

PBFuzz: Agentic Directed Fuzzing for PoV Generation

Haochen Zeng
hzeng013@ucr.edu

Department of Computer Science and Engineering
University of California, Riverside
Riverside, California, USA

Jiajun Cheng
jchen1192@ucr.edu

Department of Computer Science and Engineering
University of California, Riverside
Riverside, California, USA

Andrew Bao
bao00065@umn.edu

Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, Minnesota, USA

Chengyu Song
csong@cs.ucr.edu

Department of Computer Science and Engineering
University of California, Riverside
Riverside, California, USA

Abstract

Proof-of-Vulnerability (PoV) input generation is a critical task in software security, and has numerous downstream applications such as path generation and validation. Fundamentally, generating a PoV input must solve two sets of constraints: (1) *reachability constraints* for reaching the vulnerable code location(s), and (2) *triggering constraints* for triggering the target vulnerability. Unfortunately, existing approaches, including directed greybox fuzzing and LLM-assisted fuzzing, struggle to effectively and efficiently solve these constraints. This paper presents an agentic approach that mimics human experts for PoV input generation. Human experts iteratively analyze code to extract semantic-level reachability and triggering constraints, hypothesize PoV triggering plans, encode them as test inputs, and leverage debugging feedback to refine their understanding when plans fail. We automate this process with PBFuzz, an agentic directed fuzzing framework. PBFuzz addresses four critical challenges in agentic PoV generation. First, autonomous code reasoning enables dynamic hypothesis validation through semantic constraint extraction. Second, custom MCP tools provide on-demand program analysis for targeted constraint inference. Third, persistent memory prevents hypothesis drift across long-horizon reasoning tasks. Fourth, property-based testing enables efficient constraint solving while preserving structural validity. Experimental evaluation on the Magma benchmark demonstrates decisive superiority. PBFuzz triggered 57 vulnerabilities, outperforming all baselines. Critically, PBFuzz exclusively triggered 17 vulnerabilities compared to existing fuzzers. PBFuzz achieved this within a 30-minute budget per target, compared to 24-hour allocations for conventional approaches. Median time-to-exposure: 339 seconds for PBFuzz versus 8680 seconds for AFL++ with CmpLog, representing a 25.6 \times efficiency gain with API cost of \$1.83 per vulnerability.

1 Introduction

Generating a proof-of-vulnerability (PoV) input that triggers a vulnerability at target code location(s) is a critical task in software security. Such inputs are essential for vulnerability verification, reproduction, triage, patch validation, and regression testing. The state-of-the-art (SOTA) approach for this task is directed greybox

fuzzing, where, instead of trying to maximize overall code coverage [25, 29, 56], directed greybox fuzzers (DGFs) aim to generate test inputs to exercise *specific functionalities* or *vulnerable code locations* in the program under test (PUT) [66, 67].

DGFs typically follow a two-step process. First, they perform static analysis to extract guidance (e.g., a distance metric for reaching the target locations) from the PUT’s low-level program structures such as the control-flow graph (CFG), definition-use graph (DUG), and data-flow graph [66, 67]. During fuzzing, DGFs use this guidance to evaluate generated inputs, prioritize scheduling of seeds that are more likely to reach the target, and perform more mutations on those seeds. Intuitively, one might expect DGFs to excel at reaching target locations faster. However, recent research [10, 27, 35, 59] has shown that DGFs struggle to generate PoV inputs; in some cases, they even underperform coverage-guided fuzzers like AFL++ [25, 26]. For example, our experimental results (Figure 5) on the Magma benchmark show that AFLGo can trigger only 41 CVEs, while AFL++ with Cmplog triggered 49. These results suggest that, despite their goal-oriented design, DGFs are neither particularly effective nor efficient at generating PoV inputs.

With recent advancements in large language models (LLMs), there has been growing interest in leveraging LLMs’ ability to generate syntactically valid and semantically plausible inputs to address diverse challenges in fuzzing, including: format-aware input generation [50, 63, 77], coverage plateaus [61, 68], semantics-aware mutations [23, 68], and vulnerability-targeted exploration [24, 70, 78]. A natural question therefore arises: can LLMs help PoV input generation?

To answer this question, we conducted preliminary experiments with one-shot chatbot-style prompting. Specifically, we prompted SOTA LLMs to generate PoV inputs for various vulnerabilities from the Magma benchmark [33], using code snippets extracted by static analysis (more details in §5.5). The results were mixed: while LLMs could generate valid PoV inputs for simple vulnerabilities (16 out of 129 CVEs), they struggled with more complex ones (e.g., PHP and libtiff).

Fundamentally, a PoV input must satisfy two sets of constraints: (1) *reachability constraints* that enable execution to reach the target location(s), and (2) *triggering constraints* that enable execution to trigger the target vulnerability. Therefore, to successfully generate PoV inputs, we must solve three key challenges: (1) accurately extract the reachability and triggering constraints for the target

vulnerability, (2) efficiently solve these constraints, and (3) effectively encode the solutions into test inputs. Our investigation of the failed cases revealed three main limitations of chatbot-style LLM-assisted fuzzing. First, while SOTA LLMs have demonstrated substantial capabilities in code understanding and reasoning, long-context reasoning remains a major challenge. That is, if we use static analysis to extract all code snippets that *may* be relevant to reaching and triggering the target vulnerability, and ask LLMs to find the *necessary* conditions from these snippets in one shot, they often get distracted and fail to do so. Second, even when we provided feedback to the LLMs (e.g., whether the generated input triggered the vulnerability) and iteratively prompted them to refine their input generation strategies, they frequently became trapped in an incorrect set of constraints and failed to recover. Finally, directly asking LLMs to generate PoV inputs is slow and costly; they also struggle to precisely solve complex constraints.

Our Approach. We propose PBFuzz, an *agentic approach* to unlock LLMs’ reasoning and planning capabilities for effective and efficient PoV input generation. Rather than treating LLMs as auxiliary components to traditional fuzzers (and static analysis), PBFuzz leverages LLMs as autonomous agents that can perform high-level code reasoning, planning, tool orchestration, and self-reflection to *iteratively*

- (1) Analyze code and extract reachability and triggering constraints,
- (2) Design solving strategies and encode them as parameterized input generators,
- (3) Leverage property-based testing [21, 28] to systematically search the input space for PoV inputs, and
- (4) Collect fine-grained execution feedback to validate and refine their hypotheses.

This agentic design allows us to address the aforementioned limitations. First, instead of overwhelming LLMs with extensive code snippets and asking them to identify the critical elements, LLM agents can explore the codebase based on their own hypotheses regarding the reachability and triggering conditions. As highlighted in our case studies (§5.3), this approach significantly improves the accuracy of constraint extraction and the efficiency of solving. Second, LLM agents can seek custom fine-grained feedback to refine their hypotheses; with persistent memory, they can also abandon their current plans and backtrack to pursue new hypotheses. Finally, with property-based testing [21, 28], LLM agents can efficiently explore the input space without violating reachability constraints.

Indeed, LLM agents have been explored in security domains [58], including DARPA’s Artificial Intelligence Cyber Challenge (AIxCC) [4]. At the same time, we have observed the emergence of coding agents like OpenAI Codex [17], Claude Code [7], and Cursor [3]. This raises a natural question: why do we need a specialized design for PoV-generation agents? The answer is that, even with a powerful coding agent, an effective PoV-generation system still needs to address four key challenges:

- *Challenge 1: Dynamic Hypothesis Validation.* Traditional directed fuzzers compute guidance up front via fixed analyses. Agentic fuzzing requires dynamic context search: agents develop hypotheses about input constraints, then autonomously determine and gather the program information needed to validate them [73].

- *Challenge 2: Persistent Memory Management.* Long-horizon tasks like PoV generation risk memory drift, where agents lose track of validated reasoning and stagnate [62].
- *Challenge 3: Fine-Grained Execution Feedback.* Traditional fuzzing provides only coarse-grained coverage feedback, which is insufficient for hypothesis refinement when tests fail [8, 65]. Reasoning errors can cascade in agentic systems [79], necessitating timely detection and correction.
- *Challenge 4: Efficient Constraint Solving.* LLM inference is orders of magnitude slower than traditional fuzzing [72], making direct test generation impractical at scale. Furthermore, LLMs lack precision in solving complex arithmetic constraints required for directed fuzzing [37]. While LLMs can identify vulnerability conditions through semantic reasoning, systematic exploration of constraint hierarchies exceeds their inference capabilities.

Our four-phase workflow design (Figure 2), together with a custom toolset and memory management, enables PBFuzz to effectively address these challenges.

To demonstrate the capability of our approach, we implemented a prototype of PBFuzz based on the cursor-cli tool. Experiments on the Magma benchmark [33] show that our agentic approach significantly outperformed previous solutions. PBFuzz successfully triggered 59 out of the 129 CVEs, including 17 CVEs that none of the previous approaches triggered. More importantly, PBFuzz achieved these results with a single trial and a 30-minute budget, while baseline fuzzers were run with 10 trials of 24 hours each. Our ablation study also confirms the contributions of each design component and major innovation.

Contributions. This paper makes the following key contributions:

- We present a novel approach for effective and efficient PoV input generation, where we first conduct code analysis to extract semantic-level reachability and triggering constraints, then leverage property-based testing to solve the constraints at the input-space level.
- We introduce an agentic framework that realizes and automates our approach. Our agent, PBFuzz, employs a custom four-phase workflow that features autonomous code reasoning, on-demand tool orchestration, persistent memory management, fine-grained execution feedback, and property-based test generation.
- We provide experimental validation of our approach on the Magma benchmark, which shows that PBFuzz outperformed all baselines, including traditional fuzzers, LLM-assisted fuzzers, and off-the-shelf coding agents.

2 Background

2.1 Fuzzing and Directed Fuzzing

Coverage-guided greybox fuzzing (CGF) is an automated software-testing technique that dynamically feeds mutated inputs to programs to discover vulnerabilities. CGF employs lightweight instrumentation to collect runtime code-coverage feedback, guiding input generation toward unexplored program paths through evolutionary mutation. Tools such as AFL [75] and libFuzzer [56] have discovered thousands of vulnerabilities.

Despite its success, CGF blindly expands code coverage and lacks prioritization. Only a small fraction of source code contains vulnerabilities (e.g., merely 3% of Mozilla Firefox files [66]). Blindly maximizing coverage wastes resources on irrelevant code regions, limiting efficiency in reaching specific vulnerability targets.

Directed greybox fuzzing (DGF) aims to address this by focusing testing resources on specific target locations, such as recently patched code or sensitive operations [13, 66, 67]. It does so by computing distance metrics between current program locations and targets, prioritizing seeds that are closer to these targets, and allocating more mutations to such seeds. Unfortunately, as discussed in §1, recent studies and our own experiments show that existing DGF techniques are often ineffective despite their design goals.

2.2 Property-Based Testing

Software testing traditionally relies on manually crafted test cases that verify specific input–output pairs. While intuitive, this approach suffers from limited coverage and difficulty in anticipating edge cases. Property-based testing (PBT) offers a different paradigm: instead of writing individual test cases, developers specify general properties that their programs should satisfy, and automated tools generate extensive, diverse inputs to verify these properties [52]. Modern PBT frameworks have evolved significantly since QuickCheck’s introduction in Haskell [21]. Contemporary implementations include Hypothesis [49] for Python, Google FuzzTest [31] for C++, and proptest [1] for Rust.

Consider the buffer overflow vulnerability in libxml2’s `xmlSnprintfElementContent` function (Figure 1). Instead of writing specific test cases, PBT expresses the safety property that this function should never overflow as executable specifications [2]:

```
void XmlSnprintfNeverOverflows(int nesting_depth,
    int prefix_len, int name_len, ContentModelType type) {
    auto xml = GenerateXML(nesting_depth, prefix_len,
        name_len, type);
    xmlSnprintfElementContent(buf, size, content, englob);
    // Sanitizer detects overflow if triggered
}
FUZZ_TEST(Libxml2Test, XmlSnprintfNeverOverflows)
    .WithDomains(InRange(1,50), InRange(1,200),
        InRange(1,200), Arbitrary<ContentModelType>());
```

This example illustrates how input domains encode vulnerability-triggering constraints. The framework generates diverse input combinations, efficiently exploring the input domain to uncover violations.

2.3 Property-Based Directed Fuzzing

Table 1: Property-Based Testing vs. Fuzzing

Aspect	Property-Based Testing	Fuzzing
Goal	Verify invariant properties	Discover vulnerabilities
Bug Oracle	Explicit assertions	Implicit sanitizers
Input Space	Typed parameter domains	Raw byte arrays
Scope	Functional correctness	End-to-end testing
Duration	Minutes to hours	Hours to days

As Table 1 shows, PBT and fuzzing are complementary approaches. PBT operates on typed parameter domains with explicit

correctness specifications, encoding domain knowledge in generators. Fuzzing mutates raw byte arrays, guided by indirect coverage feedback. While both aim to discover software defects, they differ fundamentally in input representation and constraint encoding.

In this work, we make a key observation: in directed fuzzing, triggering the target vulnerability represents a special case of safety properties, with sanitizers acting as implicit oracles. For instance, a buffer overflow vulnerability violates the safety property that all array accesses must remain within bounds. This perspective transforms directed fuzzing into a specialized form of property-based testing with two key components:

- (1) **Preconditions:** Constraints that enable program execution to reach the target vulnerability location.
- (2) **Triggering condition:** The condition at the target location that triggers the actual vulnerability.

Therefore, *finding PoV inputs can be reduced to discovering counterexamples that violate safety properties*: inputs that satisfy both reachability constraints and triggering conditions.

However, PBT faces a fundamental challenge: developers must write generators that produce diverse valid inputs satisfying complex structural preconditions [28]. Straightforward preconditions like positive integers are trivial to generate randomly, but real-world constraints demand deep domain knowledge—well-formed XML, balanced red-black trees, and valid S-expressions rarely emerge from naive generation. Designing such generators requires substantial engineering effort, often rivaling the complexity of the code under test itself [28].

Recent work has explored LLM-assisted PBT generation from specification documents, synthesizing tests from API documentation [64] or inferring properties from code through agentic analysis [48]. Our agentic approach also addresses the generator-synthesis bottleneck. But unlike specification-based approaches that generate tests from API documentation, our approach targets PoC input generation, which requires extracting implicit constraints from entry points to target locations in the PUT’s code. Our autonomous LLM agent PBFUZZ conducts program analysis, inspects execution states, and refines hypotheses through feedback to extract constraints for unspecified security vulnerabilities. The agent synthesizes typed parameter domains and generator functions that encode learned constraints as testable input domains.

3 A Motivating Example

Figure 1 shows a buffer overflow in the libxml2 project. The bug occurs when processing namespace prefixes: after concatenating the prefix (line 19), the function still uses the stale `len` to concatenate the element name (line 23), resulting in a buffer overflow when deeply nested content models trigger recursive calls (lines 26–33).

Generating a proof-of-vulnerability (PoV) input to trigger this bug requires satisfying four types of constraints. The first three are *reaching constraints* to reach the vulnerable code, and the fourth is the *triggering constraint* that causes the overflow. First, *format-level*: the PoV input must be syntactically valid XML with well-formed Document Type Definition (DTD) ELEMENT declarations. Second, *structure-level*: the DTD content models must use nested SEQ/OR operators like `(a, (b | (c, d)))` rather than flat lists like

```

1 void xmlSnprintfElementContent(char *buf, int size,
2     xmlElementContentPtr content, int englob) {
3     int len;
4     if (content == NULL) return;
5     len = strlen(buf);
6     if (size - len < 50) {
7         if ((size - len > 4) && (buf[len - 1] != '.')) {
8             strcat(buf, "...");
9             return;
10        }
11    if (englob) strcat(buf, "(");
12    switch (content->type) {
13        case XML_ELEMENT_CONTENT_ELEMENT:
14            if (content->prefix != NULL) {
15                strcat(buf, (char *) content->prefix); //BUG
16                strcat(buf, ":" );
17            }
18            if (content->name != NULL)
19                strcat(buf, (char *) content->name); //BUG
20            break;
21        case XML_ELEMENT_CONTENT_SEQ:
22            xmlSnprintfElementContent(buf, size, content->c1, 1);
23            strcat(buf, " , ");
24            xmlSnprintfElementContent(buf, size, content->c2, 0);
25            break;
26        case XML_ELEMENT_CONTENT_OR:
27            xmlSnprintfElementContent(buf, size, content->c1, 1);
28            strcat(buf, " | ");
29            xmlSnprintfElementContent(buf, size, content->c2, 0);
30            break;
31    }
32    if (englob) strcat(buf, ")");
}

```

Figure 1: CVE-2017-9047 buffer overflow in libxml2’s `xmlSnprintfElementContent()` function. The bounds check uses the stale buffer length `len` instead of the updated `strlen(buf)` after appending namespace prefixes, allowing writes beyond the allocated memory.

