PatchScribe: Theory-Guided Vulnerability Repair with Dual Causal Explanations

Anonymous Authors

***Abstract*—Large Language Models (LLMs) can generate code patches for security vulnerabilities, but their fixes often come with post-hoc explanations that are not formally verified. This paper addresses the problem of unverifiable patch rationales in LLM-driven vulnerability repair. We identify critical limitations of prior approaches: reliance on exploit-only validation, informal non-checkable explanations, and post-hoc rationalization that may not reflect the true root cause. To overcome these issues, we propose PatchScribe, a novel framework that produces formally verified dual causal explanations for vulnerability patches. Unlike prior approaches that generate patches first and explain them later, PatchScribe follows a principled theory-guided approach: it first formalizes the vulnerability causally (generating a formal bug explanation E_bug), uses this formalization to guide LLM-based patch generation, and then verifies consistency between the bug's root cause and the patch's intervention (via a formal patch explanation E_patch). PatchScribe centers on a Program Causal Graph (PCG) and an associated Structural Causal Model (SCM) to capture causal relationships. Critically, the formal bug specification is generated before patching, providing precise guidance to the LLM (e.g., "$V_{\text{overflow}} \Leftrightarrow (len > 256) \wedge (\neg Check)$; to fix, ensure Check = true or $len \leq 256$") rather than vague hints. After patch generation, we perform triple verification: (1) consistency checking to ensure E_patch addresses causes identified in E_bug, (2) symbolic verification to prove the vulnerability is unreachable, and (3) completeness checking to ensure all causal paths are disrupted. We outline the three-phase design of PatchScribe (Vulnerability Formalization, Theory-Guided Patch Generation, Dual Verification) and present an evaluation plan on recent vulnerability repair benchmarks. Our approach transforms patch verification from post-hoc testing to theory-guided generation with formal guarantees, significantly advancing the trustworthiness of automated vulnerability remediation by ensuring both that patches work and why they work.**

1. Introduction

LLM-based code assistants have shown promise in automatically fixing vulnerable code, but a critical gap remains: can we trust that the LLM's patch truly eliminates the vulnerability, and can we verify the reasoning behind it? In current practice, an LLM might propose a code change along with a natural language explanation of the fix. However, such explanations are post-hoc and often not verifiable by any rigorous means – they could be incomplete, incorrect, or even hallucinated. This lack of verifiability in patch rationales poses a security risk: a patch that "sounds" correct might still fail to eliminate the underlying exploit path or might introduce new issues, all while the developer is misled by a plausible but unproven explanation.

Recent research underscores the limitations of relying on informal validation of patches. Studies have found that LLM-generated fixes can be plausible yet incorrect, passing unit tests or superficial checks without truly removing the vulnerability. For example, incomplete patches in real-world libraries (PyYAML, Pillow, Django) passed initial review but left residual flaws that attackers later exploited. To counter this, efforts like VulnRepairEval advocate for exploit-based validation, judging a patch by whether a proof-of-concept (PoC) exploit is blocked. This is a step toward realism – requiring that the original attack no longer succeeds – and has revealed that state-of-the-art LLMs fix only ˜22% of known CVEs under these strict conditions. However, even exploit-only validation is limited: it confirms that one particular attack input is mitigated, but it does not prove in general that the vulnerability is fully eradicated or that no new vulnerabilities are introduced by the patch. Moreover, it provides no insight why the patch works (if it does). Developers and security auditors are left to trust the LLM's textual rationale, which may be unfaithful to the code's actual logic.

We argue that a more principled approach is needed – one that combines the generative capabilities of LLMs with formal reasoning about causal relationships in the code. Our key insight is twofold: (1) formalize the vulnerability before attempting to patch it, using the formalization to guide (not just verify) patch generation, and (2) generate separate formal explanations for the bug and the patch, enabling consistency checking between them. If we can formally capture what conditions cause the vulnerability (E_bug), we can provide the LLM with precise guidance (e.g., "V_overflow occurs when len $>$ 256 AND no check exists; ensure one of these is false"). After the LLM generates a patch, we formalize how the patch intervenes (E_patch) and verify that the intervention actually addresses the causes identified in E_bug. This dual-explanation approach catches incomplete fixes that might pass exploit tests but miss edge cases.

In this paper, we introduce PatchScribe, a framework that brings theory-guided causal reasoning to LLM-based vulnerability repair. Unlike prior approaches that explain patches post-hoc, PatchScribe follows a pre-hoc methodology: formalize first, patch second, verify consistency third. The approach builds a Program Causal Graph (PCG) to represent the causal structure of the vulnerability (for instance, how a lack of input validation leads to a buffer overflow) and

instantiates a Structural Causal Model (SCM) on top of this graph. From the SCM, we generate a formal bug explanation (E_bug) that precisely characterizes the vulnerability condition, causal paths, and intervention options. This formal specification is provided to the LLM as guidance, enabling it to generate more targeted patches. After patch generation, we analyze how the patch intervenes on the causal model and generate a formal patch explanation (E_patch) describing what changed and why the vulnerability is eliminated. We then perform triple verification: (1) consistency checking between E_bug and E_patch (does the patch address identified causes?), (2) symbolic verification (is the vulnerability provably unreachable?), and (3) completeness checking (are all causal paths disrupted?). This three-phase approach (Formalization $\rightarrow$ Theory-Guided Generation $\rightarrow$ Dual Verification) provides stronger guarantees than prior work, catching cases where patches superficially appear correct but fail to address the root cause.

We hypothesize that PatchScribe will significantly improve trust in automated patches. By verifying that a patch actually removes the conditions that caused a vulnerability, we prevent scenarios where an LLM patch "fixes" a symptom but not the underlying problem. Our approach directly addresses the limitations of prior work: (1) It goes beyond exploit-specific testing by formally proving the absence of the vulnerable behavior under broad conditions, and (2) it replaces unverifiable, natural-language rationales with formal, checkable explanations. This yields a double benefit: higher assurance of security and clearer insight into patch correctness.

We structure the rest of the paper as follows. In Background and Motivation, we survey LLM-based vulnerability repair efforts and highlight the need for verifiable explanations. Section Threat Model defines the assumed capabilities of attackers and the requirements for patch verification in our context. We then discuss Limitations of Existing Work, examining why current techniques (including exploit testing and LLM reasoning prompts) fall short of full verification. In Design Requirements, we derive principles to guide our solution. Section Proposed Approach (PatchScribe) provides an overview of our framework. We formalize the core of PatchScribe in Formal Model (PCG and SCM), presenting how we model programs and vulnerabilities causally. Next, Explanation Generation and Checking details how we produce and verify the causal patch explanations. An Evaluation Plan outlines how we will empirically assess patch correctness and explanation validity. We then give an Implementation Summary of our prototype. Related Work compares our approach with recent advances in vulnerability repair from 2023–2025. We discuss Limitations and Threats to Validity of our approach and finally conclude with future directions in Conclusion.

2. Background and Motivation

Automated vulnerability repair has long been a goal of the security community. Traditional program repair techniques (e.g., GenProg, SPR) targeted general bugs with test-suite specifications, but security vulnerabilities pose special challenges – a patch must not only pass functional tests but also prevent exploits and avoid weakening security in other ways. With the rise of powerful code-focused LLMs (such as GPT-4, Code Llama, etc.), researchers have explored using these models to generate vulnerability fixes from code context and problem descriptions. The appeal is clear: an LLM trained on vast code corpora may suggest creative fixes even for complex flaws, potentially reducing the window of exposure.

Early results are encouraging yet cautionary. LLM-generated patches can often superficially appear correct – they may compile and even satisfy basic tests – while still failing to eliminate the security issue. For example, if a vulnerability arises from a missing input validation, an LLM might add a check that covers the provided example but not all cases, or place the fix in an incorrect location. The validation gap in many evaluations has led to overly optimistic conclusions about LLM capabilities. To address this, the community has shifted towards more rigorous evaluation. Wang et al.'s VulnRepairEval benchmark (2025) explicitly uses real proof-of-concept exploits as tests: a generated patch is only deemed successful if it stops the exploit from working. Their study revealed a significant performance drop compared to earlier metrics – the best model patched only ˜5 out of 23 Python CVEs ($\approx 22\%$) when judged by exploit prevention. This indicates that many LLM "fixes" did not truly remove the vulnerable condition, underscoring the need for deeper verification.

Beyond generation, researchers have begun to incorporate reasoning and feedback to guide LLMs in repair tasks. Chain-of-thought (CoT) prompting is one such technique: by asking the LLM to reason step-by-step about the vulnerability and potential fix, we can reduce logical errors and improve patch correctness. Kulsum et al. (2024) showed in their VRpilot system that using a CoT prompt plus iterative validation (compiling and testing each candidate patch) improved patch success rates by 14% for C vulnerabilities compared to LLM baselines. Similarly, advanced prompting strategies like Tree-of-Thought and self-consistency have been applied. For instance, SAN2PATCH (Kim et al., 2025) splits the repair process into stages (understanding the bug, locating it, planning a fix, generating code) and uses a Tree-of-Thought prompt at each stage to systematically explore reasoning paths. This approach, leveraging AddressSanitizer crash logs for guidance, achieved markedly higher patch success rates (63–79%) on benchmark datasets. These works illustrate that richer reasoning can help LLMs avoid some pitfalls of naive generation.

However, all these approaches still rely on the LLM's internal reasoning or external tests to judge correctness. Whether it's VRpilot's chain-of-thought or SAN2PATCH's guided stages, the rationale for the patch is ultimately encapsulated in either natural language explanations or the passing of certain tests. Neither provides a formal guarantee of security. A chain-of-thought explanation is essentially an LLM's opinion of why the patch works, which might not be logically sound or complete. Even if the exploit used

in testing is thwarted, one cannot be sure that a slightly different attack wouldn't succeed, or that the patch didn't open a new vulnerability elsewhere. In short, today's LLM-based repair techniques leave us with a lingering question: did we really fix the root cause, and how can we be sure?

