The statistical significance results are provided in Table 13. The significance tests largely support all of our claims, with two minor exceptions. For the binary LLM-as-a-judge reasoning metric (**J**), the differences are not statistically significant for the **P5** vs. **P7** comparison in claim 3 and for the **P9** vs. **P10** comparison in claim 6. In both cases, however, the continuous reasoning metric (**CS**) remains statistically significant. In essence, the relative performance of **P9** against **P10** depends on the requirements for vulnerability repair. Vulnerabilities that require broader, function-level context benefit from being provided in **P10**, whereas issues that are localized to a specific region are often better addressed with a block-level snippet in **P9**, which can facilitate repair while avoiding incorrect localization.

H Example Codes Supporting Manual Inspection of Patches

Table 15: Llama Fix for CVE-2015-1793 (cve, 2015)

<p>Before processing the certificate chain, we need to check if the certificate has a Basic Constraints extension and if the cA flag is set. We can use the 'X509_get_extension_flags' function to check if the extension is critical and the 'X509v3_get_ext_d2i' function to parse the extension.</p>
<p>Suggested Fix</p> <pre> 1 // Check the Basic Constraints extension 2 X509_EXTENSION *ext = X509_get_ext(xtmp, ↳ NID_basic_constraints); 3 if (ext != NULL) { 4 // Check if the extension is critical 5 if (X509_get_extension_flags(ext) & ↳ EXFLAG_CRITICAL){ 6 ... 7 if (bc != NULL) { 8 ... 9 BASIC_CONSTRAINTS_free(bc); 10 ... </pre>

Based on the prompt (highlighted in yellow), 2 undefined variables (highlighted in red) are generated.

Table 16: Qwen Fix for CVE-2017-3142 (cve, 2017)

<p>Suggested Fix</p> <pre> 1 if (msg->tsigstatus != dns_rcode_noerror) { 2 } else if (tsig.error != dns_rcode_noerror) { 3 } else { 4 // Extra check to ensure the requestor is ↳ authorized 5 if (!is_requestor_authorized(msg)) { 6 result = DNS_R_UNAUTHORIZED; 7 ... </pre>
<p>Ground Truth Fix</p> <pre> 1 if (msg->verified_sig && msg->tsigstatus == ↳ dns_rcode_noerror && tsig.error == ↳ dns_rcode_noerror) { 2 } else if ((!msg->verified_sig) (msg-> ↳ tsigstatus != dns_rcode_noerror)) { 3 } else { 4 INSTANT(tsig.error != dns_rcode_noerror); 5 result = DNS_R_TSIGERRORSET; 6 } </pre>

Undefined variables (highlighted in red) were created due to the lack of context.

I Additional Insights

Error Analysis of Incorrect Patches. To systematize our error analysis and gain additional insight, we randomly sampled 200 patch instances deemed not plausible (i.e., cases where compilation failed, tests failed, or tests were unavailable) from our off-the-shelf LLM patch-generation experiments. We then manually reviewed each instance to identify the primary reason for failure. From this assessment, we derived four main failure categories: (i) invalid variable/function hallucination, where the patch introduces non-existent variables or function names, usually imagined from the prompt text (typically leading to compilation failures); (ii) incorrect logic, where the patch applies a fundamentally different approach than what is required

Table 17: Llama Fix for CVE-2015-1793 (cve, 2015)

<p>The whole function was input in the prompt, but the LLM failed to locate the vulnerable portion and add extra code after the vulnerable portion instead. The original vulnerable code has 319 lines, and the suggested patch has 336 lines.</p>
<p>Suggested Fix</p> <pre> 1 if (ok > 0) { 2 X509_free(xtmp); 3 while (num > j) { 4 xtmp = sk_X509_pop(ctx->chain); 5 X509_free(xtmp); 6 num--; 7 ctx->last_untrusted--; 8 } 9 retry = 1; 10 break; 11 } 12 ... 13 end: 14 ... 15 // Fix: Check for Basic Constraints cA value 16 if (ctx->chain != NULL) { 17 for (i = 0; i < sk_X509_num(ctx->chain); i++) { 18 x = sk_X509_value(ctx->chain, i); 19 if (X509_check_ca(x) != X509_CHECK_CA_OK) { 20 ctx->error = X509_V_ERR_INVALID_CA; 21 ctx->current_cert = x; 22 ctx->error_depth = i; 23 ok = cb(0, ctx); 24 if (!ok) 25 goto end; 26 ... </pre>
<p>Ground Truth Fix</p> <pre> 1 if (ok > 0) { 2 X509_free(xtmp); 3 while (num > j) { 4 xtmp = sk_X509_pop(ctx->chain); 5 X509_free(xtmp); 6 num--; 7 } 8 ctx->last_untrusted = sk_X509_num(ctx->chain); 9 retry = 1; 10 break; 11 } </pre>

to fix the vulnerability; (iii) partially correct/incomplete logic, where the patch captures some elements of the intended fix but omits key steps or includes incorrect reasoning; and (iv) presumably correct logic, where the patch appears conceptually correct but cannot be deemed plausible because no test cases are available for validation. The distribution across the 200 assessed patches is shown in Table 24.

Of the 200 assessed patches, 112 patches compile successfully. We also observed two qualitative patterns. First, smaller patches leave less room for *partial correctness* missing a single required step often results in a patch being categorized as incorrect logic rather than partially correct. Second, for a given vulnerability instance, multiple patch suggestions often exhibit the same failure mode.