(a, b, c, d) to trigger recursive accumulation. Third, *namespace-level*: elements must have prefixes (`prefix:name`) to reach lines 18–23. Fourth, *depth-level*: nesting depth must reach approximately ten levels to accumulate sufficient buffer consumption (about 70-character prefixes/names per level) and exhaust the 5000-byte buffer.

We first evaluated how chatbot-style LLM-assisted fuzzing would perform. We added the source code of the entry function and the target function to the context and prompted different LLMs (ChatGPT, Claude, Gemini) to generate an input that can satisfy all the conditions. The results were negative: only GPT-5 and Claude 4.5 produced inputs that reached the target function, but none could trigger the vulnerability. Adding a fuzzer would not help either: the initial seeds provided by the Magma benchmark already reached the target code, yet AFL++ still required approximately 20 hours to trigger the bug, succeeding in six out of ten trials.

We also evaluated whether off-the-shelf LLM agents can autonomously generate the PoV input. We configured Cursor Agent with Claude Sonnet 4.5 and prompted it to reproduce CVE-2017-9047 in the libxml2 codebase. The agent successfully triaged the vulnerability: it identified the recursive function

structure, recognized namespace-prefix requirements, and understood the buffer overflow mechanism. However, despite this semantic understanding, the agent failed to generate the PoV. The generated inputs consistently used flat DTD structures like `(e0, e1, ..., e399, LongElement, PrefixedElement)` with 400 elements and extremely long names/prefixes (200 characters), which reached the target function but never triggered the overflow because flat structures prevent recursive accumulation.

The agent’s failure reveals four problems that map directly to the challenges identified in §1. First, *static-analysis only*: the agent analyzed the code upfront but could not dynamically gather context during exploration—when flat structures failed, it lacked mechanisms to inspect runtime state and understand *why* buffer consumption remained low. Second, *hypothesis drift*: without persistent memory, the agent repeatedly tested similar flat structures, losing track of the validated insight that recursive calls drive buffer accumulation. Third, *coarse-grained feedback*: binary “reached/not triggered” outcomes provided no gradient information about buffer consumption at each recursion level, preventing the agent from refining its understanding of the depth×breadth interaction. Fourth, *inefficient direct generation*: the agent generated each test case through expensive LLM inference, testing only 15 cases before terminating—insufficient to explore the critical nesting-depth dimension systematically. Recent work [37] also demonstrates that LLMs cannot solve complex structural constraints through inference alone; discovering nesting depth requires systematic exploration over parameterized input dimensions. Although off-the-shelf LLM agents can analyze code and identify vulnerability conditions, they cannot autonomously translate these insights into effective input-generation strategies for PoV generation.

However, the case study also reveals a promising opportunity: LLM agents possess the semantic reasoning capabilities to understand vulnerability conditions, which traditional approaches like directed fuzzing struggle to exploit. Our key insight is that *property-based testing provides the missing bridge between semantic analysis and effective constraint solving*. Rather than generating test inputs directly through expensive LLM inference, agents can synthesize *parameterized input generators*—functions that encode vulnerability preconditions as typed parameter spaces, enabling high-throughput systematic exploration while maintaining semantic guidance.

In addition to this novel workflow, PBFuzz also addresses the aforementioned challenges through integrated design. For dynamic context search, we provide on-demand debugger access revealing runtime state at critical program points—enabling agents to understand *why* test cases fail, not just *that* they fail. To prevent hypothesis drift, we maintain persistent structured memory in the form of structured workflow documents that store validated insights, as well as previous failures and successes across iterations. We also provide fine-grained feedback showing exact divergence points where execution exits before reaching targets.

Applied to CVE-2017-9047, debugger feedback reveals that flat DTD structures produce low buffer consumption despite reaching the target. This guides the agent to hypothesize that nesting depth drives recursive accumulation, leading it to design a five-dimensional parameter space explicitly including `nesting_depth` alongside `element_prefix_length`, `element_name_length`, `content_model_type`, and `num_elements`. The

synthesized generator creates recursive structures mirroring the target function’s control flow. Systematic sampling discovers the triggering configuration at iteration 156: depth 10 with 70-character prefixes and names successfully triggers the vulnerability—demonstrating how agentic property-based fuzzing bridges the gap between semantic understanding and effective vulnerability discovery.

4 Design

4.1 Overview

4.1.1 Prerequisites. Similar to directed fuzzers, PBFuzz aims to generate proof-of-violation (PoV) inputs to confirm the existence of a target vulnerability in a program under test (PUT). In this work, we assume the following two prerequisites:

- Local vulnerability triggering condition. We assume an existing analysis, either manual or automatic, either patch-based or static-checker-based, has identified the safety properties that may be violated.
- Reaching and triggering signals. We assume the PUT has been instrumented (either manually or automatically) with logging statements to indicate whether the vulnerability site was reached and whether the triggering condition was satisfied during execution.

For our main evaluation dataset, the Magma benchmark [33], both prerequisites are already satisfied. While we believe LLM agents can address these prerequisites, they are orthogonal to our main contribution in this work, and we leave them as future work.

4.1.2 Architecture. PBFuzz employs a four-layer architecture to enable agentic property-based directed fuzzing, as illustrated in Figure 2. At a high level, the **workflow layer** defines a state machine with four phases: PLAN and REFLECT phases for inferring and refining semantic constraints for triggering the vulnerability; IMPLEMENT and EXECUTE phases for encoding the constraints into input-level parameter spaces and efficient constraint solving. Beneath it, the **LLM agent** acts as the “brain” to perform semantic reasoning to (1) generate hypotheses about reaching and triggering constraints, (2) infer root causes of vulnerabilities, (3) formulate plausible ways to solve the constraints, (4) synthesize the solver as a property-based fuzzer, and (5) leverage runtime evidence to diagnose failures and refine hypotheses. The **MCP tool layer** provides stateless tools to assist the agent’s reasoning, including call graph analysis, corpus analysis, deviation detection, and a generic property-based fuzzing framework with debugging feedback collection. The **memory layer** maintains evolving hypothesis state in a persistent markdown file (Figure 9.2 in Appendix) that preserves long-term project context across iterative workflow phases. It prevents memory drift during extended reasoning tasks and avoids re-exploring previously invalidated hypotheses.

The architecture of PBFuzz combines the workflow pattern and the agent pattern [32, 55]. Purely workflow-driven systems lack adaptability for unforeseeable vulnerability patterns and constraint discovery [55]. Conversely, unconstrained agents risk unbounded computational costs and cascading reasoning errors from hallucinations [79]. Our hybrid approach constrains agent autonomy within a workflow-defined state machine and corresponding state

invariants, enabling long-horizon reasoning while maintaining control. We elaborate on each workflow phase and state enforcement mechanisms in subsequent sections.

4.1.3 Workflow. The workflow starts with a system prompt (Figure 9 in Appendix) that initializes the agent with task objectives (e.g., target vulnerability to trigger), program context (e.g., how to test the target, signals for reaching and triggering), and configuration details (Figure 11 in Appendix). In each iteration, the **PLAN** phase analyzes vulnerability sites and their corresponding dependencies to formulate hypotheses about reachability and triggering constraints. The **IMPLEMENT** phase synthesizes parameterized input generators that encode program semantic-level constraints into input-level typed parameter spaces. The **EXECUTE** phase performs two-stage PoV search with property-based fuzzing. The fuzzer first evaluates a set of concrete parameters that represent plausible solutions the agent believes will satisfy the constraints. Upon failure, the fuzzer then systematically explores the constrained input spaces via random sampling. Upon discovering a PoV, the workflow terminates; otherwise, the **REFLECT** phase diagnoses failures (e.g., path divergence) and refines hypotheses for the next PLAN iteration.

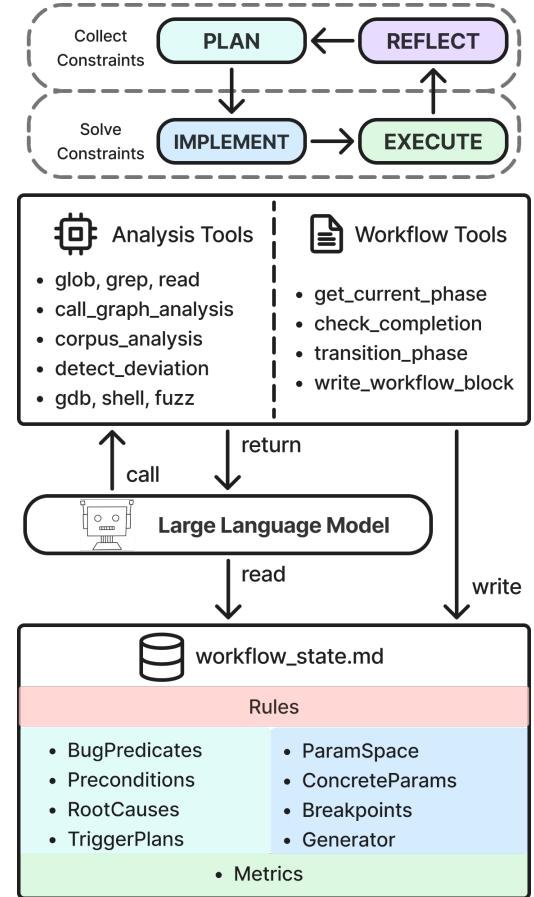


Figure 2: PBFuzz Architecture Overview.

4.2 Workflow Management

The workflow management layer enforces separation of concerns between agent reasoning and state persistence through a gatekeeper architecture. Rather than allowing direct file manipulation, three MCP tools mediate all state updates proposed by the agent:

- `write_workflow_block(target_block: string, content_json: object)`: validates syntax (JSON schema) and phase-specific write permissions. On validation failure, it rejects modifications and provides feedback to the agent.
- `transition_phase(next_phase: string)`: validates completion criteria for the current phase and enforces legal transitions according to the workflow automaton. On validation failure, it returns gating errors to prevent premature advancement.
- `check_phase_completion()`: queries prerequisites for the current phase transition, automatically detecting the phase from the workflow state.

The gatekeeper architecture prevents hallucinations from derailing the workflow by enforcing three invariants: (1) phase-specific write permissions ensure each phase exclusively modifies its target artifacts, (2) the read-only REFLECT phase prevents hypothesis revision before diagnostic analysis completes, and (3) monotonic evolution prevents redundant exploration of invalidated constraint combinations.

4.3 PLAN Phase

As the first workflow phase, the LLM agent performs constraint inference through autonomous dependency analysis to formulate hypotheses about triggering the target vulnerability. If this is not the first iteration, the agent leverages the failure diagnosis results from the previous REFLECT phase to refine its hypotheses from the last PLAN phase. In our prototype, we use the Cursor CLI agent as the base agent, which has built-in code exploration capabilities to perform backward slicing to extract dependencies. We further provide the agent with three MCP tools:

- `get_callers/get_callees(function_name: string)`: retrieves pre-computed call graph nodes. The call graph is built via a static analysis that resolves indirect calls.
- `get_reaching_routes() -> {routes: list[callstack, testcase_file]}`: retrieves callstacks and associated test-cases that successfully reach target locations from an input corpus.
- `get_corpus_status()`: reports processing progress of the current corpus before calling `get_reaching_routes`.

The constraint inference workflow proceeds through four structured updates to persistent memory in the markdown file. These constraints form a constraint hierarchy. First, the agent extracts *vulnerability triggering predicates* as safety properties and stores them in the **BugPredicates** block, one per disjunctive branch. For CVE-2017-9047 in libxml2 at valid.c:1342, its BugPredicates block records:

```
[{
  "id": "BP1",
  "location": "valid.c:1342",
  "bug_condition": "(size - len - xmlStrlen(content->prefix)) <
                    (xmlStrlen(content->name) + 10)"
}]
```

As discussed earlier, we assume the vulnerability triggering predicates, expressed as formulae of local program variables, are provided beforehand (Figure 11). Therefore, the agent only needs to extract and formalize them here. When evaluating on the Magma benchmark [33], we used the manually specified bug predicates from MAGMA_LOG statements.

Second, starting from BugPredicates, the agent performs autonomous backward program slicing up to the entry points, to infer the *reachability constraints*. The hypothesized constraints are stored in the **Preconditions** block. Four key tools are used to assist the agent's understanding and reasoning: the built-in Grep and Glob tools to search the codebase, the built-in Read tool to read source code files, and the call graph MCP tool to resolve indirect call edges. For CVE-2017-9047 in libxml2, its Preconditions block records five reachability constraints:

```
[{"id": "R1", "statement": "Must reach
    xmlNsprintElementContent function",
  "input_constraints": ["Input must be valid XML", "Must
    trigger DTD validation"]},
 {"id": "R2", "statement": "content->type must be
    XML_ELEMENT_CONTENT_ELEMENT",
  "input_constraints": ["DTD must contain ELEMENT
    declarations"]},
 {"id": "R3", "statement": "content->prefix must not be NULL",
  "input_constraints": ["Element must have namespace prefix in
    DTD"]},
 {"id": "R4", "statement": "Buffer size check at line 1302 must
  pass",
  "input_constraints": ["Initial buffer must have at least 50
    bytes"]},
 {"id": "R5", "statement": "Buffer size check at line 1326 must
  pass",
  "input_constraints": ["In unfixed version, this check is
    disabled"]}]
```

Note that compared to traditional static backward slicing, which can easily produce an overwhelming number of constraints, our agentic approach generates a substantially more concise set of semantic constraints, owing to the LLM's code understanding and reasoning capabilities.

Third, the agent examines triggering constraints from BugPredicates to identify underlying vulnerability categories (buffer overflow, integer overflow, etc.), and stores them in the **RootCauses** block. Due to uncertainty, the agent may hypothesize multiple root causes. Each RootCause also links to the related Precondition identifiers, establishing which reachability constraints enable specific vulnerability triggers. The RootCauses block of CVE-2017-9047 in libxml2 identifies it as a buffer overflow:

```
[{
  "id": "RC1", "category": "buffer_overflow",
  "description": "Disabled bounds checks allow strcat overflow",
  "related_precondition_ids": ["R2", "R3", "R4"]
}]
```

The purpose of the RootCauses block is to help the agent define boundaries of the parameter spaces during the IMPLEMENT phase, such that the generated inputs are more likely to trigger the vulnerability.

Finally, the agent reasons about RootCauses and Preconditions to formulate concrete and plausible vulnerability triggering strategies,

and stores the plans in the `TriggerPlans` block. Each `TriggerPlan` references both reachability and triggering constraint identifiers. It also contains a complexity score (1–10) based on constraint hierarchy difficulty, and maintains execution status (`pending`, `in_progress`, `completed`, `failed`) for future refinement. For CVE-2017-9047, the plan is to create deeply nested DTD content models with qualified names such that it consumes the 5000-byte buffer through recursive calls. The corresponding `TriggerPlans` block records:

```
[{
  "id": "TP1", "precondition_ids": ["R1", "R2", "R3", "R4"], "description": "Create DTD with nested element content using qualified names", "complexity": 3, "status": "pending"
}]
```

Note that the agent can generate multiple `TriggerPlans`. Each plan targets a different combination of (1) a subset of `Preconditions` and (2) a single `RootCause`. During the `EXECUTE` phase, our PBT fuzzer can explore several different `TriggerPlans` at the same time to improve the efficiency of PoV generation.

This constraint inference pipeline mimics human experts. Starting with the target vulnerability, the agent (1) examines the source code to understand the target code and to extract dependencies; (2) uses dynamic information from corpus analysis to validate reachability hypotheses (explained later); and (3) employs call graph analysis to identify reaching execution paths. Finally, the agent formulates triggering plans based on its understanding of the vulnerability and the PUT.

As mentioned earlier, if this is not the first PLAN iteration, the agent leverages the failure diagnosis results from the previous REFLECT phase to refine its hypotheses, which may involve: (1) updating `Preconditions` to include identified missing constraints; (2) revising `RootCauses` to correct inaccurate understandings of the vulnerabilities; and (3) abandoning `TriggerPlans` that are unlikely to succeed.

4.4 IMPLEMENT Phase

Building on the PLAN phase’s constraint hierarchy, in the IMPLEMENT phase, the agent reasons about how to translate the high-level semantic constraints from the `TriggerPlans` into input-level parameter spaces. Then, the agent synthesizes a custom solver (i.e., the PBT fuzzer) to search for a PoV input that satisfies the identified reaching and triggering constraints. We adopt this strategy instead of directly using LLMs to solve constraints and generate inputs because recent research demonstrates that LLMs are ineffective at solving complex arithmetic constraints [37], and generating binary inputs [77]. Moreover, invoking LLMs for input generation is slow and costly. Note that during the PLAN phase, the agent may generate multiple `TriggerPlans`. In this phase, we ask the agent to generate unified parameter spaces and a single input generator that can cover all `TriggerPlans`.