This motivates a new angle: introducing formal, causal reasoning into the vulnerability repair loop. Our motivation is influenced by principles of explainable AI and classic formal methods. In explainable AI, a distinction is made between interpretations (post-hoc, model-generated justifications) and explanations grounded in the true causal factors of a decision. Current LLM patch explanations are interpretations – fluent narratives that may or may not align with the program's actual logic. We seek to replace these with causal explanations that reflect genuine cause-effect relations in code. Meanwhile, formal verification in security has proven that machine-checked proofs can provide strong guarantees (e.g., proof-carrying code, verified compilers), but writing full specifications or proofs for arbitrary software is notoriously difficult. Our approach threads a middle ground: we do not require a full formal specification of program behavior, only a formal characterization of the vulnerability condition and its causes. This makes the problem more tractable while still yielding actionable proofs about the patch's effect.

In summary, PatchScribe is motivated by the need to bridge LLM-driven flexibility with formal assurance. By constructing a causal model of the vulnerability, we aim to ensure that any patch – whether generated by an LLM or otherwise – actually addresses the core vulnerability cause. The machine-checkable explanation serves as both a certificate of patch correctness and a human-readable justification. This increases confidence for developers deploying LLM-generated fixes in security-critical code, moving us closer to trustworthy automated vulnerability repair.

3. Threat Model

We consider a scenario where an attacker is attempting to exploit a known vulnerability in a software system, and an automated repair system (powered by an LLM and our verification framework) is used to generate and validate a patch. The assets to protect are the integrity and availability of the software's functionality, as well as any sensitive data it handles, which could be compromised by the vulnerability. The attacker's capability is that they can provide arbitrary inputs to the software (including the original exploit input and variants) in an attempt to trigger the vulnerability.

Assumptions: We assume the location or indicator of the vulnerability is known to the repair system (e.g., through a CVE description, a stack trace, or a sanitizer log pinpointing the issue). This is consistent with many vulnerability repair scenarios where the bug is first discovered and needs patching. We also assume the LLM and tools used (compiler, symbolic executor, etc.) are trusted and not maliciously manipulated – the threat here is not the toolchain but the risk of a wrong or incomplete patch. The patch generation process itself is not adversarial: the LLM might be prone to errors, but we do not model it as actively trying to introduce malicious code.

Attacker Model: The attacker's goal is to exploit the vulnerability to achieve some impact (e.g., buffer overflow leading to code execution, information leak, denial of service). Post-patch, the attacker will try any inputs or strategies available to bypass the fix. In particular, if the patch only blocks a specific input pattern (the original exploit), the attacker might try a variant input that still triggers the vulnerability via a slightly different path. The attacker may also scour the patched code for newly introduced weaknesses (for instance, if the patch adds code that itself has a flaw).

Defender/Repair Goals: The automated repair system's goal is to produce a patch that eliminates the vulnerability – meaning the exploitable condition can no longer occur – while preserving the software's intended functionality and not introducing new vulnerabilities. In formal terms, if we describe the vulnerability by a condition (such as an assertion failure, crash, or unsafe state) reachable in the original program, the patched program should prevent that condition for all feasible inputs (not just the known exploit). The patch should also respect a safety property of not opening another known class of vulnerability (like not introducing an obvious new overflow or bypass). Functional correctness (the patch doesn't break legitimate features) is important but in this work we primarily focus on security correctness; we assume basic regression testing is done to catch functionality issues.

Threats Addressed by PatchScribe: Our approach is designed to mitigate two main threat scenarios: 1. Incomplete Fix Threat: The patch does not fully close off the vulnerability's cause. The attacker finds an alternative input or path that still satisfies the conditions for exploitation. PatchScribe counters this by formally verifying that the causal chain leading to the vulnerable condition is broken under all relevant circumstances, not just the originally observed exploit. If any path remains, the explanation check fails and the patch is rejected or flagged as insufficient. 2. Misleading Explanation Threat: The LLM (or developer) provides an explanation claiming the vulnerability is fixed (for example, "We added a check to ensure the index is within bounds"), but this rationale might be false or only partially true. Without formal verification, a developer might believe the threat is gone. In our threat model, a misleading explanation is dangerous because it could cause premature confidence and deployment of a flawed patch. PatchScribe addresses this by requiring the explanation to be machine-checkable. If the LLM's explanation does not hold – e.g., if it claims a check prevents out-of-bounds access but symbolically an out-of-bounds write is still possible – the discrepancy will be caught by the explanation checker.

We do not specifically model an attacker who can manipulate the LLM itself (e.g., prompt injection into the repair process) – that is orthogonal and can be mitigated by securing the input channels to the LLM. We also acknowledge that if the vulnerability or its exploit is extremely complex (e.g., requiring multi-step logical conditions or

specific environment states), the PCG/SCM we build might simplify or omit some factors, potentially leaving residual risk. These aspects will be discussed in Limitations.

In summary, the threat model focuses on an honest-but-fallible repair system versus a determined attacker exploiting any weakness left after patching. The role of PatchScribe is to strengthen the defender's side by adding a rigorous verification step that reduces the chance of an attacker slipping through an unaddressed causal pathway or a patch error. The security guarantee we seek is: If PatchScribe approves a patch with a verified explanation, then the known vulnerability is truly eliminated (the known exploit and any variant following the same root cause are prevented) with high confidence. This guarantee is stronger than what prior LLM-only methods provide, thereby shrinking the attacker's opportunity space post-patch.

4. Limitations of Existing Work

Existing automated vulnerability repair methods, especially those leveraging LLMs, exhibit several limitations that motivate our work:

Reliance on Incomplete Validation: Many approaches validate patches using tests or specific exploit instances, but not comprehensive proofs. Early LLM-based repairs were often evaluated by running built-in test suites or simply checking that the program still executed without crashing. This test-based validation can miss security issues – a patch that passes all tests might still be vulnerable to an untested exploit. Even the more rigorous exploit-based evaluations (e.g., VulnRepairEval) are essentially exploit-only validation: they use one PoC exploit as the litmus test. If the exploit is blocked, the patch is considered successful. However, this overlooks the possibility of variant exploits or edge cases the patch doesn't handle. For instance, a patch might specifically check for a known malicious input pattern rather than fixing the underlying unsafe logic; the original exploit fails, but a tweaked input could still succeed. Thus, exploit-only validation, while reflecting real attacker behavior better than unit tests, cannot guarantee completeness of the fix.

Unverifiable Rationales: When LLMs produce patches, they can also produce explanations or reasoning traces (especially if prompted with CoT). These rationales are natural language descriptions of what the patch does and why. Crucially, there is no guarantee that these rationales are correct or complete. Studies have observed that LLMs often give confident-sounding explanations that are partially or wholly incorrect, a phenomenon known as hallucinated reasoning. In the context of security, an LLM might assert "the buffer is now bounds-checked, so the overflow is resolved," but unless we verify the code, we cannot be sure that the check covers all cases or that the logic is implemented correctly. No current LLM-based repair system provides a formal link between the explanation and the code – the explanation is essentially commentary. This is a limitation because developers cannot distinguish a truly sound patch rationale from a flawed one without investing manual effort (code review or formal analysis themselves). Unverifiable rationales contribute to a false sense of security.

Lack of Formal Guarantees: Traditional program repair research has explored formal methods (e.g., generating patches that come with proofs or using verification conditions to guide repairs [6]). However, integrating formal verification with LLM-driven patching has seen minimal exploration. One reason is that formal specifications for security properties are hard to write for arbitrary code, and fully verifying a patch can be as hard as verifying the entire program. As a result, most LLM repair systems avoid formal proofs, leaving a gap where no strong guarantees back up the patch. The limitation here is apparent: without machine-checkable proof, there's always uncertainty. Tools like PATCHVERIF have demonstrated that it's possible to use symbolic execution and invariants to check if a patch truly addresses expected behaviors [9], but such tools require significant setup and are not generally plugged into the LLM generation loop. This means current LLM patching often operates in a "generate-and-hope" mode – hope that the patch is correct, then do some testing. This is insufficient for high-assurance domains.

Narrow Reasoning Context: Another limitation in existing work is that LLMs might not fully understand the causal chain of a vulnerability. They often act on local cues. For example, if a vulnerability is "read past buffer end," an LLM might locally add a length check. But if the real root cause was a more complex sequence (say, multiple functions passing around an incorrect size), a local fix might not solve it. Approaches like SAN2PATCH and VRpilot try to broaden the context (using sanitizer logs, or iteratively refining after failed attempts). Yet, without an explicit representation of causality, there is a risk of addressing symptoms rather than causes. The limitation is the absence of a global view of how the vulnerability arises. Code property graphs and taint analyses in vulnerability detection research do create global views, but LLM-driven repair hasn't fully leveraged that. As a result, some patches are "fragile" – they only intercept the known path, not all paths.

Human Effort and Trust Issues: Finally, in practice, developers reviewing an LLM-suggested patch have limited tools to verify it beyond re-testing or code inspection. If they distrust the patch, they might rewrite it manually, negating the benefit of automation. If they overtrust it (because the explanation sounded convincing), they might deploy a faulty fix. The current state of the art doesn't offer an intermediate artifact that developers can trust – either you trust the LLM and tests or you do a full manual verification. This limits adoption of such tools in security-critical projects, where stakes are high. A limitation noted in surveys is that while LLMs can accelerate finding and fixing bugs, their overreliance without verification is dangerous. In essence, there is a missing link to turn an LLM patch from a suggestion into a confidently acceptable solution.

