DrillAgent: Towards Automated Proof-of-Vulnerability Generation via Execution-State-Aware LLM Reasoning

Haoyu Li
University of Illinois
Urbana-Champaign
Champaign, Illinois, USA
haoyuli9@illinois.edu

Xijia Che
University of Illinois
Urbana-Champaign
Champaign, Illinois, USA
xijiac2@illinois.edu

Yanhao Wang
Independent Researcher
Shanghai, Shanghai, China
wangyanhao136@gmail.com

Xiaojing Liao
University of Illinois
Urbana-Champaign
Champaign, Illinois, USA
xjliao@illinois.edu

Luyi Xing
University of Illinois
Urbana-Champaign
Champaign, Illinois, USA
lxing2@illinois.edu

Abstract

Proof-of-Vulnerability (PoV) generation is a critical task in software security, serving as a cornerstone for vulnerability validation, false positive reduction, and patch verification. While directed fuzzing effectively drives path exploration, satisfying complex semantic constraints remains a persistent bottleneck in automated exploit generation. Large Language Models (LLMs) offer a promising alternative with their semantic reasoning capabilities; however, existing LLM-based approaches effectively lack grounding in concrete execution behavior, limiting their ability to generate precise PoVs.

In this paper, we present DRILLAGENT, an agentic framework that reformulates PoV generation as an iterative *hypothesis–verification–refinement* process. To bridge the gap between static reasoning and dynamic execution, DRILLAGENT synergizes LLM-based semantic inference with feedback from concrete program states. The agent analyzes the target code to hypothesize inputs, observes execution divergences, and employs a novel mechanism to translate low-level execution traces into source-level constraints. This closed-loop design enables the agent to incrementally align its input generation with the precise requirements of the vulnerability. We evaluate DRILLAGENT on SEC-bench, a large-scale benchmark of real-world C/C++ vulnerabilities. Experimental results demonstrate that DRILLAGENT substantially outperforms state-of-the-art LLM agent baselines under fixed budget constraints, achieving a relative improvement of up to 52.8% in PoV reproduction rates. These results highlight the necessity of execution-aware reasoning for reliable PoV generation in complex software systems.

1 Introduction

Vulnerability reproduction is a cornerstone of software security. Given a reported vulnerability, a *Proof-of-Vulnerability* (PoV) is a concrete input that deterministically triggers the vulnerability by violating an intended security property at a specific target site. Reliable PoV generation is critical not only for filtering false alarms from static analysis [5] but also for serving as ground truth in downstream tasks such as patch validation in automated program repair (APR) [2, 11]. However, despite decades of research, automating PoV generation for real-world software remains a persistent challenge, particularly for bugs buried deep within complex logic.

State-of-the-art approaches predominantly rely on *directed grey-box fuzzing* (DGF)[7, 9, 20]. By defining a distance metric between program states and a vulnerability site, DGF heuristically steers execution toward the target. While effective for shallow vulnerabilities, prior work has shown that DGF struggles to reproduce deep, logic-heavy bugs guarded by complex semantic constraints[5, 15]. The fundamental limitation is that geometric distance is feasibility-unaware[4]: a path may be syntactically short yet logically blocked by intricate sanitization checks or data dependencies (e.g., magic bytes, checksums, or protocol invariants). As a result, fuzzers without semantic understanding often expend substantial effort mutating inputs against constraints that cannot be satisfied through coverage-oriented exploration alone.

Large Language Models (LLMs) theoretically offer the semantic reasoning capabilities needed to bridge this gap. An LLM can infer input formats, reconstruct protocol logic, and hypothesize constraint-satisfying inputs directly from source code. However, recent studies demonstrate that off-the-shelf LLMs perform poorly on end-to-end PoV generation tasks [27]. We argue that this failure is not merely due to model capacity but stems from a mismatch in problem formulation. PoV generation is not a static translation problem but an interactive debugging process that requires reasoning over execution outcomes. LLMs operate primarily in a conceptual space derived from static text and lack grounding in the program's concrete runtime behavior. Without observing how memory layouts, variable states, or branch conditions evolve during execution, LLMs tend to hallucinate inputs that appear plausible in code but fail under strict runtime semantics.