In this phase, the agent uses two primary MCP tools to assist in synthesizing the input generator:

- `extract_parameters(extractor_code: string) -> {parameter_space: dict}`: analyzes known reaching testcases from the corpus to derive typed parameter specifications by applying custom extractor functions.

- `get_generator_api_doc(topic?: string) -> {documentation: string}`: provides detailed API documentation and examples to help the agent synthesize the expected input generator.

Our custom solver consists of two key artifacts to realize a custom PBT fuzzer: (1) a Python-based parameterized input generator, and (2) a configuration file defining the parameter spaces, concrete parameter sets, and breakpoint definitions. These artifacts are generated in four steps.

First, the agent materializes input constraints from `TriggerPlans` into a typed parameter space specification, and stores it in the `ParameterSpace` section of the configuration file. Each parameter is defined as a key-value pair, where the keys are parameter space identifiers used by the input generator, and the values define the parameter types (`int_range`, `float_range`, `categorical`, `segments`, `base_seed`) and value domains (e.g., min/max ranges for numerical types, allowed values for categorical types). In this step, the agent may invoke the `extract_parameters` tool to retrieve parameter domains from known reaching testcases from the corpus to help reason about value ranges, categorical enumerations, and boundary values that may satisfy the constraint hierarchy extracted in the PLAN phase. For CVE-2017-9047, its `ParameterSpace` section records:

```
{
  "element_prefix_length": {"type": "int_range", "min": 1, "max": 200},
  "element_name_length": {"type": "int_range", "min": 1, "max": 200},
  "nesting_depth": {"type": "int_range", "min": 1, "max": 50},
  "num_elements": {"type": "int_range", "min": 1, "max": 100},
  "content_model_type": {"type": "categorical",
    "values": ["SEQ", "OR", "MIXED"]}
}
```

Second, the agent synthesizes a parameterized input generator as a Python function implementing the `generate(**params) -> bytes` interface, where `params` is a concrete set of parameter values from the parameter spaces defined in `ParameterSpace`. In other words, the generator maps a potential solution from the parameter space to a concrete test input that can be consumed by the PUT. To reduce hallucinations, we provide the `get_generator_api_doc` MCP tool to retrieve reference documentation and code examples. Although we allow the agent to use any Python libraries to help with input generation, in our evaluation, we observed that the agent mainly relies on its own knowledge of input formats to craft the input. For CVE-2017-9047, the generator constructs XML files with recursive DTD models matching `xmlSprintfElementContent`’s control flow:

```
def generate(**params) -> bytes:
    element_name_length = params.get("element_name_length", 50)
    nesting_depth = params.get("nesting_depth", 5)
    num_elements = params.get("num_elements", 5)
    element_prefix_length = params.get("element_prefix_length", 50)
    content_model_type = params.get("content_model_type", "SEQ")

    def build_content_model(depth, num_elems, model_type):
        if depth <= 0:
            return f'{prefix}:{name}' # Base case: qualified element name
        sep = ' ', ' if model_type == "SEQ" else " | '
        elements = []
        for i in range(num_elems):
            elements.append(build_content_model(depth-1, 1, model_type))
        return sep.join(elements)

    prefix = ' ' * element_prefix_length
    name = ' ' * element_name_length
    content = build_content_model(nesting_depth, num_elements, content_model_type)
    return f'<?xml version="1.0"?>\n<!DOCTYPE root [{content}]>\n<root>\n{prefix}{name}\n</root>'
```

```

        for i in range(min(num_elems, 10)):
            if depth > 1 and random.random() > 0.5:
                # Recursive nesting
                nested = build_content_model(depth - 1,
                    num_elems // 2, model_type)
                elements.append(f"({nested})")
            else:
                elements.append(qualified_name)
        return sep.join(elements)

    content_model = build_content_model(nesting_depth,
        num_elements, content_model_type)
    xml_content = f'<!DOCTYPE root [<!ELEMENT root
        ({content_model})]><root></root>'
    return xml_content.encode()

```

While it is possible to find a PoV input by randomly sampling from the parameter spaces, or brute-force enumerating all parameters, the LLM agent may have a better intuition about which parameter combinations are more likely to satisfy the constraints. Thus, to improve search efficiency, in the third step, the agent generates 5–10 concrete parameter sets that it believes are plausible solutions, and stores them in the **ConcreteParameters** section of the configuration file.

Here, the agent may leverage the root cause identified in the PLAN phase to prioritize malformed or boundary inputs over fully valid ones. For CVE-2017-9047, the agent includes seven parameter sets from shallow (depth=1, names=150) to balanced (depth=10, names=70). Among them, Test A6 records:

```

[{
    "plan_description": "Test A6: Balanced approach",
    "element_name_length": 70,
    "element_prefix_length": 70,
    "nesting_depth": 10,
    "num_elements": 10,
    "content_model_type": "SEQ"
}, ...]

```

Because the agent can make mistakes (e.g., constraints inferred in the PLAN phase could be incomplete or inaccurate; the parameter space specifications may contain mistakes), in the fourth step, we ask the agent to specify debugging breakpoints to collect actual runtime evidence during test execution. This evidence serves as the ground truth for validating the agent’s hypotheses, and for refining constraints in the REFLECT phase if PoV generation fails. The debugger breakpoints are stored in the **Breakpoints** section of the configuration file, where each entry captures a set of runtime states through inline expressions. For example, the CVE-2017-9047 breakpoint at line 1342 captures size, len, prefix/name lengths, and size - len - xmlStrlen(prefix). The corresponding Breakpoints block records:

```

[{"location": "valid.c:1342",
    "inline_expr": ["size", "len", "xmlStrlen(content->prefix)",
        "size - len - xmlStrlen(content->prefix)"],
    "hit_limit": 5},
 {"location": "valid.c:1331",
    "inline_expr": ["size", "len", "content->prefix", "buf"],
    "hit_limit": 3},
 {"location": "valid.c:5558",
    "inline_expr": ["cont", "cont->type", "cont->prefix"],
    "hit_limit": 2}]

```

In summary, the IMPLEMENT phase translates the high-level semantic constraint hierarchy from the PLAN phase (Preconditions, RootCauses, TriggerPlans) into executable input-space artifacts (ParameterSpace, generate, ConcreteParameters, Breakpoints) that guide the EXECUTE phase to search for a PoV input.

4.5 EXECUTE Phase

With executable artifacts ready, in the EXECUTE phase, the agent performs a single task—invoking the fuzzing MCP tool to search for PoV inputs.

- `fuzz(setting:dict) -> {results, metrics}`: executes property-based fuzzing using computational artifacts from the IMPLEMENT phase, combining Stage 1 targeted testing with Stage 2 property-based exploration.

Recall that this is the key step to improve the efficiency of PoV generation, i.e., instead of invoking LLMs to enumerate concrete input parameters/constraints, which is slow and expensive, we leverage property-based testing (PBT) to perform high-throughput search over the parameter spaces.

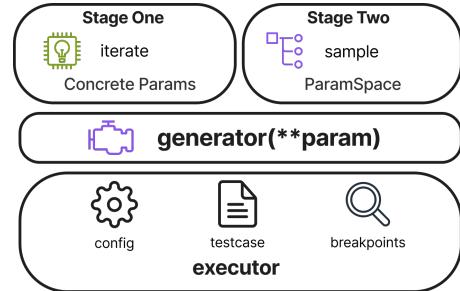


Figure 3: Property-based fuzzing workflow with two-stage test generation: Stage 1 iterates concrete parameters, Stage 2 performs heuristic parameter sampling.

The fuzz MCP server performs a two-stage search for a PoV (Figure 3). First, it evaluates the preferred input parameters from the **ConcreteParameters** section. If no PoV is found, the fuzzer performs property-based testing by sampling from the **ParameterSpace** defined in the configuration file. During execution, the MCP tool also employs the debugger for inspection, based on the guidance from the **Breakpoints** section. Due to performance overhead, the inspection is only enabled during Stage 1 targeted testing.

Stage 1 performs targeted testing by exploring the **ConcreteParameters** entries sequentially. Each **ConcreteParameters** entry is designed to satisfy specific parameter combinations from **TriggerPlans**, directly testing hypotheses about reachability and triggering conditions. The fuzzer invokes the parameterized generator synthesized in the IMPLEMENT phase with these concrete parameters, producing test inputs (as sequences of bytes) that target particular precondition and root cause identifiers. Debugger integration enables runtime state inspection at Breakpoints locations specified in the IMPLEMENT phase. When breakpoints are triggered during execution, inline expressions capture variable values and program state, providing evidence of constraint satisfaction or violation. This instrumented execution validates whether the hypothesized constraints accurately characterize the vulnerability trigger conditions.

If Stage 1 fails to generate a PoV, Stage 2 performs property-based testing [22] through systematic parameter space sampling. The property-based fuzzer samples from ParameterSpace, which encodes the preconditions and safety properties, and invokes the parameterized generator to produce diverse test inputs. Compared to traditional fuzzing, we believe PBT is well-suited for this task because (1) by sampling parameters that can satisfy the (reaching) preconditions, the generated inputs are more likely to reach the target vulnerability; and (2) by exploring constraint boundaries and edge cases in the parameter spaces, the generated inputs are also more likely to satisfy the (triggering) vulnerability conditions. For integer and float ranges, the fuzzer applies heuristic sampling strategies that prioritize boundary values, overflow-prone extremes, and other values likely to trigger vulnerabilities while respecting type constraints. For base_seed parameters, the fuzzer applies mutations to reaching testcases from corpus analysis. Stage 2 continues until a PoV is discovered or the iteration threshold is reached.

Upon discovering a PoV, the workflow terminates with success. For CVE-2017-9047, the fuzzer discovers triggering inputs at iteration 156 with parameters depth=10, prefix=name=70, reaching the target in 12.2% of executions (19/156) and triggering the vulnerability in 26.3% of reached cases (5/19). The **Metrics** block records execution statistics:

```
{
  "total_iterations": 156,
  "total_reached_count": 19,
  "triggered_count": 5,
  "timeout_count": 20,
  "error_count": 1
}
```

If no PoV is discovered after completing both stages, the agent updates the **Metrics** in the workflow state and transitions to the REFLECT phase. The accumulated execution evidence—including breakpoint evaluation results, pattern matching outcomes, and statistical metrics—persists across phase boundaries, enabling iterative constraint refinement in subsequent workflow cycles.

4.6 REFLECT Phase

If PoV discovery fails, it typically means (1) the inferred constraints from the PLAN phase are insufficient or inaccurate to reach and trigger the vulnerability, and/or (2) the parameter spaces from the IMPLEMENT phase do not adequately cover the solution space. Thus, in the REFLECT phase, the agent leverages the evidence collected from the previous EXECUTE phase to perform failure diagnosis and refine the hypotheses for the next PLAN iteration. The agent uses two primary MCP tools in this phase:

- `detect_deviation(input_file_path: string)`: identifies violated reachability constraints via execution trace analysis.
- `launch_gdb(cmd: list[string], timeout: string)`: enables interactive debugging for program state inspection.

The failure diagnosis distinguishes between reachability failures and triggering failures, based on the feedback signals from the instrumented logging statements in the PUT (e.g., MAGMA_LOG). For testcases that fail to reach vulnerability sites, the agent invokes `detect_deviation` to identify where execution diverges prematurely

before reaching the target. We first employ static analysis to pre-compute critical locations—program points at the boundary between target-reachable and unreachable regions, e.g., a conditional branch where one branch can reach the target, but the other cannot. When the agent invokes the `detect_deviation` tool with the failed testcase, the tool sets debugger breakpoints at these critical locations, and executes the testcase to (1) determine the exact divergence point, and (2) pinpoint which Precondition was violated. The tool then returns the callstack at the deviation point to the agent. Using the returned information, the agent examines branch conditions at deviation points to determine which input constraints must be refined to achieve target reachability.

For testcases that successfully reach vulnerability sites but fail to trigger the vulnerability condition, the agent performs backward dependency analysis from the bug predicate. This analysis traces data flow dependencies to identify which program variables influence the triggering condition and determines why their runtime values failed to satisfy the vulnerability predicate. The agent may invoke the interactive debugging MCP tool to inspect variable values at critical program points and validate hypotheses about triggering constraints. Internally, `launch_gdb` spawns a GDB session via named pipes in a background process, enabling asynchronous bidirectional communication for command execution and result inspection. Both analysis strategies accumulate diagnostic evidence that persists to the workflow state, documenting which constraint hypotheses were invalidated and what alternative constraints should be explored.

After diagnosis, the agent returns to the PLAN phase with refined constraints. Here we do not ask the agent to update the workflow state to record the diagnosis results; instead, the results are passed to the next PLAN phase via the agent’s short-term memory (i.e., the current conversation). As discussed earlier, in the next PLAN phase, the agent leverages the diagnosis results to refine its previous hypotheses by updating Preconditions, RootCauses, and TriggerPlans. This cycle continues until a PoV is discovered or resource limits are exhausted. Persistent memory ensures monotonic evolution, preventing redundant exploration of invalidated constraints.

4.7 Implementation Details

We now detail the concrete realization of our architecture across three infrastructure components.

Agent Infrastructure. We implement the agent layer using `cursor-cli`, which provides built-in semantic search and grep tools for efficient codebase navigation. The system invokes the agent via a shell interface, passing system prompts through `stdin`.

MCP Tool Implementation. We implement five MCP servers with 6,256 lines of Python code. They communicate with `cursor-cli` via JSON-RPC over standard I/O. The fuzzer tool dynamically loads and executes Python generator code in an isolated namespace at runtime.

Static Analysis. Our static analysis tool (Figure 4) compiles the target project using Clang with link-time optimization (LTO) enabled. This allows us to reuse existing build systems to compile source code to LLVM bitcode. We then perform whole-program static analysis based on the pre-opt bitcode dumped by LLD during linking. We use a type-based call graph analysis [44] to pre-compute call graphs (599 lines of C++). Although imprecise, this analysis is

much faster, and we believe LLM agents can overcome the imprecision through source code reasoning. To identify critical locations for deviation detection, we implement a backward breadth-first search (921 lines of C++) to mark (context-insensitive) basic blocks that (1) can reach the target locations, and (2) can reach known exit points (e.g., `exit()` syscall, return of `main()`). We perform the static analysis before the workflow starts, and save the results to files. MCP tools `get_callers/get callees` and `detect_deviation` directly query these pre-computed results, enabling efficient reachability constraint collection during the PLAN phase and failure diagnosis during the REFLECT phase.

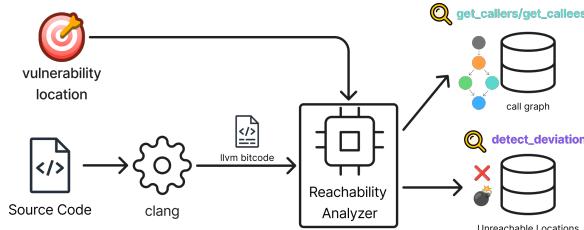


Figure 4: Static analysis workflow for call graph analysis and deviation detection.

5 Evaluation

We evaluated PBFuzz to determine whether the agentic fuzzing architecture enabled effective PoV generation through iterative constraint inference and solving. Our evaluation addressed two hypotheses: (H1) agents can successfully infer reachability and triggering constraints, with the help of MCP tools, and map these constraints to input parameter spaces; and (H2) property-based testing can solve extracted constraints efficiently. We validated our hypotheses by systematically investigating the following research questions:

- RQ1. Overall Effectiveness and Efficiency:** Can our agentic approach successfully generate proof-of-vulnerability (PoV) inputs against CVEs in real-world benchmarks efficiently, especially those that challenge existing techniques like directed fuzzers?
- RQ2. Strengths:** What unique strengths does our approach provide over existing PoV-generation techniques that enabled it to trigger more vulnerabilities?
- RQ3. Limitations:** What are the limitations of our approach, and how did they affect PoV generation?
- RQ4. Ablation Study:** Which components (MCP tools, LLM models) contributed to system effectiveness?

5.1 Experimental Setup

5.1.1 Benchmarks. We evaluated PBFuzz on the Magma benchmark [33], a ground-truth fuzzing suite with real-world CVE vulnerabilities. Each target includes instrumentation via `magma_log` function calls that distinguish vulnerability code execution from triggering. The `magma_log` function emits diagnostic messages: “MAGMA: Bug X reached” indicates vulnerable code execution, while “MAGMA: Bug X triggered” indicates condition satisfaction. This separation enables precise measurement of PoV effectiveness. We evaluated 129 CVEs and 9 security bugs across 9 projects:

libpng, libtiff, libxml2, libsndfile, poppler, openssl, php, lua, and sqlite3.

5.1.2 Baselines. We compared PBFuzz against three categories of baselines representing SOTA in PoV generation:

- **Directed Greybox Fuzzers:** AFLGo [13] (commit fa125da), the first directed fuzzer with recent updates; SelectFuzz [45] (docker 11ff923), SOTA open-source directed fuzzer that considers both control- and data-flow dependencies for target reachability.
- **Coverage-Guided Fuzzer:** AFL++ [25] (release 4.32c) with `cmpLog` enabled and disabled, respectively.
- **LLM-Assisted Fuzzer:** G2Fuzz [77] (commit f62cc55), SOTA chatbot-style LLM-assisted fuzzer.