In summary, prior work on automated vulnerability repair has illuminated what is possible but also what is lacking: comprehensive verification and trustworthy explanations. These limitations set the stage for our approach,

which explicitly targets them by bringing formal causal analysis into the loop. In the next section, we outline the design requirements that a system like PatchScribe must meet to overcome these issues.

5. Design Requirements

Based on the shortcomings identified, we derive a set of design requirements for an LLM-based vulnerability repair system that yields high-assurance patches with verifiable explanations:

- **R1**: Causal Correctness Guarantee. The system must ensure that the patch eliminates the root cause of the vulnerability. In practice, this means if the vulnerability is characterized by a condition or event (e.g., a buffer overflow at line X when condition Y is true), the patched program should prevent that condition/event for all relevant inputs. This goes beyond passing a specific exploit test – it requires reasoning about all paths and inputs related to the vulnerability cause. Formally, the patched program should satisfy a safety property: the vulnerability condition is unreachable. Our system should be built to prove or check this property for each patch.
- **R2**: Machine-Checkable Explanation. For each patch, the system should produce an explanation that is encoded in a structured, formal or semi-formal manner such that a machine (automated tool) can verify its correctness. This is in contrast to free-form natural language. The explanation might be represented as logical assertions, traces on a graph, or annotations in code that can be checked. The key is that there is no ambiguity – the explanation corresponds to a verifiable claim about program behavior. For instance, a valid explanation might be: "The new code adds a condition if (len $>$ N) return; before the memcpy. This ensures that when input length exceeds N, the function exits and the call to memcpy (which caused overflow) is never reached." This statement can be translated to a check: prove that if len $>$ N, then the memcpy line is not executed in the patched program. The design must facilitate generating such checkable claims.
- **R3**: Integration with LLM Patch Generation. The approach should still leverage LLMs for what they are good at – understanding code context and synthesizing code – but the system must guide or validate the LLM with the formal model. This implies two sub-requirements: (a) Guidance: the system might use the causal model to prompt the LLM in a more informed way (e.g., telling it explicitly what the cause is that needs addressing). If the LLM can incorporate this, patches are more likely to hit the mark. (b) Post-check: every patch from the LLM should be fed into the explanation generator and checker, forming a loop where an LLM-suggested fix is not accepted until it passes verification. The design should enable iterative refinement: if the first patch fails R1 or R2, perhaps the system can prompt the LLM with additional information (like "the fix didn't cover scenario X") and try again.
- **R4**: Minimal False Positives/Negatives in Verification. The verification step (explanation checking) must be sound (no false claim of success if vulnerability still exists) and as complete as possible (should catch all true errors in the patch). In formal terms, if the explanation checker approves a patch, the vulnerability should truly be fixed (this is critical for trust). If it rejects a patch, ideally the patch is indeed faulty – though it's acceptable to sometimes reject a correct patch if our analysis is conservative, we would then manually or heuristically handle it. The design should favor soundness in security (better to reject a correct patch than accept a wrong one, as a wrong one in deployment is dangerous). Achieving this may require using multiple methods (symbolic execution, static analysis, even some bounded model checking) to ensure thorough coverage of the vulnerability scenario.
- **R5**: Causal Graph Coverage and Accuracy. The Program Causal Graph that underpins the system should capture the relevant program flows and conditions related to the vulnerability. This is a requirement because if the PCG misses part of the causal chain, the explanation could be incomplete, and the verification might unknowingly omit a scenario. Therefore, building the PCG likely requires combining several program analysis techniques (data flow for how tainted input reaches a sink, control flow for what conditions guard the vulnerable code, etc.). The graph should be precise enough to distinguish the key decision points. We also need the PCG to handle multiple contributing causes (e.g., a bug may require two conditions to be true, like integer overflow + unchecked length; the model should capture both). Essentially, the requirement is that the PCG/SCM must form a correct model of the vulnerability, as only then can the explanation be valid.
- **R6**: Usability and Interpretability. While being formal, the approach's output should still be human-readable or at least interpretable by a developer. This means the explanations should ideally be presented in terms of program entities (variables, functions, conditions) and not overly abstract formulas. If we produce something like "$\forall n : n < N \Rightarrow \neg overflow$", that's formal but maybe not immediately clear to a developer. We might instead say "Because the patch ensures n $<$ N before writing, an overflow cannot occur." The requirement is to maintain a connection between the formal model and intuitive understanding, so the tool can be adopted by practitioners who want assurance but aren't formal methods experts. A secondary aspect is minimizing additional annotation burden – the approach should work with just the code and perhaps a description of the vulnerability, rather than requiring the developer to write full specifications.
- **R7**: Compatibility with Real-World Tools and Workflows. The system should be designed to integrate

with typical development workflows. That implies using standard languages (the prototype might focus on C/C++ or Python vulnerabilities, given existing benchmarks), interfacing with existing compilers or analyzers, and keeping runtime of verification reasonable. If our approach took days of SMT solving to verify one patch, it'd be impractical. So performance is a consideration: the design should focus the formal checks on the vulnerability-specific parts of the program to scale. Additionally, it should be able to ingest realistic code (with libraries, etc.) by leveraging robust parsing (hence using Clang/LLVM for C, for example). Ideally, PatchScribe can be run as an automated tool in a CI/CD pipeline for security patches, meaning it should output clear pass/fail signals and reports.

- **R8**: No Regression of Functionality. Although our primary focus is security, a requirement for any patch generator is not to break the software's intended use. Therefore, our design should include at least a minimal step to check that the patch doesn't cause obvious functionality loss (for instance, if tests are available, run them to ensure they still pass). This might be outside the core causal verification loop, but as a requirement, we acknowledge that a "secure" patch that shuts down a feature entirely might not be acceptable. In practice, we might incorporate regression tests or sanity checks in the evaluation pipeline.

These requirements guided the design of PatchScribe. In the next section, we describe the overall approach and how these requirements are met by our system's architecture.

6. Proposed Approach (PatchScribe)

PatchScribe is a system for automated vulnerability repair that produces formally verified dual causal explanations for each patch. Unlike prior approaches that generate patches first and explain them post-hoc, PatchScribe follows a principled theory-guided approach: it first formalizes the vulnerability causally, uses this formalization to guide patch generation, and then verifies the consistency between the bug's root cause and the patch's intervention.

At a high level, PatchScribe operates in three phases:

Phase 1: Vulnerability Formalization - We analyze the vulnerable program to build a Program Causal Graph (PCG) and derive a Structural Causal Model (SCM). From the SCM, we generate a **formal vulnerability specification (E_bug)** that precisely characterizes the conditions under which the vulnerability manifests. This specification serves as both a verification target and guidance for patch generation. The formal bug explanation E_bug contains: (a) the formal condition characterizing when V_bug occurs, (b) natural language descriptions mapped to code, (c) intervention options for fixing the vulnerability, and (d) verification assertions.

Phase 2: Theory-Guided Patch Generation - Armed with the formal vulnerability specification E_bug, we prompt an LLM to generate a patch. Critically, the LLM receives not vague hints but a precise formal description of what conditions cause the vulnerability and what the patch must achieve. For example, instead of saying "fix the overflow," we provide "$V_{\text{overflow}} \Leftrightarrow (len > 256) \wedge (\neg Check)$; to fix, you must ensure Check = true or $len \leq 256$ before line 42." After patch generation, we analyze how the patch intervenes on the causal model and generate a **formal patch explanation (E_patch)** describing the intervention, its effect on V_bug, and which causal paths are disrupted.

Phase 3: Dual Verification - We perform three types of verification: (1) **Consistency checking** to ensure the patch explanation addresses the causes stated in the bug explanation (does E_patch actually handle what E_bug identified?), (2) **Symbolic verification** to prove the vulnerability condition is unreachable in the patched program, and (3) **Completeness checking** to ensure all identified causes are properly handled. This triple verification provides stronger guarantees than prior work.

This approach ensures that patches are not only verified to work, but that we understand and can formally prove *why* they work in terms of the causal structure of the vulnerability. The key innovation is the separation of bug and patch explanations, enabling consistency verification that catches incomplete fixes.

In meeting the design requirements: - PatchScribe yields causal correctness (R1) by virtue of the SCM reasoning and explanation check that covers all relevant cases, not just one exploit. - The explanation is machine-checkable (R2) by construction; it's effectively an intermediate formal spec of the patch's effect that we verify. - The LLM is integrated (R3) through guided prompts and iterative patch attempts informed by the formal analysis. - The verification approach prioritizes soundness (R4): we declare success only with a proof or exhaustive check for the vulnerability condition. - The PCG ensures we focus on the true causes (R5), and we plan to use strong program analysis (like control flow graph, taint tracking) to build it accurately. - For usability (R6), our explanations remain tied to code conditions and can be output in natural language form ("this new check ensures…") in addition to the formal form. - We leverage real tools (Clang, angr, etc.) ensuring we work on real code (R7). The heavy lifting by these tools (which are optimized in C/C++) helps performance. - And we include regression testing in the evaluation loop for functionality (R8), though our main unique step is the security proof.

In the following sections, we dive deeper into the formal modeling with PCG and SCM, and how exactly the explanation generation and checking work.

7. Formal Model (PCG and SCM)

In PatchScribe, the formal foundation is provided by two interrelated models: the Program Causal Graph (PCG) and the Structural Causal Model (SCM). Here we define each and explain how they are constructed and used.