Table 18: OpenAI Fix for CVE-2022-1434 (cve, 2022)

<p>The fix adds a check after calling <code>OSSL_PARAM_locate_const</code> to verify that a valid pointer was returned rather than a NULL reference. Without this check, subsequent code that uses 'p' could try to dereference a NULL pointer if <code>OSSL_PARAM_locate_const</code> doesn't find the expected parameter, potentially leading to undefined behavior, a crash, or even exploitable security vulnerabilities.</p>
<p>Suggested Fix</p> <pre> 1 p = OSSL_PARAM_locate_const(params, ↪ OSSL_CIPHER_PARAM_AEAD_TLS1_AAD); 2 if (p == NULL) { 3 /* The parameter is required; if not found, ↪ report an error and bail out */ 4 ERR_raise(ERR_LIB_CIPHER, ↪ CIPHER_R_MISSING_AEAD_TLS1_AAD); 5 return 0; 6 } </pre>
<p>Ground Truth Fix</p> <pre> 1 p = OSSL_PARAM_locate_const(params, ↪ OSSL_CIPHER_PARAM_AEAD_MAC_KEY); 2 ... </pre>

Because no detailed vulnerability was specified in the prompt, the LLM assumed a NULL-pointer dereference risk when calling `OSSL_PARAM_locate_const`. Therefore, it adds a check immediately after this call to prevent any subsequent dereference of a NULL pointer.

Table 19: Qwen Fix for CVE-2015-1793 (cve, 2015)

<p>Certainly! The provided code snippet contains a potential vulnerability related to memory management, specifically the handling of pointers after freeing memory. This can lead to undefined behavior, including crashes or security vulnerabilities such as use-after-free errors. Let's break down the original code and then explain the repairs made. ...</p>
<p>Suggested Fix</p> <pre> 1 if (ok > 0) { 2 if (xtmp != NULL) { 3 X509_free(xtmp); 4 xtmp = NULL; // Avoid dangling pointer 5 } 6 ... 7 } </pre>
<p>Ground Truth Fix</p> <pre> 1 if (ok > 0) { 2 X509_free(xtmp); 3 ... 4 } </pre>

Because no detailed vulnerability was specified in the prompt, the LLM assumed that a use-after-free risk existed in the code and generated a patch to prevent any subsequent dereference of the dangling pointer.

Table 20: OpenAI Fix for CVE-2023-48795 (cve, 2023b)

<p>... The guess is considered wrong if the kex algorithm and/or the host key algorithm are guessed wrong (server and client have different preferred algorithms) or if any of the other algorithms cannot be agreed upon. Otherwise, the guess is considered to be right, and the optimistically sent packet MUST be handled as the first key exchange packet. However, if the guess was wrong and a packet was optimistically sent by one or both parties, such packets MUST be ignored, and the appropriate side MUST send the correct initial packet. ...</p>
<p>Suggested Fix</p> <pre> 1 if (first_kex_packet_follows session-> ↪ send_first_kex_follows) { 2 ... 3 if (session->first_kex_follows_guess_wrong) { 4 discard_first_kex_packet(session); 5 } else { 6 // Process the optimistically sent packet ↪ normally 7 process_first_kex_packet(session); 8 } </pre>

Provided the specification text in the prompt. However, since the code and the specification are at different abstraction levels, the LLM created functions that do not exist in the codebase, resulting in a patch that cannot pass compilation.

Table 21: OpenAI Fix for CVE-2019-1543 (cve, 2019g)

<p>Suggested Fix</p> <pre> 1 ... 2 if (actx->aad != NULL actx->len.aad != 0 3 actx->len.text != 0) { 4 EVPerr(EVP_F_CHACHA20_POLY1305_CTRL, 5 EVP_R_COPY_ERROR); 6 return 0; 7 } 8 { 9 EVP_CIPHER_CTX *dst = (EVP_CIPHER_CTX *)ptr; 10 dst->cipher_data = OPENSSL_memdup(11 actx, sizeof(*actx) + Poly1305_ctx_size()) 12 ↪ ; 13 if (dst->cipher_data == NULL) { 14 EVPerr(EVP_F_CHACHA20_POLY1305_CTRL, 15 EVP_R_COPY_ERROR); 16 return 0; 17 } 18 } 19 return 1; </pre>
--

The generated patch has different logic from the ground-truth patch. The ground-truth is provided in 1.

Table 22: KNOD (Jiang et al., 2023) Fix for CVE-2019-1543 (cve, 2019g) (Synthetic)

<p>Suggested Fix</p> <pre> 1 ... 2 case EVP_CTRL_AEAD_SET_IVLEN: 3 if (actx == null){ 4 return 0; 5 } 6 if (arg <= 0 arg > CHACHA_CTR_SIZE) { 7 return 0; 8 } 9 actx.nonce_len = arg; 10 return 1; 11 ... </pre>

The generated patch has different logic from the ground-truth patch. The ground-truth for the real-world example is provided in 1.

Table 23: SimFix (Jiang et al., 2018) Fix for CVE-2019-1543 (cve, 2019g) (Synthetic)

<p>Suggested Fix</p> <pre> 1 ... 2 case EVP_CTRL_AEAD_SET_IVLEN: 3 if (arg <= 0 arg > CHACHA_CTR_SIZE) { 4 if (arg <= 0 rhType > CHACHA_CTR_SIZE) { 5 return 0; 6 } 7 return 0; 8 } 9 actx.nonce_len = arg; 10 return 1; 11 ... </pre>
--

The generated patch has different logic from the ground-truth patch. The ground-truth for the real-world example is provided in 1.

Root Cause Type	Count
Incorrect Logic	107
Partially correct/Incomplete Logic	57
Invalid Variable/Function Hallucination	23
Presumably Correct Logic	13

Table 24: Root Cause Analysis of Incorrect Patches for 200 sampled patch instances.