Since Magma is a standard benchmark, and numerous directed fuzzers do not release source code or artifacts for reproduction, in RQ1, we also include results from nine recent fuzzers published between 2022 and 2025 that were evaluated on Magma, as reported in their respective papers. These fuzzers are: MC2 [57], Titan [35], Llamarfuzz [76], AFLRun [53], SeedMind [61] augmented AFL++, LibAFLGo [27], Lysos [10], Locus [78] augmented AFL++, CSFuzz [19]. These fuzzers all use a 24-hour timeout across 10 trials, except for Llamarfuzz which employs a *one-month* execution period.

5.1.3 Experimental Setup. Experiments ran in containerized environments hosted on a server with 2x Intel(R) Xeon(R) CPU E5-2695 v4 processors (36 cores) and 512 GB RAM. Each fuzzing instance executed within a Docker container limited to one CPU core and 4 GB RAM for consistency.

All baseline greybox fuzzers based on AFL(++) were configured with non-deterministic mode enabled. Baseline fuzzers ran for 24 hours per target using Magma’s built-in initial seeds for all targets. We repeated baseline experiments for 10 trials to capture randomness.

PBFuzz was run with *a single round* per target due to budget constraints from external LLM API calls. LLM-based methods are constrained by API quota rather than compute. PBFuzz used a 30-minute timeout per target for constraint refinement. PBFuzz used Cursor CLI (version 2025.11.25-d5b3271) configured with Claude Sonnet-4.5 as the default agent model. Model-variant effects are discussed in subsequent sections. G2Fuzz was configured to use the GPT-5 API, with G2Fuzz-generated initial seeds only.

Each project in Magma contains 1 to 5 fuzz harnesses targeting different vulnerability sites. This yields 361 total targets across all projects. Static analysis cost varies significantly across projects. For libpng, libtiff, lua, libsndfile, and sqlite3, analysis completes within 10 seconds per target. Poppler and PHP targets require 10 to 20 minutes each. OpenSSL targets require 1 to 5 hours with several timing out. To maintain experimental feasibility, we disable program analysis MCP tools for OpenSSL targets that exceeded the analysis time budget. Our reachability static analysis identifies 83 targets as unreachable in the whole-program call graph. We evaluate PBFuzz on the remaining 278 reachable targets.

Complete experimental scripts are published at <https://github.com/R-Fuzz/magma> to ensure full reproducibility of our evaluation results.

5.1.4 Metrics. We measure PoV generation systems with two primary metrics:

- **CVE coverage** quantifies the number of Magma vulnerabilities triggered. This number is the union across all trials for baseline fuzzers, while PBFUZZ was only run for a single trial. This is the effectiveness metric used in RQ1. By comparing vulnerabilities that can only be triggered by a system, we can also understand the unique strengths of different approaches.
- **Time-to-exposure (TTE)** records elapsed time until first PoV generation. This metric captures the efficiency of a PoV generation approach. For vulnerabilities triggerable from multiple fuzzing harnesses, and were triggered multiple times across trials, we report the optimal (minimum) TTE.

In addition, we report some auxiliary metrics: Time-to-reach (TTR) records elapsed time until first input reaches the vulnerable code location. This is a metric the Magma benchmark reports, but we found it less informative than TTE as 60 vulnerabilities are reachable with the initial seed inputs.

For efficiency analysis of PBFUZZ, we also measured (1) the iteration count of PLAN-IMPLEMENT-EXECUTE-REFLECT cycles until PoV discovery, and (2) LLM API costs in total tokens consumed across all workflow phases. We supplemented quantitative metrics with qualitative case studies analyzing collected constraint hierarchies and failure modes.

5.2 RQ1. Overall Effectiveness and Efficiency

5.2.1 CVE Coverage. Figure 5 summarizes PoV generation effectiveness across all evaluated fuzzers on Magma, where we take the union of CVEs triggered across all trials of fuzzing campaigns. Overall, PBFUZZ triggered 57 vulnerabilities across Magma and outperformed all baseline fuzzers (first row). Upon inspection, we found that 6 vulnerabilities uniquely triggered by PBFUZZ (PHP001, PHP003, PHP010, PHP013, PHP014, SSL006) were due to the use of different harnesses (more details in §5.3). However, even after excluding these 6 harness-specific vulnerabilities, PBFUZZ still outperformed the second-best fuzzer (AFL++ with CmpLog). It also outperformed all directed greybox fuzzers and LLM-assisted fuzzers. Notably, Llamafuzz executed for a month, while other baseline fuzzers were run for 10 trials of 24 hours each. In contrast, PBFUZZ only ran for 30 minutes in a single trial.

5.2.2 Unique CVE Coverage. The second row of Figure 5 shows the number of vulnerabilities that were uniquely triggered by PBFUZZ compared to each baseline; and the third row shows the number of vulnerabilities uniquely triggered by each baseline compared to PBFUZZ. Overall, PBFUZZ consistently triggers more unique vulnerabilities than any baseline. PBFUZZ triggered 17 vulnerabilities that none of the other fuzzers could trigger: LUA001, PDF002, PDF005, PDF008, PDF009, PDF012, PDF022, SND016, SQL007, SQL010, XML010 with the same harnesses as other fuzzers; and the aforementioned 6 harness-specific vulnerabilities. Among different fuzzers, PBFUZZ uniquely triggered 19 vulnerabilities over AFL++ with CmpLog, and 18 over AFLGo; while AFL++-cmplog uniquely triggered 11 vulnerabilities over PBFUZZ, and AFLGo uniquely triggered 9 vulnerabilities over PBFUZZ. This result shows that PBFUZZ provides unique strengths in PoV generation capabilities that complement

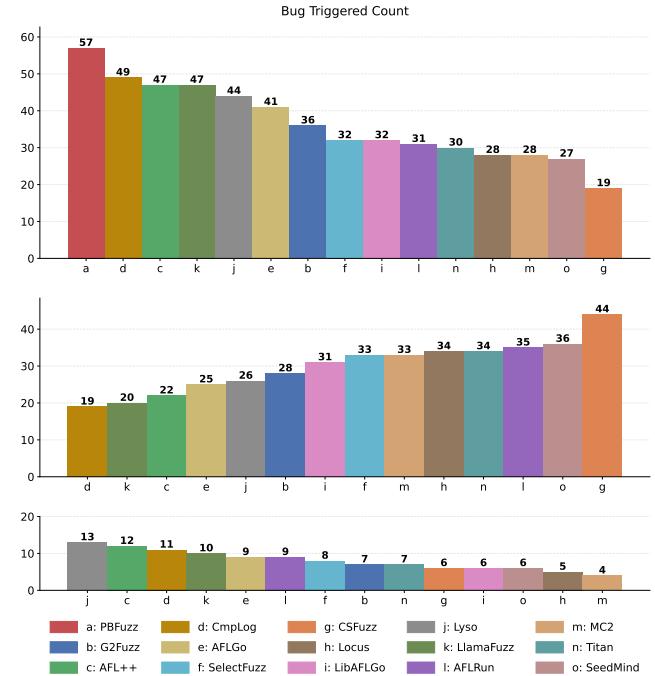


Figure 5: Effectiveness across Magma benchmarks ordered by performance. Top: total unique vulnerabilities triggered. Middle: number of vulnerabilities exclusively triggered by PBFUZZ compared to other fuzzers. Bottom: compared to PBFUZZ, number of vulnerabilities only discovered by baselines.

existing fuzzing techniques. We provide in-depth analysis of how PBFUZZ triggered these unique vulnerabilities in RQ2 and RQ3.

5.2.3 Efficiency. Table 2 details the Time-to-Exposure (TTE) statistics of different approaches. PBFUZZ achieves minimum 83s, median 339s, and maximum 1441s TTE. In comparison, AFL++ with CmpLog requires minimum 5s, median 8680s, and maximum 85322s. AFLGo exhibits minimum 3s, median 6170s, and maximum 85200s TTE. SelectFuzz demonstrates minimum 2s, median 3397s, and maximum 80394s TTE. G2Fuzz demonstrates minimum 30s, median 5400s, and maximum 68400s TTE. Evidently, PBFUZZ not only triggered more vulnerabilities and more unique vulnerabilities, but more critically, PBFUZZ achieves this in a single trial within a 30-minute budget. This result highlights that despite the involvement of LLMs, our novel workflow of constraint inference (PLAN), constraint encoding (IMPLEMENT), and constraint solving (EXECUTE) enables substantially more efficient PoV generation than conventional approaches, including LLM-assisted fuzzers like G2Fuzz.

This performance gap aligns with the theoretical “efficiency boundary” of random testing [11, 12], where the discovery rate of greybox fuzzers decays exponentially. Consequently, simply extending the time budget—as seen with Llamafuzz’s one-month campaign—yields diminishing returns. PBFUZZ overcomes this by adopting a white-box approach where the LLM agent, with fine-grained feedback, performs autonomous semantic reasoning to trim the search space and prioritize plausible solutions. The adoption of

property-based testing further enables efficient exploration of the reduced input space.

Table 3 presents the Time-to-Reach (TTR) for different fuzzers. As mentioned earlier, some vulnerabilities are reachable by the provided initial seeds, therefore we excluded those vulnerabilities from the comparison. The results demonstrate that PBFuzz reaches vulnerabilities significantly faster than other fuzzers on most benchmarks (only 2 are slower and 2 were never reached). Compared to TTE (Table 2), it is evident that most vulnerabilities reached by PBFuzz are triggered shortly thereafter, except for SQL013, SQL014, TIF001, and TIF002. Other fuzzers generally require significantly more time to transition from reaching to triggering the vulnerabilities, or never managed to trigger within the time limit.

5.2.4 Consistency Analysis. We also evaluate reproducibility via consistency analysis across repeated executions. Due to cost concerns, we selected 23 vulnerabilities that all fuzzers could trigger at least once. Each system executes 10 independent runs per vulnerability.

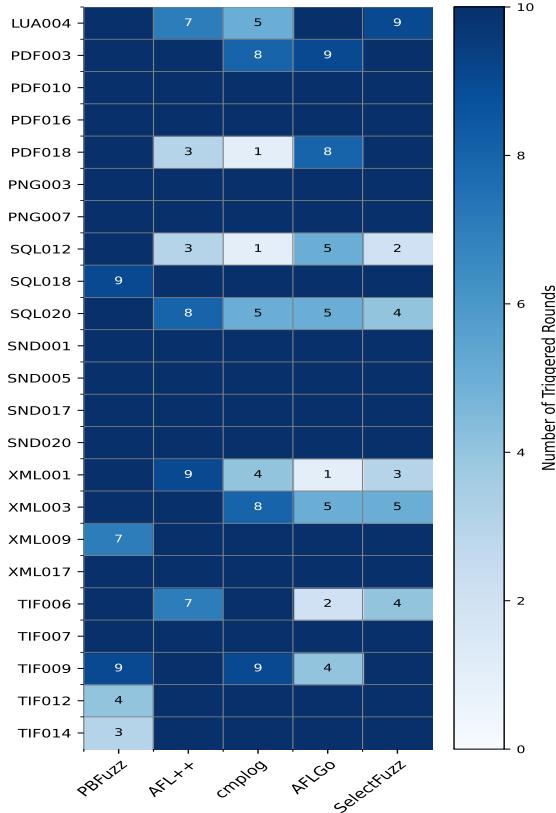


Figure 6: Reproducibility heatmap: triggering frequency across runs. Vertical axis lists vulnerabilities, horizontal axis lists fuzzers. Numeric annotations highlight instances below maximum.

Figure 6 summarizes consistency results. PBFuzz achieves superior consistency, attributed to its deterministic constraint-solving mechanism. Empirically, PBFuzz triggers 18 of 23 tested vulnerabilities consistently (10/10 runs), including SQL012 and PDF018

Table 2: Time-to-Exposure (TTE) comparison across benchmarks. TTE denotes the time required to generate a PoV for each vulnerability. T.O denotes timeout at 24 hours. NA denotes not applicable, when the fuzzer cannot handle the target programs. We excluded vulnerabilities that cannot be triggered by all tested fuzzers.

Bug ID	PBFuzz	G2Fuzz	AFL++	cmplg	AFLGo	SelectFuzz
LUA001	2.1m	T.O	T.O	T.O	T.O	T.O
LUA002	7.2m	T.O	22.7h	23.7h	T.O	T.O
LUA003	5.5m	6h	T.O	T.O	T.O	T.O
LUA004	4m	29m	11.3h	18.1h	8.8h	4.7h
PDF002	2.9m	T.O	T.O	T.O	T.O	T.O
PDF003	5.3m	T.O	6.7h	8.7h	6.2h	5.3h
PDF005	12.4m	T.O	T.O	T.O	T.O	T.O
PDF006	T.O	17h	20.3h	T.O	T.O	T.O
PDF008	3.2m	T.O	T.O	T.O	T.O	T.O
PDF009	3.1m	T.O	T.O	T.O	T.O	T.O
PDF010	8.4m	6h	5h	5.2h	20.9m	8.6m
PDF011	T.O	T.O	18.8h	17.7h	T.O	21.8h
PDF012	5.6m	T.O	T.O	T.O	T.O	T.O
PDF016	3.2m	60s	19s	29s	69s	87s
PDF018	3.1m	31m	10.7h	22.5h	21.9m	10.6m
PDF019	22.9m	T.O	2h	6h	T.O	T.O
PDF021	8.1m	T.O	T.O	21.2h	23.5h	22.3h
PDF022	3.1m	T.O	T.O	T.O	T.O	T.O
PHP001	11m	T.O	T.O	T.O	T.O	NA
PHP003	12.3m	T.O	T.O	T.O	T.O	NA
PHP004	10m	19h	43.5m	91.5m	2.3h	NA
PHP009	15.4m	T.O	3.6m	17.7m	74m	NA
PHP010	19.6m	T.O	T.O	T.O	T.O	NA
PHP011	10.2m	8m	26s	23s	14s	NA
PHP013	6.7m	T.O	T.O	T.O	T.O	NA
PHP014	3.9m	T.O	T.O	T.O	T.O	NA
PNG001	10.8m	5h	17.4h	22.5h	T.O	T.O
PNG003	6.7m	3m	4s	5s	3s	2s
PNG006	4.5m	4m	T.O	8s	T.O	T.O
PNG007	11.9m	8h	72.2m	1.8h	3.8h	44.3m
SND001	83s	6h	18.6m	19.8m	5.3m	83s
SND005	4m	15h	71.2m	94.7m	34s	17s
SND006	T.O	11h	77.8m	99m	6.8m	4.4m
SND007	T.O	60m	2.6h	2.1h	6.7m	1.9m
SND016	6.4m	T.O	T.O	T.O	T.O	T.O
SND017	3.3m	22m	97.3m	1.7h	27.7m	60.2m
SND020	2m	28m	1.9h	2.5h	1.8h	1.7h
SND024	T.O	60m	59.2m	99.3m	6.6m	1.8m
SQL002	T.O	5h	5.1m	9.5m	55.6m	16.9m
SQL003	2.4m	T.O	T.O	23.7h	T.O	T.O
SQL007	5.7m	T.O	T.O	T.O	T.O	T.O
SQL010	5.6m	T.O	T.O	T.O	T.O	T.O
SQL012	2m	T.O	20.2h	22.5h	16.2h	21.2h
SQL013	T.O	T.O	T.O	21.9h	T.O	T.O
SQL014	T.O	T.O	47.1m	2.4h	9.1h	3.3h
SQL015	8.2m	T.O	20.6h	20.6h	T.O	T.O
SQL018	3.5m	7h	15.7m	53.8m	60.1m	34.8m
SQL020	2.1m	15h	11.5h	17.3h	17.8h	17.8h
SSL001	T.O	10h	2.7h	8.3h	10.9h	NA
SSL002	4m	25m	94s	2.4m	1.8m	NA
SSL003	7.7m	30s	86s	2.6m	1.9m	NA
SSL006	24m	T.O	T.O	T.O	T.O	NA
SSL009	T.O	T.O	14.4h	20h	15.6h	NA
SSL020	9.1m	T.O	12.6h	21.7h	20.7h	NA
TIF001	T.O	T.O	23h	22.5h	T.O	T.O
TIF002	T.O	11h	5.8h	8.9h	18.9h	21.7h
TIF005	9.7m	2h	T.O	37.1m	T.O	T.O
TIF006	1.9m	4h	15.5h	44.1m	22.3h	20.6h
TIF007	6.8m	60s	21s	8s	25s	29s
TIF008	7.6m	4h	16h	16.6h	20h	T.O
TIF009	20.2m	47m	4.7h	6.2h	16.9h	2.3h
TIF012	19.6m	34m	6m	2.6m	1.7h	53m
TIF014	12.8m	5m	13.2m	10.3m	53.9m	2.3h
XML001	3.4m	14h	14.8h	20.9h	22.1h	21.4h
XML002	1.8m	T.O	17.7h	22.9h	T.O	T.O
XML003	2.8m	T.O	10.6h	17.9h	16.1h	14.3h
XML009	7.9m	15m	78m	72.6m	11.4m	19.6m
XML010	8.5m	T.O	T.O	T.O	T.O	T.O
XML012	T.O	T.O	23.1h	T.O	19.8h	6.5h
XML017	3.4m	6m	13s	61s	6s	8s

Table 3: Time-to-Reach (TTR) comparison across benchmarks. TTR denotes the time required to reach each vulnerability. T.O denotes timeout at 24 hours. NA denotes not applicable, when the fuzzer cannot handle the target programs. We excluded vulnerabilities that are reachable by initial benchmark seeds or that cannot be reached by all tested fuzzers.