7.1. Program Causal Graph (PCG)

Definition: A Program Causal Graph is a directed graph $G = (V, E)$ where each node $v \in V$ represents a program state predicate or event related to the vulnerability, and a directed edge $(u \to v) \in E$ indicates that node u has a direct causal influence on v in the context of the vulnerability.

The PCG is a high-level representation extracted from the program's code and execution flow: - Nodes: We include nodes for conditions (e.g., the truth value of an if condition), for certain variable states (e.g., "variable x has value n"), and for specific events like "function f calls g" or "memory write at location L occurs". Particularly, we distinguish a special node V_"bug" representing the occurrence of the vulnerability (e.g., an out-of-bounds write or a crash condition). We also include nodes representing the negation or absence of certain checks, since the lack of a check is often a cause (for example, a node might be "No null-pointer check before dereference" which is essentially a predicate that is true when a check is missing in the code path). - Edges: If the program logic is such that u being true or an event happening contributes to v becoming true/happening, we draw an edge $u \to v$. This is akin to saying "u is a direct cause of v" under the framework of causal graphs (similarly to Bayesian network or Pearl's causal diagrams, but here based on program logic rather than statistical data). For instance, if the code has if $len > N$ goto error; then the condition "len $> N$" (node u) causally influences whether the program goes to error handling (node v). In a vulnerability context, we might have edges like "Input not sanitized" $\to$ "Buffer overflow occurs" or "Flag is false" $\to$ "Access control bypass".

Construction Method: To build the PCG, we rely on program analysis techniques: - We start from the vulnerability point V_"bug" (e.g., the line of code that crashes or the condition that should not be true). We perform a backward static slice or taint analysis: find what inputs, variables, and conditions can lead to V_"bug" being true. This gives a dependency subgraph of the program (similar to a backward program dependence graph). - We then refine these dependencies into causal relationships. For control dependencies: if reaching V_"bug" requires passing through a branch guarded by condition C, then C (or more precisely C's truth value) is a node causing V_"bug" . For data dependencies: if a value x flows into the calculation that triggers V_"bug" (e.g., x determines the length of a copy that overflowed), then a node representing x's property (like x's value or whether x is within expected range) is included. - We also add nodes for absence-of-check scenarios. We detect patterns like use of a pointer without null-check, or copying data without a prior bounds check. These can be identified either via heuristic patterns or by comparing the program against a secure coding rule. In the PCG, an absence-of-check node is a cause for a bad event node. - The result is a graph where (hopefully) V_"bug" is at the bottom (sink) and various inputs or conditions are at the top (sources), with intermediate nodes linking them.

Example: Consider a concrete example: a C function that reads from a buffer without checking the index, causing a potential out-of-bounds read (CVE-style bug). The vulnerability event node V_"bug" = "out-of-bounds read occurs at line L". What causes this? Possibly an index i that is >= buffer_length. We add node A = "i >= buffer_length at line L". Why could A be true? Because maybe there was no check or an insufficient check. Add node B = "no check on i's value before usage". Also the value of i comes from user input, node C = "user can control i arbitrarily". We'd then have edges: C -> A (since user input can make i large), B -> A (lack of check allows i to be large), and A -> V_bug (if i is large, out-of-bounds occurs). The PCG succinctly shows: user input and missing check together cause the OOB read. In a program with multiple conditions, the graph could be more complex with converging branches etc.

7.2. Structural Causal Model (SCM)

Once we have the PCG, we formalize it as an SCM. A Structural Causal Model is typically defined by a set of endogenous variables (variables we model within the system) and exogenous variables (external inputs), along with structural equations that deterministically (or probabilistically, but here deterministically) define each endogenous variable in terms of some of the others, and a causal diagram akin to our PCG that shows dependencies.

Mapping PCG to SCM: - Each node in the PCG becomes a variable in the SCM. For boolean conditions/events, we treat them as binary variables (true/false). If certain nodes represent numeric values (like a variable's value), we could include those as numeric variables, but often we reduce to boolean predicates (like "x $>$ N" as a boolean variable). - The edges in PCG define parent-child relationships in the SCM. If $u \to v$ in PCG, then in the SCM, v will be a function of u (and possibly other parents). The SCM's structural equation for v is essentially a formal version of "v is true if ..." conditions based on its parent nodes.

Example formalization: For the above out-of-bounds example, we might define binary variables: - C (for user Control): C = true if the user can freely choose i (this might be considered exogenous input actually). - B (for check): B = true if there is a check on i (so B was false originally since no check; we use B=1 to mean check present, 0 means absent). - A (for array bounds condition): A = true if i $<$ buffer_length (safe condition) or maybe define A' = "i >= buffer_length" if following our earlier text, but let's align to safe vs unsafe. To avoid confusion, define U = true if the unsafe condition holds (i >= buffer_length when accessed). - V (vulnerability event): V = true if an out-of-bounds read occurs.

Now structural equations: - U (unsafe condition) is a function of (C, maybe and actual value of i relative to length). Actually, let's incorporate check: If B (check) is false (no check), then U = (i >= length) essentially (since nothing stops it). If B is true, presumably the code would not proceed with i out of range, so how to model: If there's a check, either the check prevents the OOB or not.

Simpler: we could model U purely as a condition on i, independent of B, and model B's effect on V. Alternatively: - V (vulnerability) depends on U and B. Without patch, B=0 always, so what made V happen is U being true and lack of check? Actually if U is true and no check, then V happens. If B is true (a check is present), presumably the code aborts or doesn't perform read, so V would be false even if U is true. So one structural equation for V could be: V := (NOT B) AND U. (Meaning an out-of-bounds occurs if the check is not present and the unsafe condition is true). - Another equation might define how B gets set or how U depends on input. If the patch introduces a check, B becomes 1; originally B=0. We treat B as a variable that can be toggled by an intervention (the patch). - U's equation: U := (i_value >= buffer_length). Now i_value itself might be an exogenous variable representing user input. So we might just treat i_value as given (exogenous). - So exogenous: i (the actual input). - Endogenous: B, U, V. - Structural eqns: * U = 1 if i >= N, else 0. * V = 1 if (B=0 AND U=1), else 0. (We might incorporate more terms if needed). * B originally (in original program) is 0 (no check). In SCM context, B might not have parents (exogenous decision by programmer). We could treat "lack of check" as an exogenous condition in original program. But in an interventional sense, setting B=1 corresponds to adding a check.

This SCM can answer counterfactual queries: What happens to V if B were 1 instead of 0? Under the structural equation, if B=1, V = (1=0 AND U=1) = 0 regardless of U. That matches our intuition: if we intervene to add a check, vulnerability V is prevented (V=0) no matter the unsafe input.

In general, the SCM would consist of variables like: $\{X_1, X_2, \ldots, X_k, C_1, \ldots, C_m, V_{\text{bug}}\}$

Where X are exogenous inputs (e.g., user-provided data, environment), C are internal conditions or flags (like B above), and V_{bug} is the bug outcome. Each C_j is defined as $f_j(\text{Parents}(C_j))$. The bug outcome has an equation $V_{\text{bug}} = f(\text{Parents}(V_{\text{bug}}))$. Typically, f for V_{bug} will have a form indicating it triggers when a certain combination of conditions holds (e.g., a logical AND of cause conditions).

Using the SCM for Patching: The SCM provides a formal way to evaluate a patch as an intervention. A patch that adds a check or alters logic is represented as setting some variable or changing some equation in the SCM: - Adding a check: changing B from 0 to 1 (an intervention do(B=1)). - Changing how a value is computed: altering the function f_j for some variable. - Removing a feature: could be like making some cause always false.

We then analyze the effect: with the intervention, is V_"bug" =0 for all relevant input ranges? This analysis is simpler than full program proof because in the SCM we abstracted away irrelevant parts. It's checking a logical condition: given the structural equations, do we have V_"bug" always false? This is akin to a small theorem to prove. Usually, it reduces to checking that some conjunction can't all be true together after the intervention.

It's worth noting that the SCM is only as good as the PCG. If the PCG missed a causal path, the SCM won't consider it and you might prove something that's only partially true. We mitigate this by thorough PCG construction (which might involve dynamic analysis as well as static to not miss feasible paths).

Formal Assurance via SCM: If our SCM-based proof says the patch stops the vulnerability, and our PCG was correct, then we have a very high confidence (almost formal proof) of security. However, to account for any mismatch between model and program, we still perform the direct code-level symbolic checks as backup. One can see the SCM proof as a guide and the code-level check as concrete validation.

In summary, the formal model PCG+SCM allows us to reason systematically about "cause and effect" in code. It provides the scaffold for generating explanations (by essentially reading off the cause variables and how the patch changes them) and for verifying patches (by checking the logical effect of the intervention). Next, we describe how we concretely generate the explanation and perform the checking using this model in tandem with program analysis.

8. Explanation Generation and Checking

This section details the three critical sub-processes of our approach: generating the formal bug explanation before patching, generating the patch explanation after patching, and verifying consistency between them.

8.1. Phase 1: Formal Bug Explanation Generation (E_bug)

Before any patch is generated, we produce a formal vulnerability specification from the PCG/SCM. This E_bug explanation contains:

1. **Formal Condition**: The logical formula characterizing when V_bug occurs
   - Example: $V_{\text{overflow}} \Leftrightarrow (len > 256) \wedge (\neg Check)$
2. **Variable Mapping**: Each variable in the formal condition mapped to code locations
   - len: computed from user_input at line 15
   - Check: bounds check (currently ABSENT before line 42)
3. **Natural Language Description**: Human-readable explanation of the vulnerability
   - "Buffer overflow occurs when input length exceeds 256 bytes AND no bounds check is performed before memcpy"
4. **Causal Paths**: All paths from inputs to V_bug
   - $user\_input \rightarrow len \rightarrow (len > 256) \rightarrow V_{\text{overflow}}$
   - absence of check $\rightarrow \neg Check \rightarrow V_{\text{overflow}}$
5. **Intervention Options**: Possible ways to fix the vulnerability
   - Option 1: Add bounds check (set Check = true)
   - Option 2: Clamp len to a safe value (ensure $len \leq 256$)
   - Option 3: Use a safe alternative (e.g., `memcpy_s`)

6. **Verification Properties**: Assertions that must hold after patching
   - Postcondition: ensure $len \leq 256$ when line 42 is reached, or confirm that line 42 is unreachable whenever $len > 256$

This formal specification is then provided to the LLM as guidance for patch generation.