Bug ID	PBFuzz	G2Fuzz	AFL++	cmflog	AFLGo	SelectFuzz
LUA001	2.1m	T.O	T.O	19.2h	T.O	T.O
LUA002	15s	8h	5.3h	12.3h	15.1h	19.6h
LUA003	15s	4h	T.O	T.O	T.O	T.O
LUA004	17s	4h	3.6h	7.2h	8.5h	4.7h
PDF002	18s	4h	15.7h	14.4h	3.8h	17s
PDF004	T.O	30s	12.5h	17.1h	22.8h	T.O
PDF005	11s	40s	16.8h	14.4h	21.6h	17s
PDF008	54s	30s	T.O	T.O	T.O	T.O
PDF018	3.1m	9.4h	10.7h	22.5h	21.8m	10.6m
PHP001	11m	9s	T.O	T.O	T.O	NA
PHP010	19.6m	T.O	T.O	T.O	T.O	NA
PHP013	6.7m	T.O	T.O	T.O	T.O	NA
PHP014	3.9m	T.O	T.O	T.O	T.O	NA
SND017	7s	5.2m	72.8m	53.3m	17.9m	59.1m
SND020	2m	22m	82m	66.7m	80.4m	99.1m
SQL003	2.4m	30s	15.2h	20.7h	19.8h	T.O
SQL006	2.2m	3m	11.2h	9.9h	13h	14h
SQL009	3.7m	19h	3.9m	7.1m	23.3m	12.1m
SQL011	T.O	9h	20.9h	T.O	T.O	22.4h
SQL012	1.8m	22m	6h	10.3h	10.9h	15.6h
SQL013	11.4m	T.O	20.6h	17.7h	20.9h	T.O
SQL014	3.1m	2.1h	5.4m	16.6m	62.6m	45m
SQL020	2.1m	4h	11.5h	15.8h	17.7h	17.8h
SSL006	22.4m	T.O	T.O	T.O	T.O	NA
TIF001	10.8m	88.6m	7.9h	17.5h	23.5h	18.4h
TIF002	8.8m	5h	4.2h	2.4h	18.4h	19.8h
TIF005	9.7m	2h	T.O	37.1m	T.O	T.O
TIF006	1.9m	10s	15.5h	44.1m	22.3h	20.6h
TIF008	3.2m	10s	14.1h	13.8h	19.9h	T.O
TIF009	4m	45m	4.7h	6.2h	16.9h	2.3h
TIF010	4.9m	60s	12m	6.1m	17.9m	6.9h
XML002	1.8m	T.O	17.7h	22.9h	T.O	T.O
XML010	8.5m	T.O	T.O	T.O	T.O	T.O

where AFL++ and cmflog achieve only 3/10 and 1/10 respectively. This result further highlights the effectiveness of PBFuzz’s novel approach, which leverages agentic reasoning to directly synthesize parameterized generators. This significantly improves the likelihood that generated inputs satisfy reachability preconditions by construction. As a result, PBFuzz not only can trigger the vulnerabilities faster, but also does so more consistently across repeated runs. In contrast, as discussed in §5.3, subsequent mutations from traditional fuzzers often destroy accidentally discovered constraint structures.

5.2.5 Cost Analysis. Table 4 shows the API cost incurred by PBFuzz when generating PoVs for each vulnerability across all projects. This expenditure correlates with tool invocation frequencies (Figure 8), confirming that extensive analysis drives token consumption. On average, PBFuzz consumed 2.18 million tokens per vulnerability, incurring a cost of \$1.83 per PoV. Given the time-sensitive nature of vulnerability verification, this expenditure remains acceptable compared to the substantial computational resources required by 24-hour greybox fuzzing campaigns. Furthermore, PBFuzz demonstrated robust scalability: PHP (1192K LOC) incurred only 3x the cost of lua (23K LOC),

Table 4: API cost from Cursor per vulnerability

Project	Input Tokens	Cache Read	Output Tokens	Total Tokens	Cost (\$)
libpng	64.0k	2737k	20k	2875k	1.53
openssl	49.044k	2109k	11k	2170k	0.99
php	160.120k	7261k	32k	7455k	3.28
libsndfile	95.074k	3754k	21k	3870k	1.80
sqlite3	154.091k	4730k	29k	4915k	2.45
poppler	48.035k	1923k	18k	1989k	1.04
libxml2	164.0k	98k	27k	5214k	2.66
lua	76.054k	1474k	24k	1574k	1.09
libtiff	182.089k	6704k	57k	6943k	3.55

showing that agentic reasoning selectively retrieved relevant context rather than exhaustively processing the entire codebase.

Answer to RQ1: PBFuzz significantly outperforms SOTA proof-of-vulnerability (PoV) generation approaches on all performance metrics: it triggered more vulnerabilities, more unique vulnerabilities, and did so faster and more consistently.

5.3 RQ2. Unique Strengths

In this subsection, we perform in-depth qualitative analysis to understand PBFuzz’s unique strengths in PoV generation. Specifically, we analyze the 17 vulnerabilities that only PBFuzz can trigger, and 2 vulnerabilities that only PBFuzz can reach, to identify which capabilities of PBFuzz enabled it to overcome baseline fuzzer limitations. Our analysis reveals four strengths of PBFuzz that allowed it to overcome the constraints blocking traditional fuzzers from generating valid PoV inputs.

5.3.1 Autonomous Reachability Analysis. Among the 17 unique vulnerabilities, we found that 6 vulnerabilities cannot be reached through the harnesses chosen by Magma. For PHP001, the phar:// protocol requires the CLI entry point sapi/cli/php rather than the library fuzzer harness. For PHP003, PHP010, PHP013, PHP014, the vulnerabilities are only reachable through the php-fuzz-execute harness, but not through the provided php-fuzz-harness. For SSL006, the vulnerable function EVP_EncodeUpdate is unreachable via the provided asn1parse harness, due to library-level input sanitization (chunking).

Upon reviewing PBFuzz’s logs, we found that the LLM agent identified these reachability barriers during its autonomous call graph analysis. Once it identified the reachability barriers, for PHP vulnerabilities, PBFuzz searched the code repository for alternative entry points via symbol analysis, and successfully located the correct harnesses. It then modified test orchestration to instantiate the correct harness and successfully triggered the vulnerabilities. For SSL006, PBFuzz synthesized a custom harness to make the vulnerability triggerable by (1) directly invoking EVP_EncodeUpdate, and (2) decoupling logical argument values from physical input sizes by interpreting input bytes as parameter encodings (in1 from 8-byte header).

While utilizing alternative harnesses might appear unconventional from a benchmarking perspective, we contend that this demonstrates the distinct capability of PBFuzz. Specifically, when

evaluating on Magma, all previous work failed or was unable to reason about why some vulnerabilities are not triggered, so these issues remained hidden until now. In contrast, PBFuzz’s reasoning ability allows it to identify and fix such issues autonomously. Significantly, in real-world library testing scenarios, newly introduced features or code typically do not have a corresponding fuzz harness. Therefore, PBFuzz’s ability to autonomously identify correct harnesses or modify existing harnesses to reach vulnerable code paths will be highly advantageous. In fact, synthesizing fuzz harnesses itself is another critical research topic [9, 36, 70].

5.3.2 Property-based Testing. Two vulnerabilities (PDF005, PDF022) require maintaining input structural (syntactic) invariants across nested layers, where single-field mutations will corrupt global validity and cause the input to be rejected. Triggering these vulnerabilities requires synchronized updates: incrementing structure counts, inserting payloads, and updating all affected length fields. Baseline fuzzers failed because they cannot maintain these invariants during input mutations. On the contrary, by synthesizing parameterized generators that encode input structural dependencies as compositional functions, PBFuzz can maintain these syntactic invariants during input generation. This highlights the unique strength of our property-based testing approach.

Specifically, PDF005 targets JPEG2000 streams embedded in PDF objects. Triggering the component dimension mismatch bug requires modifying the SIZ marker’s Csiz field (component count) by inserting 3-byte component metadata tuples, and updating both the marker’s Lsiz length field and the enclosing jp2c box length. If a fuzzer increments Csiz without adding the metadata tuples, the parser reads past the marker boundary and fails. If a fuzzer inserts bytes without updating Lsiz, OpenJPEG truncates prematurely. PBFuzz was able to overcome these challenges as follows. During the PLAN phase, PBFuzz identified all the related fields and their dependencies. During the IMPLEMENT phase, it synthesized a surgical binary patcher that locates the Csiz field via pattern matching, computes the new length as `new_lsiz = lsiz + len(comp_data_to_add)`, and propagates updates to all enclosing container lengths. This allows PBFuzz to generate valid JPEG2000 streams with mismatched component counts, successfully triggering the vulnerability.

PDF022 is triggered when the /Separation color space is set to /DeviceGray with name “Black”. However, changing the color space from /DeviceRGB to /DeviceGray has cascading implications: the transformation function’s /Range array must change from 6 floats to 2 floats, and the PostScript calculator body must return 1 value instead of 3. Mutating the color space name without adjusting dependent fields causes PDF parser validation failures before reaching the vulnerable logic. For PDF022, PBFuzz implemented coherent structural transformations, updating all coupled fields (/Range, function body, stream length) atomically within generator logic, ensuring structural validity.

These case studies validate H2: the adoption of property-based testing enables PBFuzz to systematically search the solution space without breaking structural invariants.

5.3.3 Hierarchical Semantic Constraints Solving. Seven vulnerabilities (LUA001, PDF002, PDF008, PDF009, PDF012, SQL010, XML010) require

solving nested constraints coupling syntactic structures with *semantic values* and *runtime state*. These vulnerabilities are reachable by fuzzers, but solving the hierarchical constraints exceeds the capabilities of random mutation. PBFuzz resolves constraint hierarchies through hypothesis-driven exploration in PLAN and structured synthesis in IMPLEMENT. These successes validate H1 and H2: PBFuzz not only can successfully infer constraint hierarchies, but can also encode them as typed parameter spaces for effective solution space search.

Triggering LUA001 requires calling `debug.getlocal(level, nvar)` within a variadic Lua function context (syntax), where `nvar = INT_MIN (-231, semantic constraint)`. The negation operation `-nvar` overflows the value back to `INT_MIN`, bypassing the guard condition while triggering pointer arithmetic overflow. Generating a PoV thus faces three nested constraints: (1) synthesizing valid Lua (preserving function-end keywords), (2) embedding “-2147483648” in argument position correctly, and (3) establishing variadic context (defining `function(...)` before invocation). Random mutations have nearly zero probability of satisfying all three constraints simultaneously. PBFuzz successfully triggered LUA001 by analyzing `ldebug.c` to identify the negation overflow predicate; then synthesized an input generator parameterizing both `nvar` magnitude and variadic function structure to exhaustively explore the constrained space.

For PDF002 and PDF008, the vulnerabilities reside in arithmetic checks on specific dictionary fields (`Length < 0` and `VerticesPerRow <= 0`). These create coverage plateaus where valid positive integers and safe boundary values execute identical paths. Mutation-based fuzzers receive no feedback to guide them toward the specific syntax of negative ASCII integers (inserting a minus sign) or the exact value zero, while maintaining the dictionary’s syntactic integrity. PBFuzz successfully triggered these by identifying the semantic bounds via source analysis and using regex-based generation to inject negative values while preserving PDF structure.

Triggering PDF009 and PDF012 requires the `/Length` field to contain specific 19-digit ASCII strings (near 2^{63} and `LLONG_MAX`) to trigger integer overflows. Baseline fuzzers can inject the binary representations of `0x7FFFFFFFFFFFFFFF` into their interesting-value dictionary. But the PDF parser’s `getInt64()` function expects a decimal digit sequence. The semantic gap between binary representation and ASCII parsing creates a barrier that baseline fuzzers cannot overcome. For these cases, PBFuzz’s semantic understanding capability allowed it to calculate the required 19-digit numbers and generate the corresponding ASCII strings directly, bypassing the representation gap.

SQL010 is a use-after-reallocation vulnerability that only manifests when the `WhereClause` array exhausts its allocated slots, and triggers `whereClauseInsert` reallocation to invalidate cached pointers. Generating valid `(a,b) IN (SELECT...)` syntax can reach the vulnerable code in `exprAnalyze`, as baseline fuzzers did. However, when adding extra AND clauses to test inputs, fuzzers will observe neither new edge coverage nor closer CFG distance to the target. Therefore, they will discard these user inputs that actually fill the array slots, preventing themselves from satisfying the reallocation condition. PBFuzz successfully triggered SQL010 by recognizing that reachability differs from triggering. It then correctly hypothesized

array exhaustion as the triggering plan, and explicitly parameterized `num_and_conditions` to control memory pressure, allowing it to perform efficient targeted state space exploration.

Triggering XML010 requires solving nested constraints that are almost impossible for fuzzers. First, it requires loading an external DTD file via a `SYSTEM` entity with a `file:///...` URI in the input XML file. Crafting this additional file and setting up a link to it autonomously is not possible for single-stream fuzzers like AFL++. Second, it demands complex nested XML entity structures that are fragile to random mutation. Third, loading external DTD files requires specific parsing options, but the harness selects parsing options pseudo-randomly by hashing the input content. As a result, slight content changes would invalidate configuration requirements. PBFuzz overcame these by synthesizing a multi-stage generator that (1) established the external DTD environment, (2) constructed structurally valid nested entities, and (3) appended whitespace suffixes to manipulate the hash until the correct parser options were selected.

5.3.4 Semantic Constraint Solving. Triggering SQL007 and SND016 is challenging because it requires driving the execution to transition through a series of program states (schema reload, or specific parser selection). In other words, a PoV generator must be able to (1) infer the state machine, (2) reason about state machine transitions to reach the vulnerable state, and (3) generate inputs that can drive these transitions. PBFuzz successfully overcame these challenges by performing autonomous backward dependency analysis to infer the constraints, LLM-based reasoning to solve these semantic (state) constraints, and synthesizing parameterized generators to drive state transitions. This validates H1: PBFuzz can infer and solve reachability barriers that distance metrics cannot capture.

Specifically, for SQL007, the vulnerability requires parsing malicious SQL statements embedded in the `sqlite_schema` system table during schema reload. PBFuzz traced `db->init.busy` backward to identify reload operations, then constructed a multi-stage generator producing the required state sequence (`PRAGMA writable_schema=ON → INSERT → ATTACH/VACUUM`).

For SND016, the vulnerability depends on differential validation logic: WAV and AIFF parsers validate channel counts (≤ 1024), while the MAT5 parser does not. PBFuzz recognized that reachability varies across format parsers, performed differential analysis to locate the validation gap in MAT5, and targeted generation specifically to that code path.

5.3.5 Summary. Through qualitative case studies of unique vulnerabilities, the unique strengths of our agentic approach are clearly demonstrated. First, unlike traditional static analysis used by directed fuzzers, PBFuzz’s LLM-driven PLAN stage can (1) correctly identify control- and data-dependencies, (2) reason about complex program semantics (state machines, representation gaps), and (3) precisely narrow down the search space for efficient exploration. Second, unlike traditional fuzzers that have trouble mapping guidance from static analysis, PBFuzz’s novel IMPLEMENT stage enables (1) bridging the gap between semantic constraints and syntactic input structures, and (2) encoding complex constraint-solving plans as parameterized generators. Finally, unlike traditional fuzzers that rely on random mutation, which often breaks input structural

invariants, PBFuzz’s property-based testing approach (1) systematically explores solution spaces without violating constraints, and (2) enables atomic multi-field updates to maintain global validity. These unique strengths collectively empower PBFuzz to overcome the limitations of traditional fuzzers for effective and efficient PoV generation.

Answer to RQ2: PBFuzz’s superior performance stems from several key innovations: its workflow, LLM-driven semantic constraint inference, constraint-to-parameter-space encoding, and the adoption of property-based testing.

5.4 RQ3. Limitations

Compared to SOTA fuzzer AFL++ with Cmplog, PBFuzz failed to trigger 10 vulnerabilities (SSL001, SQL013, TIF001, TIF002, PDF006, XML012, SQL002, SSL009, SQL014, PDF011). In this subsection, we perform an in-depth root cause analysis over PBFuzz’s execution traces and AFL++-generated PoV inputs to understand why PBFuzz fails on these cases. Our analysis identifies three fundamental limitations in the agentic approach that collectively explain these failures. These limitations stem from fundamental tensions between LLM training objectives (generating correct, well-formed outputs) and vulnerability exploitation requirements (generating malformed, boundary-violating inputs).

5.4.1 Systemic Scope Limitation. The first limitation manifests during the PLAN phase when PBFuzz only focused on local vulnerability contexts but missed dependencies from code far away. Among failed cases, 3 vulnerabilities (XML012, PDF011, SQL013) are triggered by side effects from components the agent dismissed as irrelevant.

This scope limitation stems from context efficiency strategies in LLM reasoning. To manage cognitive load and token budgets, agents employ heuristics to prune “irrelevant” code paths, focusing on direct control-flow and data-flow dependencies. While, as discussed in the previous subsection, this strategy and ability greatly improve the efficiency of the agent’s analysis, false negatives from LLMs’ reasoning can lead to missing critical dependencies. Complex vulnerabilities often involve indirect interactions: character encoding conversions triggering buffer reallocations, annotation processing affecting memory management, or system table contents controlling optimizer behavior.

For XML012, PBFuzz analyzed entity expansion and buffer management logic, hypothesizing that explicit GROW operations would trigger reallocation. AFL++ applied encoding attribute mutation to seed XML files, testing various character encodings until EUC-JP with repeated `0xC8` bytes caused UTF-8 expansion ($174 \rightarrow 261$ bytes), triggering reallocation. Through 4,000 iterations, PBFuzz never varied the encoding attribute, treating it as orthogonal to the buffer bug.

For PDF011, the vulnerable function `XRef::getEntry` performs array access with insufficient bounds checking. PBFuzz focused on XRef table structures, analyzing seeds for object reference patterns. AFL++ applied arithmetic mutations to annotation fields in seed PDFs, escalating /Rect coordinate 123 to 8,859,022,461 through repeated increments. This value triggers integer overflow during annotation processing, propagating negative indices

to XRef::getEntry. PBFuzz never examined annotations because static call graph analysis showed no direct path from annotations to XRef; the connection only materializes during page rendering.

For SQL013, PBFuzz attempted to construct data patterns causing the skip-scan optimizer to allocate insufficient buffer space. After 2,000 iterations achieving 30 target reaches, the agent concluded the vulnerability was difficult to trigger. AFL++ applied SQL keyword substitution and system table injection, discovering that `DELETE FROM sqlite_stat1; INSERT INTO sqlite_stat1 VALUES(...)` directly controls optimizer statistics. The seed corpus contained ANALYZE commands, but PBFuzz only generated application-layer queries, never exploring system table manipulation—an attack vector outside its conception of “normal” SQL usage.

This limitation reveals a fundamental challenge in agentic program analysis: identifying long-distance dependencies, especially indirect ones. While this is a well-known hard problem for traditional static analysis too, we posit that providing LLM agents with results from more sophisticated static analysis (e.g., pointer analysis), or dynamic analysis (e.g., taint analysis), could address this challenge in the future.

5.4.2 The Constraint Search Gap. The second limitation arises in the IMPLEMENT phase, when PBFuzz has correctly identified high-level semantic constraints but fails to map them to the specific parameter space. Among failed cases, 3 vulnerabilities (SSL009, SSL001, PDF006) depend on logical conditions (e.g., “length equals zero”) that can be satisfied by multiple byte-level encodings. While PBFuzz explored standard encodings, it failed to enumerate alternative ASN.1 encoding forms, overlooked unusual tag values (e.g., 266 for `V ASN1_NEG_ENUMERATED`), and omitted integer boundary values (± 32768). By examining the seed corpus, AFL++ discovered alternative valid forms (e.g., long-form ASN.1 length) or boundary-value realizations (e.g., integer overflows) that the agent overlooked. This highlights a critical blind spot in our current design: despite having access to control-flow analysis tools (e.g., `get_reaching_routes`), the agent lacked the explicit enumeration heuristics or domain-specific knowledge needed to systematically explore the input-level parameter spaces of complex formats.

For SSL009, the vulnerability requires a zero-length OCTET STRING in ASN.1 DER encoding. PBFuzz attempted short-form encoding `04 00`, but modifying existing certificates broke structural integrity. AFL++ applied byte replacement to mutate length encoding forms: replacing `04 04` with `04 82 00 00` (long-form zero length). This 4-byte substitution maintains alignment without cascading updates. Despite analyzing seed certificates containing similar structures, PBFuzz never enumerated alternative ASN.1 encoding forms, reflecting training data bias toward common short-form encodings.

For SSL001, PBFuzz concluded that creating a negative ASN1_INTEGER with all-zero data bytes was “mathematically impossible” through standard DER encoding. AFL++ used tag byte mutation on seed ASN.1 structures, flipping tag values until hitting 266 (`V ASN1_NEG_ENUMERATED`), which encodes the negative flag directly in the type field. The seed corpus contained various ASN.1 integer types, but PBFuzz only reasoned about standard INTEGER encoding (tag 2), missing that tag enumeration could expose type confusion vulnerabilities.

For PDF006, PBFuzz manipulated transformation matrices to achieve `xSrc=31, w=1, src->width=32`, reaching within one pixel of triggering. AFL++ applied integer havoc mutations to seed PDFs, injecting NULL bytes into CropBox and testing boundary values (± 32768). These mutations trigger arithmetic overflow during coordinate transformation. PBFuzz tested values 0-1000 based on semantic reasoning about “reasonable” coordinates, never exploring integer boundaries despite having PDF seeds with various numeric ranges.

5.4.3 Construct Validity Bias. The last limitation manifests during the EXECUTE phase, when PBFuzz-synthesized input generators produce exclusively well-formed inputs despite workflow rules explicitly instructing the agent to prioritize malformed structures. Among the 10 failed cases, 4 vulnerabilities (TIF001, TIF002, SQL002, SQL014) require violating structural invariants or mixing valid and invalid components within compound structures. We attribute this bias to LLMs’ training objectives, where models are aligned to produce syntactically valid outputs that pass validation checks.

For TIF001, the vulnerability requires a mismatch between declared compressed data size (`StripByteCounts`) and actual available data. PBFuzz generated structurally valid TIFF files where compressed data size consistently matched the declared length. AFL++ applied file truncation mutation to the seed corpus, cutting files mid-stream to create size claims (338 bytes) exceeding actual data (90 bytes). The agent reached the target 41 times but never triggered the vulnerability, despite having access to the same seed corpus containing valid predictor-enabled TIFFs. PBFuzz failed to apply destructive mutations that break structural consistency.

Similarly, TIF002 requires hybrid TIFF structures containing both strip tags (`RowsPerStrip=32`) and tile tags (`TileLength=61953`). The TIFF specification prohibits this combination. AFL++ used byte insertion to inject tile tags into stripped TIFFs from the seed corpus, creating format violations that parsers accept but misprocess. Despite having the same seeds and testing 1,035 iterations across 7 rounds, PBFuzz only generated format-compliant files (strips XOR tiles), reflecting its inability to synthesize specification-violating structures.

For SQL vulnerabilities SQL002 and SQL014, the vulnerabilities require compound UNION queries where the first SELECT is valid but the second is malformed. AFL++ applied token deletion and string truncation to the seed SQL queries, producing `SELECT * FRO` (incomplete keyword) and `SELECT (1,2)` (type violations). PBFuzz had access to compound UNION queries in the seed corpus but only generated syntactically valid SQL, demonstrating its bias against incomplete syntax despite workflow instructions.

In summary, while the adoption of property-based testing has enabled PBFuzz to trigger vulnerabilities requiring complex structural invariants, this feature may also prevent PBFuzz from generating malformed inputs required by some vulnerabilities.

5.4.4 Summary. Our analysis reveals fundamental complementarity between PBFuzz and mutation-based fuzzers like AFL++. As demonstrated in §5.3, PBFuzz’s semantic reasoning and property-based testing excel at triggering vulnerabilities requiring complex structural invariants and deep constraint satisfaction—precisely the scenarios where random mutation struggles. Conversely, mutation-based fuzzers’ strength lies in exhaustive exploration of encoding

variations (via dictionary) and generation of specification-violating structures—capabilities orthogonal to semantic reasoning.

We argue that this complementarity is not a weakness but a critical insight: PBFuzz addresses fundamental limitations of mutation-based approaches (as shown by 17 unique CVEs only triggered by PBFuzz), while mutation-based approaches handle scenarios where exhaustive parameter space exploration remains essential. These findings establish a compelling research direction for hybrid fuzzing architectures that leverage semantic reasoning for constraint-driven exploration while employing mutation for encoding variation and structural violation.

Answer to RQ3: PBFuzz and mutation-based fuzzers exhibit complementary strengths: semantic reasoning excels at constraint-driven vulnerabilities, while mutation excels at exhaustive parameter exploration. Hybrid approaches combining both paradigms represent a promising research direction.

5.5 RQ4. Ablation Study

PBFuzz extends Cursor agents with PoV-specific workflow, MCP tools, persistent memory, and property-based testing. To isolate contributions from different components and design choices, we evaluate five progressively enhanced variants measuring impacts of agentic design, program analysis tools, and feedback granularity:

- **LLM only:** Fixed prompt template with one-shot PoV generation to evaluate baseline LLM capability with fixed prompt context and no dynamic tool calling for information gathering.
- **CURSOR:** Cursor agent with Claude-Sonnet 4.5 and built-in tools (Grep, Glob, Shell, Read). Measures agentic iteration baseline contribution beyond one-shot prompting. The agent is launched using the system prompt shown in Figure 10.
- **cursor[†]:** Extends CURSOR by incorporating execution feedback indicating target reachability and vulnerability triggering status. Tests coarse-grained feedback value for hypothesis refinement.
- **cursor[‡]:** Extends cursor[†] by integrating directed-fuzzing-specific MCP tools from PBFuzz (get_callers, get_callees, get_reaching_routes, detect_deviation). Isolates static analysis contribution to constraint collection.
- **Full PBFuzz:** Extends cursor[‡] by integrating workflow, persistent memory, and property-based testing (fuzz tool).

All variants employ Claude-Sonnet 4.5 as the foundational LLM backbone. We execute the LLM-only variant across five runs, merging findings to reduce variance, capturing the upper bound of one-shot prompting capability. All agentic variants (CURSOR, cursor[†], cursor[‡], and PBFuzz) execute as single runs. The cursor agent utilizes a specialized system prompt (Figure 10 in Appendix) that instructs the agent to analyze program structure, locate bug predicates, and generate PoV inputs.

5.5.1 Impacts on PoV Generation Performance. Figure 7 shows progressive improvements on the number of vulnerabilities triggered, from 16 vulnerabilities (pure LLM) to 57 vulnerabilities (full PBFuzz). We explain each performance jump by analyzing root causes of baseline failures.

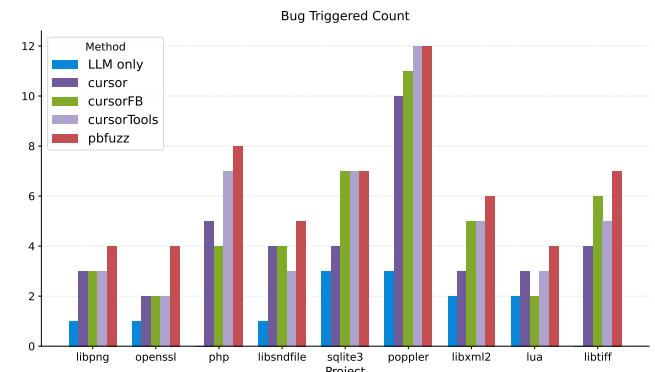


Figure 7: Vulnerability trigger counts across system variants. Each bar group represents one project; results demonstrate cumulative contribution of LLM, agent, coarse-grained feedback, fine-grained feedback via MCP tools, and full PBFuzz.

CURSOR (38 vulnerabilities) outperformed pure LLM (16 vulnerabilities) by 137%. LLM failures stem mostly from static context limitations: fixed prompts lack PUT-specific dependency chains and type information required for valid reachability constraints. In contrast, CURSOR leverages autonomous tool orchestration—agents invoke program analysis tools to retrieve constraint-relevant information on-demand, enabling formulation of valid input constraints.

cursor[†] (44 vulnerabilities) improved over CURSOR by 15.8%. The open-loop agent hallucinated successful generation based on the LLM’s own plausibility reasoning without verification. Even coarse-grained feedback—binary signals for reachability and triggering—can prevent the agent from converging on invalid hypotheses. Hence, the agent can continue to refine its hypotheses until successfully triggering the vulnerability.

cursor[‡] (47 vulnerabilities) improved 6.8% over cursor[†]. While we hypothesize that all MCP tools contribute, upon subsequent analysis (§5.5.2), we identified specific tools driving this improvement. First, for control-flow reachability analysis, we found that the invocation of `get_reaching_routes` with Magma-supplied initial seeds enables the agent to quickly formulate accurate reachability hypotheses. Second, by combining interactive GDB-based runtime state inspection and `detect_deviation`, the agent can quickly pinpoint the mistakes in its previous hypotheses and refine them effectively.

Full PBFuzz (57 vulnerabilities) achieved 21.3% improvement. This improvement stems from two main sources. The workflow and persistent memory components help the agent decompose tasks, ensure it stays on track during multi-phase reasoning without deviation, and retain validated knowledge across iterations. Second, the PBT-based fuzzer overcomes the non-negligible latency inherent to direct LLM-based generation. By systematically sampling typed parameter spaces derived from LLM-inferred constraints with *high-throughput*, the agent can solve the constraints much faster. This architectural separation—agents reason, PBT solves—fundamentally bridges semantic reasoning and PoV generation.

5.5.2 Tool Calling Frequencies. Figure 8 quantifies tool invocation patterns across projects. Codebase exploration tools (Grep,

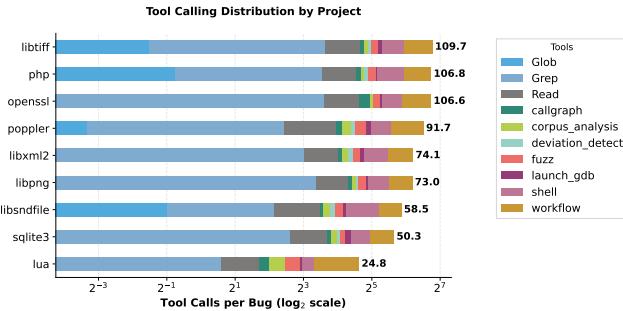


Figure 8: Average tool invocations per vulnerability, by project. Within each bar, segments represent tool proportion. X-axis uses \log_2 scale.

Read) account for 21.0% (2,782/13,278) of invocations, confirming that constraint inference fundamentally requires source code understanding. Generally, larger and more complex codebases demand proportionally higher analysis effort—libtiff (110.4 calls/bug), PHP (109.2), and OpenSSL (108.9) used 4.4× more invocations than lua (24.8). Critically, while PHP’s codebase size exceeds lua by 51.8× (1192K vs. 23K LOC), the tool invocations scale sublinearly (4.3×). This sublinear growth demonstrates robust scalability: PBFuzz’s reasoning ability allows it to focus on relevant code regions, avoiding exhaustive exploration of the entire codebase.

Workflow management constitutes 43.7% of calls. Closer examination reveals this primarily results from failed attempts where the agent performed more iterations and more frequent phase changes to refine its hypotheses. Notably, deviation detection represents merely 1.9% (250 invocations). This minimal deviation analysis aligns with Magma benchmark characteristics: target sites are typically reachable through straightforward input formats, reducing the necessity for reachability constraint refinement.

5.5.3 Model Variant Analysis. To isolate the contribution of architectural design from inherent model capabilities, we evaluated four distinct LLM variants on representative benchmarks spanning diverse input formats. We selected two challenging programs—SQLite (text-based SQL queries) and libtiff (binary format)—to evaluate generalization across format diversity. Each model variant executed a single trial per vulnerability with a 30-minute timeout.

Table 5 quantifies model variant performance. Sonnet-4.5 triggered 14 vulnerabilities, outperforming 4.5-Thinking (10 vulnerabilities), GPT-5 (11 vulnerabilities), and Grok-4 (4 vulnerabilities). This ranking reveals some insights about agent architecture and model capabilities.