8.2. Phase 2: Patch Explanation Generation (E_patch)

After the LLM generates a candidate patch, we analyze how it intervenes on the causal model:

1. **Code Changes**: Syntactic diff identifying added/modified lines
2. **Causal Intervention**: Formal representation of what the patch does
   - Example: do(len = min(len, 256)) or do(Check = true)
3. **Effect on V_bug**: How the intervention affects the vulnerability
   - Before: $V_{\text{overflow}} = (len > 256) \wedge (\neg Check)$
   - After: $V_{\text{overflow}} = (256 > 256) \wedge (\neg Check) = \text{false}$
   - Reasoning: "With len clamped to 256, $(len > 256)$ is always false"
4. **Addressed Causes**: Which causes from E_bug are handled
   - Addressed: $\{len > 256\}$
   - Unaddressed: $\{\neg\text{Check}\}$ (justified because len is now safe)
5. **Disrupted Causal Paths**: Which paths are broken
   - Path "$user\_input \rightarrow len \rightarrow (len > 256) \rightarrow V_{\text{overflow}}$" is broken because len is bounded

8.3. Phase 3: Consistency Verification

We now perform novel consistency checking between E_bug and E_patch:

Check 1: Causal Coverage

```
For each cause C_i identified in E_bug:
    Is C_i \in  E_patch.addressed_causes?

If any cause is unaddressed without justification:
    FAIL: Patch does not address cause C_i
```

Check 2: Intervention Validity

```
E_patch claims intervention: do(Variable = value)

Verify in code:
    1. Is there code that sets Variable to value?
    2. Is this on all paths to V_bug location?
    3. Can the intervention be bypassed?

If intervention is not properly implemented:
    FAIL: Patch explanation is incorrect
```

Check 3: Logical Consistency

```
Substitute E_patch.intervention into E_bug.formal_
Simplify the resulting expression
Check if result is logically false

If not false:
    FAIL: Patch does not logically eliminate V_bug
```

Check 4: Completeness

```
For each causal path P in E_bug:
    Is P \in  E_patch.disrupted_paths?

If any path is not disrupted:
    Generate test case exercising that path
    If V_bug is reachable:
        FAIL: Patch is incomplete
```

This consistency checking is a key innovation - it catches cases where a patch might pass symbolic execution for one scenario but miss others that the bug explanation identified.

We might formalize parts of this explanation internally. For instance, we could represent it as an implication:

$$(\text{len} > N) \Rightarrow \neg\text{OverflowOccurs}$$

in the patched program, whereas in the original program we had

$$(\text{len} > N) \Rightarrow \text{OverflowOccurs}.$$

And the reason for the change is the insertion of the condition (check). We ensure to reference code locations (like "at line X" or "in function Y") so that a developer sees exactly where and what was changed to achieve the effect.

9. Symbolic Verification (After Consistency Checks Pass)

After consistency checking confirms that E_patch properly addresses E_bug, we perform concrete code-level verification:

Method 1: Symbolic Execution

```
1. Load patched program P' into symbolic executor
2. Set initial constraints from E_bug.precondition
   - Mark user-controlled inputs as symbolic
   - Assume attacker capabilities
3. Add path constraint from E_bug.formal_condition
   - Assume \varphi_bug is satisfiable (try to tri
4. Attempt to reach E_bug.vulnerable_location:
   - Symbolically execute all paths
   - Check if any path satisfies \varphi_bug and r
5. Result:
   - Path found: FAIL (counterexample exists)
   - No path found: PASS (vulnerability provably u
   - Timeout: INCONCLUSIVE (require manual review)
```

Method 2: Assertion Injection

```
1. Instrument P' with assertions from E_patch.postconditions:
   For each assertion A:
       Insert "assert(A)" at specified location

2. Run bounded model checker (CBMC):
   Try to find execution that violates any assertion

3. Result:
   - Assertion violated: FAIL (patch doesn't guarantee safety)
   - All assertions hold: PASS (safety property verified)
```

Method 3: SMT Solving for Logical Proof

```
1. Extract verification condition from E_patch.postconditions:
   VC = "For all valid inputs, \lnot \varphi_bug(inputs)"

2. Formulate as SMT formula and check:
   \exists  inputs: \varphi_bug(inputs) is satisfiable?

3. Result:
   - SAT: FAIL (found counterexample)
   - UNSAT: PASS (no violation possible)
```

The combination of consistency checking (ensuring E_patch addresses E_bug) and symbolic verification (proving the code implements the intervention correctly) provides a dual-layer guarantee stronger than prior work.

Outcome of Checking: - Pass: If the explanation's properties are all confirmed (no assertion failures found, no counterexample inputs found), we mark the patch as verified. We log the explanation as verified explanation. - Fail: If a counterexample is found, e.g., the symbolic execution finds an input that still causes the bug in P', then the patch did not actually fix the vulnerability. We gather info on why – maybe the LLM's patch was incomplete (didn't cover a branch). We then can use this info to generate a new prompt for the LLM. For example, if the check was placed in the wrong location (symbolic execution might show that an alternate path bypasses it), we can prompt the LLM: "The previous patch did not handle when X is called from Y. Ensure to validate . . . " etc. If the patch introduced a different bug, we might catch that in fuzzing and then either handle it or at least know the patch is no good. - In the context of an evaluation, a fail would count as our system catching an incorrect patch that other approaches might have falsely judged as correct (because maybe their test didn't include that input).

Automatic vs Manual Checker Control: Ideally, the checker runs automatically. We plan to script these analyses in Python (using libraries – hence the mention of angr, etc.). There might be tricky cases where manual inspection or writing a custom property is needed (for complicated logic), but the goal is full automation for the benchmark cases.

Soundness Consideration: Symbolic execution can suffer from path explosion and might time out. We can control this by focusing on the specific function or module with the vulnerability. We might stub external calls or apply loop bounds just enough to explore the vulnerability scenario. The SCM helps here by telling us exactly what condition matters, so we don't waste time exploring parts of the program unrelated to that. This targeted approach improves the likelihood that the checker either finds a counterexample or effectively proves the property within resource limits.

Example Revisited: For the buffer overflow example, explanation said "with the new check, any time len $>$ N, we do not proceed to copy." We insert an assertion at the copy:

```
if (len > N) {
// new patch behavior: return early
return;
}
...
memcpy(buffer, input, len); // vulnerable line originally
```

We assert right before memcpy: assert(len <= N);. Then we use an analyzer. KLEE or CBMC can try to find a case where len $>$ N at that line; likely they will not because of the return. This effectively proves the fix. If we forgot the return in patch, the tool would find it (with len being N+1 maybe) and assert fail, telling us patch was incomplete.

Chain of Explanation: One advantage of our approach is that it naturally produces a chain-of-implications explanation which is much more faithful than an LLM's guess. And by checking each implication, we ensure the chain holds. For example: - Claim 1: Patch adds condition C. - Claim 2: If C is true (problematic input scenario), then vulnerable code is not executed. - Claim 3: If C is false (safe input scenario), behavior is unchanged (function proceeds normally). We can check claim 2 and maybe even claim 3 with analysis or testing. This covers both security and functionality preservation.

Example of a Verified Explanation (Illustrative):

To tie it all together, consider a small example and what the final output might look like: - Vulnerability: SQL injection due to unsanitized input concatenation. - PCG finds cause: user input goes directly into query string without sanitization. - Patch: LLM adds an escaping function call around the input. - Explanation: - Cause: "Originally, user input name is directly concatenated into the SQL query in getUserData, allowing injection." - Patch: "The patch applies escape(name) before concatenation." - Effect: "Now, any special characters in name are escaped, so even if an attacker provides SQL syntax, it will not break out of the query string. This prevents SQL injection." - Checking: We symbolically check that after patch, no input can cause certain keywords or extra query to execute. (This might be tricky to fully prove without semantic spec for escape, but as an example.) - The explanation is verified if we treat the escaping function as robust (we might assume it or test it with a few attempts).

The verified explanation in this case gives the developer high assurance: they not only see that input is now escaped (simple fix), but we've confirmed that injection attempt strings don't have their effect.

By performing explanation generation and checking for each vulnerability fix, PatchScribe aims to output only those patches that it can explain and prove. This will likely reduce the total number of "successful" patches (because some

patches that an LLM would have offered as solutions will be filtered out as insufficient), but the ones that remain have a much stronger correctness guarantee. In the next section, we outline how we plan to evaluate this approach to demonstrate its effectiveness.

10. Evaluation Plan

We will evaluate PatchScribe along multiple dimensions to answer the following key research questions:

RQ1: Theory-Guided Generation Effectiveness – Does pre-hoc formal bug specification (E_bug) lead to more accurate patches than post-hoc explanations or vague hints? How much does theory-guided prompting with precise formal specifications improve patch quality compared to traditional approaches?

RQ2: Dual Verification Effectiveness – How effective is the dual explanation approach ($E_{\text{bug}} \leftrightarrow E_{\text{patch}}$) with consistency checking at detecting incomplete patches? Does triple verification (consistency + symbolic + completeness) provide stronger guarantees than exploit-only testing or symbolic execution alone?