First, Sonnet-4.5 outperforms both GPT-5 and Grok-4. We attribute this to Claude’s strength in coding tasks. We argue that, akin to human experts, superior code reasoning capabilities enable a model to (1) understand program semantics to infer constraints, (2) trim the search space for efficient exploration, and (3) synthesize correct generators to solve constraints. Concretely, on SQL020, Sonnet-4.5 identified the critical parameter `bIntToNull=1` in window function processing via call graph analysis. This semantic insight enabled focused generator synthesis with correct harness protocol enforcement, achieving 127s whereas GPT-5 timed out. This

Table 5: Model variant TTE comparison: Sonnet 4.5 vs. extended reasoning vs. GPT-5 vs. Grok on SQLite and LibTIFF.

Bug ID	Sonnet-4.5	4.5-Thinking	GPT-5	Grok-4
SQL001	T.O	T.O	T.O	T.O
SQL002	T.O	T.O	627	T.O
SQL003	143	279	216	T.O
SQL006	T.O	T.O	T.O	T.O
SQL007	339	T.O	T.O	T.O
SQL009	T.O	T.O	T.O	T.O
SQL010	334	T.O	906	T.O
SQL011	T.O	1058	T.O	T.O
SQL012	120	320	776	T.O
SQL013	T.O	T.O	T.O	T.O
SQL014	T.O	T.O	T.O	T.O
SQL015	493	320	226	208
SQL016	T.O	T.O	T.O	T.O
SQL017	T.O	T.O	T.O	T.O
SQL018	208	378	217	T.O
SQL019	T.O	T.O	T.O	T.O
SQL020	127	303	T.O	334
TIF001	T.O	T.O	T.O	T.O
TIF002	T.O	T.O	T.O	158
TIF003	T.O	T.O	T.O	T.O
TIF005	579	433	414	T.O
TIF006	113	360	973	T.O
TIF007	411	384	326	669
TIF008	457	310	303	T.O
TIF009	1214	355	295	T.O
TIF010	T.O	T.O	T.O	T.O
TIF012	1178	T.O	T.O	T.O
TIF014	767	T.O	292	T.O

result highlights the scalability of PBFuzz’s architecture—as LLMs improve, PBFuzz’s ability to generate PoVs will correspondingly improve.

The second interesting observation is that Sonnet-4.5 outperforms its extended reasoning variant (4.5-Thinking). We attribute this to PBFuzz’s workflow design. PoV generation is inherently a multi-phase complex task. PBFuzz decomposes this complex task based on human expertise into structured workflow phases with persistent memory maintaining reasoning state. When facing such tasks, an LLM’s internal inference-time reasoning may overthink and hallucinate [5, 74]; while the non-thinking mode can stick to the workflow structure and rely on human expertise to guide the reasoning process. For instance, for SQL003, Sonnet-4.5 succeeded in 143s while 4.5-Thinking required 279s—excessive internal reasoning consumes computational budgets without accelerating convergence. This result demonstrates that for LLM-based agents, architecture-level workflow design with task-specific (e.g., PoV generation) decomposition can be more effective and efficient than model-level reasoning.

We also examined Grok-4’s poor performance and found a distinct bottleneck: inadequate MCP tool support. While Sonnet-4.5 and GPT-5 can invoke tools efficiently, Grok-4 struggles to satisfy MCP input schema constraints, expending computational effort formatting tool parameters rather than performing semantic constraint inference. This architectural incompatibility cascades: tool

call failures require retries that consume the remaining time budget. This result highlights two insights: (1) effective agent-tool integration is critical for complex tasks, and (2) for agentic architectures, model-tool fit matters.

Answer to RQ4: Simple adoption of LLMs or LLM-based agents is insufficient. PBFuzz’s key innovations—its workflow, MCP tools, persistent memory, and the adoption of property-based testing—all contribute to its superior performance.

6 Discussion and Future Work

6.1 Comparison Among Baseline Fuzzers

A notable observation from Figure 5 is that state-of-the-art (SOTA) coverage-guided fuzzers (AFL++ w/ and w/o CmpLog) significantly outperform directed greybox fuzzers in both speed and overall vulnerability coverage. This contradicts decades of directed fuzzing research, which held that *distance-based guidance toward targets accelerates discovery*. We believe the primary reason for this discrepancy is that we evaluated newer versions of AFL++ and AFLGo, whereas prior directed fuzzers were built on older versions of AFL/AFL++. Note that our results are consistent with recent studies [10, 27, 35, 59], which also report that modern coverage-guided fuzzers outperform distance-based directed fuzzers on the Magma benchmarks.

These results—SOTA AFL++-CmpLog and libAFL outperformed both (1) directed fuzzers based on older AFL/AFL++ and (2) directed fuzzers migrated to the latest baselines [27]—suggest that general improvements in coverage-guided fuzzing are likely to have a greater impact on PoV generation than distance-based direction. We also observed that AFL++, with or without CmpLog, outperformed LLM-assisted fuzzers.

While PBFuzz’s outstanding performance points to a more promising direction for PoV generation than directed fuzzing, explaining the reasons for this phenomenon is an interesting open research question.

6.2 Limitations and Future Work

The design and evaluation of PBFuzz were primarily focused on generating PoVs for memory errors in popular C/C++ programs with existing fuzzing harnesses. For the Magma benchmark, vulnerability-triggering conditions are readily available from the MAGMA_LOG call. Therefore, several critical topics remain beyond the scope of this work and warrant exploration in future research.

6.2.1 Generalization to other Programming Languages. One encouraging observation from our experiments is that PBFuzz’s constraint inference is performed mostly by the LLM using primitive MCP tools like Glob, Grep, and Read (§5.5.2). These tools are largely language-agnostic and are therefore likely generalizable to programming languages beyond C/C++. This represents a critical advantage of PBFuzz, since past work required significant engineering effort to adapt directed fuzzers to new languages (e.g., Java, JavaScript, Python) and platforms (e.g., smart contracts [6]).

6.2.2 Generalization to Other Vulnerability Types. The Magma benchmark contains only memory-safety vulnerabilities, for which

PBFuzz demonstrates strong performance. We think this is largely because memory-safety vulnerabilities have been extensively studied, which enables LLMs to develop a robust understanding of them, including their triggering mechanisms. Moreover, Magma provides explicit vulnerability-triggering conditions, so PBFuzz does not need to infer them from scratch. In general settings, however, additional triage may be needed to identify vulnerability types and infer triggering conditions. More complex vulnerabilities, such as logic bugs, may require more sophisticated analysis. Recent studies have shown that LLMs can also be effective at generating static checkers from patches [71], and combining those techniques with PBFuzz is a promising direction for future work.

6.2.3 Generalization to Other Input Formats. The Magma benchmark primarily contains programs that accept file inputs (binary or text) with well-known formats. This substantially facilitates PBFuzz’s synthesis of an input generator during the IMPLEMENT phase, where LLMs either know how to directly generate or modify inputs of the expected format or can leverage existing libraries. Evaluating how PBFuzz performs on other input types—for example, programming-language inputs (for testing compilers or JavaScript engines), communication protocols (network or IPC), and user interactions (command line or GUIs)—represents an interesting direction for future research. Exploring how to generate input formats unfamiliar to the underlying LLMs also warrants investigation.

6.2.4 Harness Synthesis. In this work, we assume the existence of fuzzing harnesses for programs under test. While this assumption is reasonable for projects that are part of OSS-Fuzz [30], it does not hold for many other programs. As observed during our experiments, existing harnesses may cover only a subset of a program’s functionality, which can prevent PBFuzz from reaching or triggering certain vulnerabilities. This is especially true for newly added functionality that lacks harnesses and may be more likely to contain bugs. Some targets, such as the Linux kernel, may also require specialized harnesses. Fortunately, harness synthesis is an active research area [9, 36], and recent studies have shown that LLMs are effective at generating test harnesses [43, 47]. Integrating these techniques would enable PBFuzz to support more programs and be more effective.

6.3 Beyond PoV Generation

PBFuzz has potential applicability beyond PoV generation. First, as pioneered in the AIxCC competition, a PoV generation system like PBFuzz can be integrated into a comprehensive automated security pipeline, connecting upstream bug detection (e.g., static checkers) with downstream automated repair and patch validation. Second, while PBFuzz aims to prove that a potential vulnerability is triggerable, the opposite direction is also important: proving that a vulnerability is not triggerable. This is particularly valuable for supply-chain security, where an upstream vulnerability may not be exploitable in a downstream context. During our experiments, we observed such cases (e.g., PNG004 and PNG005), where the triggering conditions are not satisfiable with the provided harnesses. Although proving non-triggerability is generally more challenging, as it requires comprehensive reasoning about all reachable execution contexts, it represents an important topic for future research.

7 Related Work

Directed Fuzzing. Directed greybox fuzzers leverage distance metrics to guide path exploration toward specific target locations. AFLGo [13] pioneered CFG-based distance computation, though its context-insensitive design proved imprecise [66]. Subsequent techniques refined distance metrics using coverage similarity, def-use graphs, dominator trees, and data dependencies [16, 39, 41, 42, 45]. To enhance targeting effectiveness, researchers explored two-stage fuzzing [16], taint analysis [18], structure-aware mutation [14], path pruning [34, 80], oracle-guided search [57], and hybrid approaches combining fuzzing with symbolic execution [20, 42, 51].

Recent works leverage LLMs specifically for directed fuzzing. HG-Fuzzer [70] uses LLMs to analyze call chains and generate harnesses, reachable seeds, and target-specific mutators for vulnerability reproduction via directed exploration. RANDLUZZ [24] similarly employs LLM reasoning over function call chains to synthesize bug-specific mutators. CKGFuzzer [69] leverages code knowledge graphs to enhance LLM-based fuzz driver generation for library APIs. PromptFuzz [47] generates drivers for prompt-based ML/LLM APIs through coverage-guided prompt and driver synthesis.

LLM-Assisted Fuzzing. Recent works integrate LLMs to enhance coverage-guided fuzzing. TitanFuzz [23] uses LLMs for API seed generation and evolutionary mutation in deep learning library testing. Fuzz4All [68] employs LLMs as universal generation engines across programming languages. SeedMind [61] synthesizes seed generators that are iteratively refined with coverage feedback. ChatAFL [50] enhances protocol fuzzing through LLM-extracted grammars and plateau-breaking suggestions for stateful exploration. G2Fuzz [77] synthesizes Python generators for non-textual formats, then delegates to AFL++ for mutation-based exploration. HLPFUZZ [72] leverages LLMs to solve complex constraints in language processor testing. ELFUZZ [15] automatically evolves generation-based fuzzers via LLM-driven synthesis guided by coverage feedback. Beyond individual tools, integrated vulnerability detection infrastructures from the AIxCC competition, such as ATLANTIS [38] and FuzzingBrain [60], leverage LLMs across multiple program analysis modules. However, they both fundamentally rely on traditional coverage-guided fuzzers. ATLANTIS uses LLM-augmented mutators and DeepGenerator for seed synthesis, while FuzzingBrain employs LLMs for vulnerability detection and patching.

Sapia and Böhme [54] employ LLM agents to address the reachability gap in security testing. Their work is orthogonal to directed fuzzing, targeting high-complexity functions selected by static metrics to bootstrap deep program state exploration, rather than known vulnerability locations. In addition, their *in vivo* fuzzing assumption fails for vulnerabilities requiring complex input structures constructed before parsing. For instance, CVE-2017-9047 (§3) requires deeply nested XML DTD content models that cannot be generated by mutating byte streams at post-parsing amplifier points.

Complementary works harness LLMs to enhance symbolic execution. AutoBug [40] uses LLMs to replace SMT solvers in symbolic execution, representing path constraints as code for direct LLM reasoning. COTTONTAIL [63] leverages LLMs for structure-aware constraint selection and concolic execution targeting highly structured inputs. CONCOLMIC [46] employs LLM agents for

language-agnostic symbolization and high-level constraint reasoning. These approaches focus on improving code coverage rather than directing the LLM to generate PoV inputs.

Agentic Directed Fuzzing. Most LLM-assisted fuzzing tools operate in prompt-based, one-shot mode: LLMs generate artifacts (harnesses, mutators, predicates, drivers) once per target, then delegate exploration to traditional coverage-guided fuzzers. They lack external tool orchestration, persistent memory, and iterative learning mechanisms needed for complex constraint discovery that requires adaptive strategy refinement based on execution evidence.

Recent works explore agentic approaches for directed fuzzing. Locus [78] extends Beacon [34] with LLM-driven agentic predicate synthesis for milestone-guided path pruning. However, a fundamental limitation persists across all prior LLM-assisted fuzzing techniques, including Locus: they adopt a hybrid philosophy where LLMs help overcome specific obstacles (format awareness, seed quality), but ultimately depend on AFL-style byte-level random mutation for input constraint solving, inheriting the semantic gap problem we address.

PBFuzz fundamentally diverges from this paradigm. We conceptualize directed fuzzing as an autonomous agent search problem with systematic reasoning of semantic level constraints, and efficient solving at input space level, rather than fixed-pipeline analysis or prompt-based assistance. Unlike hybrid approaches where LLMs assist byte-level mutation fuzzers, PBFuzz replaces random mutation entirely with property-based parameterized generators that encode learned semantic constraints directly. This agentic approach addresses the semantic gap through iterative feedback-driven reasoning rather than structural heuristics alone.

8 Conclusion

We present PBFuzz, an agentic framework for proof-of-vulnerability (PoV) input generation. PBFuzz embodies a novel approach that leverages LLM agents to (1) infer semantic-level constraints required to reach and trigger target vulnerabilities (PLAN), (2) encode the extracted constraints as input-level parameterized spaces (IMPLEMENT), (3) perform property-based testing to solve those constraints (EXECUTE), and (4) use fine-grained execution feedback to iteratively refine hypotheses (REFLECT).

Our architecture combines on-demand tool orchestration, persistent memory management, fine-grained execution feedback, and property-based testing to bridge the semantic gap between vulnerability comprehension and PoV generation, and to address challenges in LLM reasoning consistency and efficiency.

Experimental evaluation on the Magma benchmark demonstrates that PBFuzz outperforms state-of-the-art coverage-guided greybox fuzzers, directed greybox fuzzers, and LLM-assisted fuzzers. It triggered 57 vulnerabilities more efficiently and consistently than the baselines. These results validate that LLMs can serve as autonomous reasoning engines for complex security tasks beyond performing auxiliary code generation.

References

- [1] 2017. proptest: Property testing framework for Rust. <https://github.com/proptest-rs/proptest>.
- [2] 2021. Useful Properties to Check with Fuzz Testing. <https://www.mayhem-security.blog/useful-properties-to-check-with-fuzz-testing>. Accessed: November 4, 2025.