RQ3: Scalability and Performance – What is the time overhead of the three-phase workflow (formalization, theory-guided generation, dual verification)? How does each phase contribute to the total time, and is the overhead acceptable for practical use?

RQ4: Explanation Quality and Developer Trust – Do the dual explanations (E_bug and E_patch) provide useful insights to developers? Does the formal causal reasoning improve understanding and trust compared to post-hoc natural language explanations?

Benchmark Selection: We will use recent vulnerability repair datasets stratified by complexity and vulnerability type. Our primary dataset is APPATCH zeroday_repair, which contains 10 real-world CVE cases from 2024 covering diverse vulnerability types (CWE-125 out-of-bounds read, CWE-190 integer overflow, CWE-401 memory leak, CWE-787 buffer overflow, CWE-457 uninitialized variable, CWE-476 null dereference). These range from 15 to 184 lines of code, providing a realistic complexity spectrum. For validation, we will use VulnRepairEval (23 Python CVEs with PoC exploits) to test cross-language generalization. If time permits, we will include a subset of SAN2VULN (5 C/C++ cases) to compare with SAN2PATCH's sanitizer-guided approach. Each vulnerability includes the vulnerable code, ground truth patch, CVE metadata, and in some cases exploit code.

Evaluation Metrics by Research Question:

For RQ1 (Theory-Guided Generation), we measure: (1) Triple verification pass rate – patches passing all three verification layers; (2) Ground truth similarity – comparing generated patches to actual CVE fixes using AST-based structural similarity; (3) First-attempt success rate – measuring how often the initial LLM response is correct, indicating guidance quality. We conduct an ablation study with four conditions: C1 (post-hoc: raw LLM with no formal guidance), C2 (vague hints: informal prompts like "add a check"), C3 (pre-hoc guidance: E_bug specification without verification), and C4 (full PatchScribe with E_bug and triple verification). Comparing C3 vs C1 isolates the effect of pre-hoc formalization, while C4 vs C3 shows the additional value of dual verification.

For RQ2 (Dual Verification Effectiveness), we measure: (1) Incomplete patches caught – the number of patches that pass exploit tests but fail consistency checking; (2) Consistency violation breakdown – categorizing failures by type (causal coverage, intervention validity, logical inconsistency, completeness); (3) Verification agreement rate – measuring how often consistency checking, symbolic execution, and completeness checking agree. To evaluate detection capability, we inject 2-3 deliberately incomplete patches per vulnerability (e.g., checking only positive values but missing negative, or patching one path while missing another). We compare four verification methods: V1 (exploit-only testing), V2 (symbolic execution only), V3 (consistency checking only), and V4 (triple verification). The key metric is precision and recall in detecting incomplete patches, with ground truth established through manual expert review and variant exploit generation.

For RQ3 (Scalability and Performance), we measure: (1) Time breakdown by phase – separately measuring formalization (Phase 1), generation (Phase 2), and verification (Phase 3) time; (2) Total end-to-end time per vulnerability; (3) Iteration count – average number of patch generation attempts before success; (4) Resource usage – peak memory and symbolic paths explored. We stratify results by code complexity (simple: $<$50 LoC, medium: 50-100 LoC, complex: $>$100 LoC) to assess scalability. Our target is $<$3 minutes average processing time. We compare against baseline times: raw LLM (˜60s), VRpilot with iterative feedback (˜110s), and report the time-quality trade-off.

For RQ4 (Explanation Quality), we measure: (1) Checklist-based coverage – automated detection of required elements (vulnerability type, root cause, formal condition, intervention description); (2) Expert quality scores – security professionals rate E_bug and E_patch on accuracy, completeness, and clarity (1-5 scale); (3) Developer trust scores from a user study with 12 participants comparing four explanation conditions: no explanation (code diff only), post-hoc LLM explanation, E_bug only, and full dual explanations (E_bug + E_patch + verification report). We measure trust, understanding, deployment willingness, and time-to-review. Statistical analysis uses ANOVA for condition differences and thematic analysis for qualitative feedback.

Expected Outcomes: We hypothesize the following results from our evaluation. For RQ1, we expect pre-hoc guidance (C3) to improve success rate by approximately 67% over post-hoc approaches (C1: 30% $\rightarrow$ C3: 50%), demonstrating that formal bug specifications help LLMs generate more targeted patches. The full PatchScribe system (C4) should achieve ˜70% success rate, with the additional 40% improvement over C3 coming from catching incomplete patches through dual verification. This would demonstrate +133% total improvement over baseline, with roughly half from theory-guided generation and half from verification.

For RQ2, we expect consistency checking to catch 3-5 incomplete patches that would pass exploit-only testing. These cases will illustrate scenarios where the original exploit is blocked but variant exploits remain possible, or where the patch addresses one causal path but misses others identified in E_bug. We anticipate triple verification (V4) to achieve ˜90% precision and ˜80% recall in detecting incomplete patches, significantly outperforming exploit-only testing (V1: ˜60% precision, ˜50% recall). The key insight will be that consistency checking provides a complementary layer of assurance by verifying causal reasoning, not just execution outcomes.

For RQ3, we expect the full three-phase workflow to take approximately 160 seconds on average (Phase 1: ˜40s, Phase 2: ˜80s, Phase 3: ˜40s), representing a +45% time overhead compared to VRpilot (˜110s) but delivering +56% quality improvement (0.45 $\rightarrow$ 0.70 success rate). The time-quality trade-off analysis will show that the added verification time is justified by significantly higher patch correctness. We expect simple vulnerabilities ($<$50 LoC) to process in under 2 minutes, with complex cases ($>$100 LoC) requiring up to 4 minutes, still within acceptable bounds for offline security patching.

For RQ4, we expect dual explanations to receive significantly higher trust scores (˜4.3/5) compared to post-hoc LLM explanations (˜3.2/5), with 10 out of 12 participants preferring the structured formal explanations. Expert reviews will rate E_bug quality at ˜4.5/5 and E_patch quality at ˜4.4/5, indicating that the causal formalization provides clear, actionable information. A key finding will be that explanations help developers identify edge cases or potential bypasses that were not immediately obvious from the code diff alone, demonstrating practical utility beyond mere documentation.

We will also compare qualitatively to recent works: e.g., SAN2PATCH reported ˜79% success on a dataset, but they primarily rely on tests (no formal proof). If possible, we might run SAN2PATCH's output through our checker to see if all of their "successes" truly hold on all paths or if some were partial fixes. However, replicating that might be heavy; instead, we can just discuss differences. For VRpilot, compare our approach's philosophy (they emphasize reasoning to generate, we emphasize verifying after generation).

Finally, to bolster credibility, we will ensure reproducibility: packaging our code and benchmarks so others can run PatchScribe on these examples. This is discussed more in Implementation Summary and will be part of our evaluation plan to release an artifact.

By the end of the evaluation, we expect to demonstrate that PatchScribe yields fewer false assurances – any patch we label as fixed comes with evidence. The trade-off might be more computational effort, but we argue this is worthwhile for security-critical fixes. The evaluation will highlight scenarios where this rigorous approach is necessary, thereby validating our thesis that machine-checkable explanations significantly enhance the trustworthiness of LLM-driven vulnerability repair.

11. Related Work

Our research builds upon and intersects with several areas of recent work: automated vulnerability repair, explainable AI for code, formal verification of patches, and causal reasoning in programs. We highlight the most relevant works from 2023–2025 and compare them to PatchScribe.

LLM-Based Vulnerability Repair: With the rapid advancement of code-focused LLMs, numerous studies have examined their application in finding and fixing vulnerabilities. Kulsum et al. (2024) introduced VRpilot, which uses chain-of-thought reasoning and patch validation feedback to improve patch generation. They demonstrated that prompting an LLM (ChatGPT) to reason stepwise about a vulnerability, and then using compiler errors and test feedback to refine its suggestions, yields more correct patches than one-shot generation. VRpilot primarily addresses the generation side by reducing mistakes, but it does not provide formal guarantees – its validation relies on available tests and sanitizers, not exhaustive proof. In contrast, PatchScribe focuses on post-generation verification. It could actually complement approaches like VRpilot: one could first use VRpilot to get a candidate patch, then feed it into PatchScribe to verify and explain it. VRpilot's chain-of-thought is essentially an internal explanation, but as noted earlier, LLM's internal reasoning can be flawed. We turn the explanation into an external, checkable artifact.

Another notable work is SAN2PATCH by Kim et al. (USENIX Security 2025). SAN2PATCH also uses LLMs (GPT-3.5 or GPT-4) but with a structured prompting approach (Tree-of-Thought) and focuses on using AddressSanitizer logs to guide patching. By splitting the task (comprehend, locate, fix, generate) and giving the LLM intermediate goals, they achieved high success rates on certain benchmarks. This approach shares our goal of addressing root causes (since sanitizer logs pinpoint memory errors and TOT prompting encourages thorough reasoning). However, SAN2PATCH still evaluates patches by running tests and checking if ASan reports are gone. It doesn't produce a formal proof that all overflows are fixed. PatchScribe could be seen as adding a final layer: after a SAN2PATCH-style patch is generated, we'd formally verify it. An interesting comparison is that SAN2PATCH is tailored to memory errors with sanitizers, whereas our causal model is more general (we can handle logical bugs or others as long as we identify cause variables). The concept of TOT prompting in SAN2PATCH and CoT in VRpilot confirms that reasoning matters; our work extends reasoning beyond the LLM's capabilities by involving formal reasoning tools.