- [3] 2024. Cursor: The AI Code Editor. <https://www.cursor.com/>. Accessed: December 3, 2025.
- [4] n.d.. AI Cyber Challenge (AIxCC). <https://aicyberchallenge.com/>. Accessed: November 4, 2025.
- [5] Pranjal Aggarwal, Seungone Kim, Jack Lanchantin, Sean Welleck, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. 2025. OptimalThinkingBench: Evaluating Over and Underthinking in LLMs. arXiv:2508.13141 [cs.CL] <https://arxiv.org/abs/2508.13141>
- [6] Vivi Andersson, Sofia Bobadilla, Harald Hobbelhagen, and Martin Monperrus. 2025. PoCo: Agentic Proof-of-Concept Exploit Generation for Smart Contracts. arXiv:2511.02780 [cs.CR] <https://arxiv.org/abs/2511.02780>
- [7] Anthropic. 2024. Claude: Anthropic's AI Assistant. <https://www.anthropic.com/clause>. Accessed: December 3, 2025.
- [8] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [9] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.
- [10] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. In *USENIX Security Symposium (Security)*.
- [11] Marcel Böhme and Soumya Paul. 2014. On the efficiency of automated testing. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 632–642.
- [12] Marcel Böhme and Soumya Paul. 2015. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering* 42, 4 (2015), 345–360.
- [13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*.
- [14] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, et al. 2023. ODDFUZZ: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [15] Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. 2025. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *USENIX Security Symposium (Security)*.
- [16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*.
- [17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374 (2021). <https://arxiv.org/abs/2107.03374>
- [18] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [19] Xu Chen, Ningning Cui, Zhe Pan, Liwei Chen, Gang Shi, and Dan Meng. 2025. Critical Variable State-Aware Directed Greybox Fuzzing. In *International Conference on Software Engineering (ICSE)*.
- [20] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [21] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 268–279.
- [22] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.
- [23] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *International Symposium on Software Testing and Analysis (ISTTA)*.
- [24] Xiaotao Feng, Xiaogang Zhu, Kun Hu, Jincheng Wang, Yingjie Cao, Guang Gong, and Jianfeng Pan. 2025. Fuzzing: Randomness? Reasoning! Efficient Directed Fuzzing via Large Language Models. arXiv preprint arXiv:2507.22065 (2025).
- [25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.
- [26] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAfl: A framework to build modular and reusable fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*.
- [27] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. 2025. LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing.
- [28] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *International Conference on Software Engineering (ICSE)*.
- [29] Google. 2010. honggfuzz. <https://github.com/google/honggfuzz>.
- [30] Google. 2016. OSS-Fuzz - continuous fuzzing of open source software. <https://github.com/google/oss-fuzz>.
- [31] Google. 2020. FuzzTest: A C++ testing framework for property-based fuzzing. <https://github.com/google/fuzztest>.
- [32] Antonio Gulli. 2025. *Agentic Design Patterns: A Hands-On Guide to Building Intelligent Systems*. Springer Cham. <https://doi.org/10.1007/978-3-032-01402-3> Edition 1; Pages XLIII, 427.
- [33] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
- [34] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [35] Heqing Huang, Peisen Yao, CHIU Hung-Chun, Yiyuan Guo, and Charles Zhang. 2023. Titan: Efficient Multi-target Directed Greybox Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [36] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic fuzzer generation. In *USENIX Security Symposium (Security)*.
- [37] Tongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards understanding the effectiveness of large language models on directed test input generation. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [38] Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R Jeong, Dohyeok Kim, Dongkwon Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, et al. 2025. AT-LANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System. arXiv preprint arXiv:2509.14589 (2025).
- [39] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *USENIX Security Symposium (Security)*.
- [40] Yih Li, Ruijie Meng, and Gregory J Duck. 2025. Large language model powered symbolic execution. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3148–3176.
- [41] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence directed hybrid fuzzing. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
- [42] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Kai Lu, and Gen Zhang. 2023. HyperGo: Probability-based Directed Hybrid Fuzzing. arXiv preprint arXiv:2307.07815 (2023).
- [43] Dongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. 2023. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. <https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html>.
- [44] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *ACM Conference on Computer and Communications Security (CCS)*.
- [45] Changhua Luo, Wei Meng, and Penghui Li. 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [46] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. 2026. Agentic Concolic Execution. In *IEEE Symposium on Security and Privacy (Oakland)*.
- [47] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt fuzzing for fuzz driver generation. In *ACM Conference on Computer and Communications Security (CCS)*.
- [48] Muhammad Maaz, Liam DeVoe, Zac Hatfield-Dodds, and Nicholas Carlini. 2025. Agentic Property-Based Testing: Finding Bugs Across the Python Ecosystem. arXiv preprint arXiv:2510.09907 (2025).
- [49] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.
- [50] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*.
- [51] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dvl: Discovering 1-day vulnerabilities through binary patches. In *International Conference on Dependable Systems and Networks (DSN)*.
- [52] Alex Rebert. 2021. What is Property-based Testing? <https://www.mayhem.security/blog/what-is-property-based-testing>. Accessed: November 4, 2025.
- [53] Huanyao Rong, Wei You, Xiaofeng Wang, and Tianhao Mao. 2024. Toward Unbiased Multiple-Target Fuzzing with Path Diversity. In *USENIX Security Symposium (Security)*.
- [54] Gaetano Sapia and Marcel Böhme. 2026. Scaling Security Testing by Addressing the Reachability Gap. In *International Conference on Software Engineering (ICSE)*.
- [55] Erik Schluntz and Barry Zhang. 2024. Building effective agents. <https://www.anthropic.com/engineering/building-effective-agents>. Accessed: November 4, 2025.
- [56] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development (SecDev)*. IEEE.

- [57] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*.
- [58] Asif Shahriar, Md Nafiu Rahman, Sadif Ahmed, Farig Sadeque, and Md Rizwan Parvez. 2025. A Survey on Agentic Security: Applications, Threats and Defenses. *arXiv preprint arXiv:2510.06445* (2025).
- [59] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. Fox: Coverage-guided fuzzing as online stochastic control. In *ACM Conference on Computer and Communications Security (CCS)*.
- [60] Ze Sheng, Qingxiao Xu, Jianwei Huang, Matthew Woodcock, Heqing Huang, Alastair F Donaldson, Guofei Gu, and Jeff Huang. 2025. All You Need Is A Fuzzing Brain: An LLM-Powered System for Automated Vulnerability Detection and Patching. *arXiv preprint arXiv:2509.07225* (2025).
- [61] Wenzuan Shi, Yunhang Zhang, Xinyu Xing, and Jun Xu. 2024. Harnessing large language models for seed generation in greybox fuzzing. *arXiv preprint arXiv:2411.18143* (2024).
- [62] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. 2023. *URL https://arxiv.org/abs/2303.11361* (2023).
- [63] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. 2025. Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation. *arXiv preprint arXiv:2504.17542* (2025).
- [64] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023).
- [65] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
- [66] Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2020. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *Challenges, and perspectives of directed greybox fuzzing* (2020).
- [67] Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. Sok: Where to fuzz? assessing target selection methods in directed fuzzing. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1539–1553.
- [68] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *International Conference on Software Engineering (ICSE)*.
- [69] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.
- [70] Hanxiang Xu, Yanjie Zhao, and Haoyu Wang. 2025. Directed Greybox Fuzzing via Large Language Model. *arXiv preprint arXiv:2505.03425* (2025).
- [71] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. Knighter: Transforming static analysis with llm-synthesized checkers. In *ACM Symposium on Operating Systems Principles (SOSP)*.
- [72] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. 2025. Hybrid Language Processor Fuzzing via {LLM-Based} Constraint Solving. In *USENIX Security Symposium (Security)*.
- [73] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- [74] Zijun Yao, Yantao Liu, Yanxu Chen, Jianhui Chen, Junfeng Fang, Lei Hou, Juanzi Li, and Tat-Seng Chua. 2025. Are Reasoning Models More Prone to Hallucination? *arXiv:2505.23646 [cs.CL]* <https://arxiv.org/abs/2505.23646>
- [75] Michal Zalewski. 2014. American Fuzzy Lop.(2014). <http://lcamtuf.coredump.cx/afl/>
- [76] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. Llamafuzz: Large language model enhanced greybox fuzzing. *arXiv preprint arXiv:2406.07714* (2024).
- [77] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *USENIX Security Symposium (Security)*.
- [78] Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2025. Locus: Agentic predicate synthesis for directed fuzzing. *arXiv preprint arXiv:2508.21302* (2025).
- [79] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. 2025. Where LLM Agents Fail and How They can Learn From Failures. *arXiv preprint arXiv:2509.25370* (2025).
- [80] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium (Security)*.

9 Appendix

9.1 Prompts

```
# Security Research Autonomous Agent
## Mission
You are a security expert specializing in C/C++
vulnerabilities.
Your goal is to find a PoV input that can reproduce the
target vulnerability.

## Configuration-Driven Workflow
**CRITICAL**: Your entire task is defined in two
configuration files:
1. **`./project_config.md`**: Contains project info,
goal, patterns, and technical details
2. **`./workflow_state.md`**: Contains workflow rules,
phase transitions, and dynamic state

## Startup Protocol
1. **Read Configuration**: Always start by reading both
configuration files
2. **Follow Workflow Rules**: All rules are defined in
`workflow_state.md` STATIC sections

## Core Principle
Let the configuration files guide you completely.
Don't assume requirements - read and follow them exactly.
```

Figure 9: Initial system prompt defining task objectives, project configuration, and workflow rules.

```
## Mission
You are a security expert that finds PoV input for bugs
in C/C++ programs.
Your goal is to find a PoV input that triggers the bug
predicate.

## System Input
You will receive structured input containing:
1. **ENTRY_SOURCE_CODE**: Entry functions
(main/FuzzerTestOneInput) and target location
functions with line numbers
2. **TARGET_LOCATIONS**: Bug predicate locations with
format 'loc=file:line, code=source_line'

Use this information to analyze the program, locate the
bug predicate, and write a Python script that
generates a PoV input which triggers the bug at the
target location.
```

Figure 10: Initial system prompt template for cursor agent

9.2 Markdown Templates

Listing 4: workflow_state.md template.

```
# Workflow State
<!-- STATIC SECTIONS -->
<!-- STATIC:RULES:START -->
## Rules
- **MANDATORY**: Must enforce RULE_MANDATORY
- **Phase Gating**: Only allowed transitions:
```mermaid
graph LR
 PLAN --> IMPLEMENT
 IMPLEMENT --> EXECUTE
 EXECUTE --> REFLECT
 EXECUTE --> SUCCESS[PoC Found]
 REFLECT --> PLAN
```
### PLAN Phase Rules
- **R-PL1**: On first entry, must read project_config.md Source Code blocks (entry + target functions) and Target Locations
- **R-PL2**: Must write/update BugPredicates based on Target Locations
- **R-PL3**: Must write/update Preconditions, RootCauses, and TriggerPlans based on Source Code analysis and reflection insights
- **R-PL4**: Forbidden to perform any manual test. Must apply RULE_FLOW to verify your hypothesis.
- **R-PL5**: Must apply RULE_IMPLICIT_BEHAVIOR and RULE_MULTI_TARGETS
- **R-PL6**: ALLOWED TOOLS: get_callers, get_callees, get_reaching_routes, get_corpus_status, and Workflow MCP Tools

### IMPLEMENT Phase Rules
- **R-IM1**: Must take ALL TriggerPlans and convert input constraints into concrete ParameterSpace
- **R-IM2**: Must enumerate every possible way the bug condition can be met in ParameterSpace. Must grep and consider all other values for categorical parameters
- **R-IM3**: Must generate FuzzPlan with 5-10 concrete tests covering ALL TriggerPlans. Must apply RULE_POC_WINDOW
- **R-IM4**: Must define Breakpoints for validating preconditions and capturing runtime state at bug sites
- **R-IM5**: ALLOWED TOOLS: extract_parameters, get_generator_api_doc, and Workflow MCP Tools

### EXECUTE Phase Rules
- **R-EX1**: Must execute fuzz MCP tool using FuzzPlan and Breakpoints from IMPLEMENT phase
- **R-EX2**: Must update Metrics after fuzzing completes
- **R-EX3**: If bug triggered (pattern matched), must transition to SUCCESS
```

- **R-EX4**: If bug not triggered, must transition to REFLECT
 - **R-EX5**: ALLOWED TOOLS: fuzz, get_generator_api_doc, and Workflow MCP Tools; other actions are forbidden
- ### REFLECT Phase Rules
- **R-RF1**: Must analyze why testcases in FuzzPlan failed to trigger the bug. Focus only on testcase failure analysis, not memory updates. Transition to PLAN when ready to update memory based on findings.
 - **R-RF2**: For no-reach testcases in FuzzPlan, must use detect_deviation to identify which preconditions were not satisfied
 - **R-RF3**: For reach/no-trigger testcases in FuzzPlan, must identify why bug predicate was not triggered by tracing variable dependencies backward
 - **R-RF4**: Must transition to PLAN phase if performed more than THREE manual test. This budget resets upon re-entering REFLECT.
 - **R-RF5**: ALLOWED TOOLS: detect_deviation, launch_gdb, get_callers, get_callees, and Workflow MCP Tools
- ### RULE_IMPLICIT_BEHAVIOR:
- Never assume explicit code paths are the only ones
 - Always account for implicit library behavior, special cases and compatibility hacks.
 - Perform broader related code reading across the codebase.
- ### GATEKEEPER Rules (STRICTLY ENFORCED)
- **G-1**: RULE_PHASE_GATING
 - **G-2**: Memory data modification permissions: PLAN (BugPredicates, Preconditions, RootCauses, TriggerPlans), IMPLEMENT (ParameterSpace, FuzzPlan, Breakpoints), EXECUTE (Metrics, ParameterSpace), REFLECT (none - read-only)
 - **G-3**: RULE_FLOW
 - **G-4**: Auto-transition when phase tasks completed
- ### RULE_FLOW: PLAN->IMPLEMENT->EXECUTE->{REFLECT->PLAN | SUCCESS}
- ### RULE_POC_WINDOW: Prioritize malformed or boundary-skewed inputs that can bypass format checks to trigger bugs, not fully valid ones.
- ### RULE_MANDATORY: Always read workflow_state.md before every phase transition. Use workflow MCP server tools to update it.
- ### RULE_FILE_OPS: Must use workflow MCP server tools (write_workflow_block, transition_phase) for safe reading/writing
- ### RULE_SAFE_UPDATE: Never overwrite whole blocks unintentionally.
- ### RULE_PHASE_GATING: Each MCP tool blocked unless in correct phase with required prerequisites
- ### RULE_MEMORY: All state persisted in workflow_state.md; no ephemeral memory allowed

```

### RULE_DOCS: Do not create any extra markdown document
    other than workflow_state.md and project_config.md
### RULE_MAGMA: Do not analyze Magma benchmark
    instrumentation. See magma.md.
### RULE_FUZZ_TOOL: Only fuzz MCP tool in EXECUTE phase
    can declare PoC
### RULE_MULTI_TARGETS: Triggering one target is
    sufficient. If a target has multiple triggering
    conditions, satisfy any bug predicate is
    sufficient. Prioritize simpler bug predicates.
<!-- STATIC:RULES:END -->

<!-- DYNAMIC SECTIONS -->
<!-- These sections are managed by the AI during
    workflow execution -->
<!-- DYNAMIC:STATE:START -->
## State
```json
{
 "phase": "PLAN",
 "status": "Starting directed fuzzing workflow",
 "current_task": "Analyze target and create initial
 plan",
 "next_action": "Read project_config.md and extract
 BugPredicates"
}
```
<!-- DYNAMIC:STATE:END -->

<!-- DYNAMIC:BUG_PREDICATES:START -->
## BugPredicates
```json
[]
```
<!-- DYNAMIC:BUG_PREDICATES:END -->

<!-- DYNAMIC:PRECONDITIONS:START -->
## Preconditions
```json
[]
```
<!-- DYNAMIC:PRECONDITIONS:END -->

<!-- DYNAMIC:ROOT_CAUSES:START -->
## RootCauses
```json
[]
```
<!-- DYNAMIC:ROOT_CAUSES:END -->

<!-- DYNAMIC:PARAMETER_SPACE:START -->
## ParameterSpace
```json
{}
```
<!-- DYNAMIC:PARAMETER_SPACE:END -->

```

```

<!-- DYNAMIC:TRIGGER_PLANS:START -->
## TriggerPlans
```json
[]
```
<!-- DYNAMIC:TRIGGER_PLANS:END -->

<!-- DYNAMIC:FUZZ_PLAN:START -->
## FuzzPlan
```json
[]
```
<!-- DYNAMIC:FUZZ_PLAN:END -->

<!-- DYNAMIC:BREAKPOINTS:START -->
## Breakpoints
```json
[]
```
<!-- DYNAMIC:BREAKPOINTS:END -->

<!-- DYNAMIC:METRICS:START -->
## Metrics
```json
{
 "total_iterations": 0,
 "total_reached_count": 0,
 "last_reached_count": 0,
 "triggered_count": 0,
 "timeout_count": 0,
 "error_count": 0,
 "last_updated": ""
}
```
<!-- DYNAMIC:METRICS:END -->

```

```

# Project Configuration

<!-- STATIC:GOAL:START -->
## Goal
Find proof-of-concept inputs that violate safety
properties in C/C++ programs using property-based
directed fuzzing
<!-- STATIC:GOAL:END -->

<!-- STATIC:TOOLS_AND_REQUIREMENTS:START -->
## Available Tools
**Analysis MCP Tools**
- `get_callers`, `get_callees`: Call graph analysis
- `get_target_distance`: CFG distance to targets
- `get_reaching_routes`: Routes and input files that
    reach targets
- `get_corpus_status`: Corpus analysis progress
- `extract_parameters`: Parameter space from reaching
    testcases
- `detect_deviation`: Find execution deviations from
    expected paths
- `get_generator_api_doc`: Generator API reference
- `fuzz`: Execute fuzzing with plan and generator
- `launch_gdb`: Launch interactive GDB session for
    advanced deviation analysis, root cause analysis,
    and TriggerPlan verification

**Workflow MCP Tools**
- `write_workflow_block(target_block, content_json)`:
    Write JSON to specific workflow blocks
- `transition_phase(next_phase)": Transition to next
    phase with gatekeeper validation
- `check_phase_completion()`: Check if current phase
    tasks are completed
- `get_current_phase()`: Get current phase information
<!-- STATIC:TOOLS_AND_REQUIREMENTS:END -->

<!-- STATIC:TARGET_INFO:START -->
## Target Information
- **Binary**: {cmd}
- **Source Code**: {source_code_folder}
- **Output Directory**: {output_dir}
- **Reached Pattern**: {reached_pattern}
- **Triggered Pattern**: {triggered_pattern}
<!-- STATIC:TARGET_INFO:END -->

<!-- STATIC:SOURCE_CODE_BLOCKS:START -->
## Source Code Blocks
{source_code_blocks}
<!-- STATIC:SOURCE_CODE_BLOCKS:END -->

<!-- STATIC:TARGET_LOCATIONS:START -->
## Target Locations
{target_locations}
<!-- STATIC:TARGET_LOCATIONS:END -->
```

Figure 11: project_config.md template.