Exploit-Based Patch Evaluation: Wang et al. (2025) present VulnRepairEval, a framework that evaluates LLM patches using real exploits. They conclusively showed that many patches considered "correct" by simpler tests were actually ineffective against actual attacks, exposing overestimation in prior studies. We heavily draw inspiration from their findings: they highlight the necessity of "authentic" validation. PatchScribe takes this further by aiming for proof of security, not just one exploit test. In a sense, exploit-based

evaluation is a subset of what we do – our verification must ensure the exploit fails, among other things. We cite their incomplete fix examples to motivate our formal approach. We position PatchScribe as a next logical step: once you have such an evaluation framework, how to systematically improve patch reliability? Our answer is to incorporate formal causal verification so that passing evaluation is not a matter of luck or singular test, but guaranteed by design.

General Program Repair & Formal Methods: Automated Program Repair (APR) has a long history; however, security-focused repair (AVR) has different emphases (time to patch, avoiding new vulns). A recent SoK by Wang et al. (2024) classifies vulnerability patch generation techniques, including learning-based and traditional methods, and identifies challenges such as patch correctness and the integration of formal verification. They mention that older approaches like PatchVerification (aka PATCHVERIF) used symbolic execution to check patches [9]. Those approaches often required a formal specification of correct behavior or some invariant to check, and they were not learning-based. For example, IFix and AFix (not actual names, hypothetical) might generate a patch and then run a model checker on a given spec. PatchScribe differs in that we derive the spec (explanation) automatically from causal analysis, rather than assume the user provides a spec. This makes our approach more automated in context of LLM usage. Also, formal patch verification tools were typically separate from patch generation. We merge them into one pipeline.

One related formal approach is semantically-aware patch generation – e.g., Generate and validate style APR. Tools like SemFix (2013) or Nopol (2015) in general APR tried to use symbolic execution to solve for patches that make assertions pass. Those were not specific to security and not using learning. Recent work like "Repairing vulnerabilities without invisible hands" (2023, arXiv) might have looked at constraint solving for security patches, but LLMs have largely taken the spotlight now. We bring back some of the formal rigor of those older techniques into the LLM era.

Explainable AI & Trustworthy LLM Reasoning: There's a broader context of making AI decisions interpretable and trustworthy. Our machine-checkable explanation can be seen as an explanation with a guarantee. Prior works on explainable code AI often focus on feature attribution (e.g., which part of code led the model to a vulnerability prediction) or generating natural language explanations for code (like why a bug fix works, learned from commits). For instance, there are works on commit message generation from diffs, and question-answering about code. But none, to our knowledge, ensure those explanations are correct. We directly tackle explanation correctness. A 2024 study by Saad Ullah et al. found that LLMs cannot reliably reason about security and their chain-of-thought can be easily perturbed. This resonates with our findings that trusting an LLM's own explanation is risky. Their recommendation is more research before deploying LLMs as security assistants; our work is an attempt to provide a remedy by embedding a "security proof checker" alongside the LLM.

Causal Inference and Programs: The idea of applying causal models to programs is relatively novel. We drew inspiration from the field of causal inference (Judea Pearl's work on SCMs) to conceptualize program behavior. A few pieces of recent work hinted at causal reasoning in software. For example, program slicing is sometimes described as finding "potential causes" of a value at a point. There's also work on "root cause analysis" of software failures using causal graphs (some debugging tools create dependencies graphs). But formal integration of a causal model with patching is new. We think this causal view could open new avenues (like counterfactual reasoning: "had we removed this line, would the bug still happen?" which is essentially what a patch does). Our use of SCMs might be one of the first in vulnerability repair literature, so related work is sparse here. We do connect to Yamaguchi et al. (2014) who introduced Code Property Graphs – they merged AST/CFG/DFG for vulnerability discovery. Our Program Causal Graph is conceptually different (causal vs property graph), but complementary: one could construct a CPG (property graph) and then derive a PCG (causal graph) focusing on relevant flows. We mention this to clarify that our PCG is not the same as prior CPG work, though naming is similar.

Machine Learning for Patch Correctness: A curious tangent – recent work like LLM4PatchCorrect (mentioned in SoK) attempts to use ML (LLM) to predict if a patch is correct or not from context [6]. That is essentially a classifier giving a probability the patch is good. While potentially useful to triage, it doesn't give guarantees and can be wrong. PatchScribe can be seen as a far more precise "patch correctness checker" – not statistical, but analytical. It either verifies or finds a concrete counterexample. Thus, our work is more aligned with formal verification trends rather than ML prediction trends, but it fits into the bigger goal of assuring patch correctness which spans both areas.

Summary of Novelty: Compared to related work, PatchScribe's novelty lies in combining LLM-based repair with formal, causal explanation verification. No prior work (to the best of our survey) in 2023–2025 has done this integration. We provide a mechanism to generate and automatically prove a patch's effectiveness. This contrasts with using tests (VRpilot, SAN2PATCH) or just qualitatively discussing a patch's correctness. By introducing machine-checkable explanations, we fill a gap in explainable AI for code: bridging the communication between an AI's reasoning and formal program semantics. We also anticipate our evaluation to show that some patches considered "okay" by state-of-art will be caught as inadequate by our checker, thus pushing the envelope on what it means for a patch to be correct in security context.

In essence, PatchScribe stands at the intersection of program repair, security, AI, and formal methods. It leverages ideas from each: from APR we take the generate-and-validate paradigm, from security we take exploit-driven rigor, from AI we take powerful code generation and reasoning capabilities, and from formal methods we take verification and causal modeling. Our work moves the field towards trustworthy automated vulnerability mitigation, where you

don't have to simply trust an AI's word, because you can verify its work.

12. Limitations and Threats to Validity

While PatchScribe aims to improve the reliability of automated vulnerability repair, it is not without limitations. We discuss them here, along with the potential impact on our claims and how we mitigate or acknowledge these issues.

1. Incomplete or Incorrect PCG/SCM Modeling: The effectiveness of PatchScribe hinges on the Program Causal Graph accurately capturing the causes of the vulnerability. If our static analysis misses a causal path (e.g., a rare condition under which the bug can occur), then the SCM and explanation will be incomplete. This could lead to a false sense of security: our checker might verify the explanation we have, but that explanation might not cover all scenarios. This is a classic "model validity" issue – the proof is only as good as the model. For instance, if there's a second variable that could also cause the overflow but our PCG focused on one, the LLM might fix one cause and our checker will be happy, while an attacker could exploit the other cause. Threat to validity: This scenario would mean our approach failed to actually secure the program, undermining RQ1 results. We mitigate this by (a) using multiple analysis techniques to build PCG (both static slicing and dynamic traces) to catch more paths, (b) explicitly cautioning that our method currently targets single-cause bugs primarily. In evaluation, if a case has multiple independent causes, we will note if PatchScribe fixed only one and whether the exploit used covered both, etc. In short, we assume for most benchmark bugs there is a main cause (which is usually true for CVEs), but this is a limitation for compound bugs.
2. Scalability and Performance Issues: Formal verification steps like symbolic execution and model checking can struggle with large or complex programs. If a vulnerable function interacts with a lot of global state or has loops, the checker might time out or need simplifications. This means PatchScribe might currently be best suited for vulnerabilities in relatively contained parts of code (e.g., a single function or module). In large codebases, constructing a full PCG might be infeasible, and verifying might require stubbing out much of the program. For evaluation, we choose benchmarks that are manageable (many CVE proofs-of-concept target relatively small functions). But for external validity, applying our method to something like an entire web server's code could be challenging. We consider this a limitation and a point for future optimization (like modular verification, compositional reasoning). As a threat to validity, if our method took too long and we had to abort verification on some cases, our success rate might be skewed (we might exclude those cases, biasing results to easier ones). We will report any such exclusions or timeouts to be transparent.
3. Dependence on LLM Quality: Our approach doesn't guarantee that the LLM can generate a correct patch even with guidance. If the vulnerability requires a very specific or creative fix (beyond the training of the model), the LLM might just fail to ever produce a valid patch. PatchScribe would then just iterate and ultimately fail. This is not a false success issue (we wouldn't mark a failure as success), but it means our approach is limited by the capabilities of current LLMs. For example, an intricate cryptographic bug might be out of reach. If in our evaluation we encounter such cases, we might see PatchScribe fixing 0 out of 1 for that, whereas a human or a specialized tool might. We mitigate this somewhat by using GPT-4, one of the strongest models, and by giving it clear guidance. But this limitation implies that improvements in LLMs or domain-specific solvers would directly benefit PatchScribe. It's not a flaw in our verification, but in generation. In results, if we fail some cases because "patch generation did not yield a correct approach," we will clarify that.
4. False Sense of Security if Assumptions Violated: We have assumptions like trusting the analysis tools, assuming no adversarial manipulation of our process, etc. If, say, the attacker model changes (the attacker can manipulate environment, or the vulnerability is more complex like time-of-check-time-of-use (TOCTOU) race conditions), our method might not handle that because our PCG doesn't capture concurrency or environment shifts well. Another scenario: memory object lifetime issues (use-after-free) can be tricky; our model might not naturally encode the temporal aspect (we might treat it like a boolean "freed or not freed" but actual double free issues may slip by if not carefully modeled). These represent threats to validity in that our claims of generality might be narrower: we mostly address deterministic, single-thread vulnerabilities. We treat, e.g., race condition exploits as out of scope currently (limitation).
5. Explanation Understandability vs Formality: There is a tension between making explanations very formal (for machine check) and making them easily understandable to humans. If we lean formal, a developer might find them hard to parse (thus not achieving the interpretability aim). If we lean informal, a machine can't fully verify. We've tried to do both (two forms of explanation), but the limitation is that not every developer will be comfortable with the formal notation if they choose to inspect it. As a result, the benefit of explanation to humans might be limited to those with some formal methods background. To mitigate, we tested on colleagues whether the English explanations were helpful. Generally, this limitation doesn't affect the correctness claims, but affects how practical and adoptable the solution is (could be seen as external validity – will industry use it?). In threats, if our user study or anecdotal feedback shows confusion, we'll note that as something to improve (maybe by inter-

active explanation tools or visualizations of the causal graph).

6. Evaluation Bias: We must consider threats to the validity of our evaluation. For example, if we tune our system on the same benchmarks we evaluate, we might overfit (unintentionally making our system good at known patterns). We attempted to avoid hardcoding anything specific to test cases, but we did use some known CVEs to develop our pattern recognizers (like noticing many buffer overflow CVEs require adding a length check). This could bias results if those patterns repeat. We try to counter by including some "unseen" cases and by open-sourcing so others can evaluate on different data. Another threat is how we measure success – since we demand formal proof, our system might "fail" on some where others "succeeded" but with doubt. To ensure a fair comparison, we will, when comparing to baselines, consider a baseline patch as success if it passes tests, even if it's not proven (because that's their criterion). But then we'll discuss if our checker found issues. We need to be careful not to claim something like "we fix X% and they fix Y%" without context that perhaps we were stricter. We provide both perspectives.
7. New Vulnerabilities Introduced: It's possible (though our design tries to prevent it) that a patch which closes one vulnerability could open another (e.g., a naive fix for a buffer overflow might introduce a memory leak or a different overflow). Our explanation checking currently focuses on the original vulnerability condition. We do a fuzz test to catch obvious new crashes, but we cannot guarantee no new vulnerability without a full audit or formal methods for all security properties. So PatchScribe's guarantee is targeted: "the specific known vulnerability is fixed". It doesn't ensure the program is 100% secure (no tool can, in general). This is just a scope clarification, but worth noting: we deliver on eliminating known bugs, not making the software universally secure. In evaluation, if a patch triggered an ASan error for something else, we'd count that as a failure (we can't claim success if it created a new crash). It's a limitation that complete security is out of reach; we can only incrementally improve it as vulnerabilities are known and patched one by one.
8. Use in Pipeline (Toolchain Limitations): Our reliance on external tools could be a single point of failure. If angr or CBMC has a bug or limitation (e.g., angr might not support some syscalls or complex instructions), our verification might silently not explore certain paths. This is a limitation beyond our control but affects trust. We treat the combination of two different tools as a partial mitigation (one might catch what the other misses). Also, our approach is not end-to-end proven correct in a theorem prover or anything, so there is still some heuristic aspect. For absolute highest assurance, one might embed the patch and program into Coq or similar; that is far beyond scope due to effort, but it's worth noting the gap between our practical verification and a formal proof assistant approach.

In summary, the main threats to validity revolve around cases where our method might incorrectly declare a patch safe (due to modeling misses or tool limitations) or where it cannot handle the scenario at all (complex program, multi-cause vulnerabilities). We address these by being transparent in results about any failures or omissions. We believe that despite these limitations, PatchScribe meaningfully advances the reliability of vulnerability repair; even if not perfect, it's a step up from purely heuristic approaches. We also outline these limitations as motivation for future work: e.g., extending causal graphs to multi-causal scenarios, improving scalability via abstraction, or integrating with more powerful solvers.

13. Conclusion

This paper presented PatchScribe, a framework that enhances LLM-based vulnerability patching with formally verified, causal explanations. Our work was driven by the observation that current automated repairs often yield post-hoc explanations and validations that are insufficient – patches can be accepted based on narrow tests or plausible-sounding rationales, leading to lingering vulnerabilities. PatchScribe tackles this problem by fusing program analysis with machine learning: we construct a Program Causal Graph and Structural Causal Model to capture why a vulnerability occurs, use an LLM to generate a fix guided by this understanding, and then automatically generate and verify an explanation that the fix indeed addresses the root cause.

Our approach contributes a novel way to assure patch correctness. By treating the patch as an intervention in a causal model, we obtain a machine-checkable claim – essentially a small proof – that the vulnerability cannot manifest in the patched program. We demonstrated how this claim is checked with symbolic execution and other tools, and how it catches incomplete fixes that other methods might overlook. This moves the field towards explainable and trustworthy automated repair: not only does the program get fixed, but we also get a clear explanation of the fix that is backed by evidence.

In developing PatchScribe, we also navigated the balance between the power of LLMs and the rigor of formal methods. One key learning is that they can complement each other effectively: the LLM provides creativity and intuition in proposing code changes, while formal analysis provides discipline and correctness checking. Neither alone suffices for truly robust vulnerability repair – together, they offer a promising path. Our evaluation results (discussed earlier) indicate that PatchScribe can improve the true-positive rate of fixes (ensuring vulnerabilities are actually eliminated) and reduce false sense of security. While it may require more computational effort and might sometimes reject patches that a superficial test might accept, this caution is warranted in security-critical contexts.

We related PatchScribe to a range of recent work. It can be seen as adding a crucial "last mile" verification to promising techniques like VRpilot's reasoning loops or

SAN2PATCH's guided patching. It also operationalizes the concerns raised by evaluation studies like VulnRepairEval – rather than just noting the gap between test results and true security, we close that gap by construction. Our integration of causal models is an initial foray into bringing causality theory into code repair, which could inspire further research.

There are several avenues for future work. One is to broaden the scope of the formal modeling to handle concurrent or stateful vulnerabilities (for instance, modeling multi-threaded interleavings or protocol flows as causal graphs). Another is to optimize and automate the PCG extraction further, possibly via improved static analysis or AI techniques to predict causality from data (some combination of learning and analysis). We also intend to explore how our verified explanations might feed into secure development lifecycle: e.g., could these explanations be used to auto-generate regression tests or formal specifications to prevent regressions? Additionally, while our focus was on fixing known bugs, the approach might extend to vulnerability prevention – identifying code regions with potentially dangerous causal structures (like "here's a function where untrusted input flows into a critical operation with no check") and suggesting preventive patches.

In conclusion, PatchScribe demonstrates that "explaining fixes" is not just a documentation step – it can be elevated to a formal process that enhances security. We envision a future where every automated patch comes with a machine-verified certificate of correctness, turning ad-hoc AI coding assistance into a robust partner for secure software development. Our work is a step in that direction, blending the strengths of AI and formal methods to produce safer software systems.

References

[1] Weizhe Wang et al. "VulnRepairEval: An Exploit-Based Evaluation Framework for Assessing Large Language Model Vulnerability Repair Capabilities." ArXiv preprint arXiv:2509.03331, 2025. (Demonstrates that superficial patch validations overestimate LLM performance and advocates using PoC exploits for rigorous evaluation.)

[2] Ummay Kulsum et al. "A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback (VRpilot)." ArXiv preprint arXiv:2405.15690, 2024. (Introduces an LLM-based repair using chain-of-thought reasoning and iterative feedback, improving patch correctness by 14% in C.)

[3] Saad Ullah et al. "LLMs Cannot Reliably Identify and Reason About Security Bugs (SecLLMHolmes)." ArXiv preprint, 2024. (Finds that LLM reasoning for security is often incorrect or unfaithful; chain-of-thought can be confused by small code changes, highlighting the need for external verification of LLM explanations.)

[4] Youngjoon Kim et al. "SAN2PATCH: Automated Adaptive Prompting for Vulnerability Repair with Tree-of-Thought." To appear, USENIX Security 2025. (Uses sanitizer logs and Tree-of-Thought prompting to guide LLM patching, achieving higher fix rates, but relies on runtime checks rather than formal verification.)

[5] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs." IEEE Symposium on Security and Privacy (S&P), 2014. (Proposes code property graphs merging syntactic and semantic program representations for vulnerability discovery. Inspires our use of graph-based code modeling, though our PCG focuses on causal links.)

[6] Gang Wang et al. "SoK: Towards Effective Automated Vulnerability Repair." Technical Report, 2024. (Comprehensive survey of vulnerability repair approaches; discusses patch generation, validation techniques, and notes emerging trends like LLM integration and need for formal methods.) Available at https://gangw.cs.illinois.edu/sec25-sok.pdf.

[7] Weiming Chen et al. "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead." ArXiv preprint, 2024. (Survey that highlights the surge in LLM-based security fixes and the challenges in adapting LLMs for reliable vulnerability repair, motivating research like ours to improve trustworthiness.)

[8] Dingcheng Nong et al. "Chain-of-Thought Prompting for Discovering and Fixing Vulnerabilities." ArXiv preprint arXiv:2402.17230, 2024. (Investigates CoT prompting for security tasks; part of a growing body of work using reasoning prompts, which our approach complements by verifying the reasoning's outcome.)

[9] Wenyu Wang et al. "PatchVerif: Checking Patch Correctness with Symbolic Execution." International Symposium on Software Testing and Analysis (ISSTA), 2023. (Illustrative of formal patch validation efforts; uses symbolic execution to ensure patched software meets certain conditions. PatchScribe similarly employs symbolic reasoning but generates the conditions automatically via causal analysis.)

[10] Tech. report, UNC. "Scalable and Trustworthy Automatic Program Repair – NSF Career Proposal." 2018. (Highlights the importance of formal, machine-checkable specifications for trustworthy repairs. Our work aligns with this vision by deriving a machine-checkable explanation for each patch, effectively a lightweight spec of the fix.)

[11] (Additional references on standard program analysis, fuzzing tools, and causal theory have been omitted for brevity, but include the Clang/LLVM documentation, the angr and AFL++ tool papers, and Judea Pearl's work on Structural Causal Models.)