Ultimately, progress in PoV generation hinges on the ability to interpret execution outcomes semantically. Bridging the gap between semantic hypothesis and execution reality requires addressing three key challenges:

Challenge 1: Contextual Dependency Retrieval. Vulnerability triggers often depend on non-local context, such as state transitions defined across distant functions or header files. Extracting the minimal yet sufficient code context governing a target constraint without overwhelming the LLM's context window is non-trivial.

Challenge 2: Semantic Gap in Execution Feedback. When a generated input fails, feedback from testing tools is typically low-level (e.g., coverage bitmaps, exit codes, or crash signals). There is a semantic impedance mismatch: translating these opaque binary signals into high-level source constraints that an LLM can understand remains a major bottleneck.

Challenge 3: Satisfaction of Vulnerability Constraints. Reaching the target site is necessary but insufficient for a vulnerability proof. Triggering vulnerabilities often requires satisfying precise and intricate runtime predicates—such as specific heap layouts or global state configurations—that are neither explicit in the local source code nor inferable without execution-aware reasoning.

Our Approach. To this end, we propose DrillAgent, an agentic framework that automates PoV generation by tightly closing the loop between semantic hypothesis and execution-grounded verification. Unlike prior approaches that treat LLMs as one-shot generators, DrillAgent models PoV generation as an iterative *hypothesis–verification–refinement* process. The agent first analyzes the program to hypothesize a candidate input. It then executes the program and employs a novel *Trace-to-Prompt* translator to convert execution traces—specifically failed branch conditions mapped back to source code—into natural language constraints. This execution-grounded feedback enables the agent to debug and refine its own hypotheses, incrementally converging on inputs that satisfy both reachability and vulnerability-triggering conditions.

In summary, this paper makes the following contributions:

- We present DrillAgent, an end-to-end LLM agentic framework that automates Proof-of-Vulnerability generation by tightly coupling semantic reasoning with execution-state feedback. It models PoV generation as an iterative, closed-loop process, enabling the agent to reason about both reachability and vulnerability-triggering conditions in real-world C/C++ programs.
- We introduce an LLM-agent-friendly execution feedback mechanism that bridges the semantic gap between high-level LLM reasoning and low-level program execution. Our design integrates automated code instrumentation, fine-grained coverage feedback, and sanitizer-based crash validation into a unified feedback loop. Execution outcomes are translated into source-level, semantically interpretable constraints, allowing LLMs to iteratively debug failed hypotheses and align their reasoning with concrete runtime behavior.
- We evaluate DrillAgent on SEC-bench, a large-scale benchmark of real-world C/C++ vulnerabilities. Experimental results show that DrillAgent substantially outperforms state-of-the-art LLM agent baselines under fixed budget constraints, improving validated PoV reproduction rates by up to 52.8%, and successfully reproducing a substantial number of previously unresolved benchmark vulnerabilities.

2 Background and Motivation

2.1 Problem Statement

The objective of automated Proof-of-Vulnerability (PoV) generation is to synthesize a concrete input that demonstrates the existence and exploitability of a reported security flaw in a target program.

For the program P, a reported vulnerability is characterized by a tuple $\mathcal{V} = \langle V_{location}, V_{effect} \rangle$, where:

- $V_{location}$ represents the target vulnerability location (e.g., a particular basic block or source line).
- V_{effect} describes the expected effect of the vulnerability (e.g., a crash signal, data corruption, or information leakage).

Let H be a test harness that encapsulates program P and provides the necessary execution environment. Let $I \in \mathcal{I}$ be a candidate input. We denote by $Exec(P, H, I)$ the execution of P under harness H with input I. Validation oracle O determines whether the execution triggers the target vulnerability as specified by $\mathcal{V}$.

PoV Generation Task. The goal is to find an input I such that the execution $Exec(P, H, I)$ satisfies the following conditions, as verified by the oracle O:

(1) Target Reachability: the execution must successfully reach the specified vulnerability location $V_{location}$.

(2) Triggering Effect: upon reaching the target location, the input must trigger the vulnerability with an observable effect V_{effect}. Formally, the oracle evaluates

$$O(P, \mathcal{V}, H, I) \rightarrow \{Success, Failure\}$$

Research Scope. In this work, we focus on PoV generation for real-world vulnerabilities in C/C++ open-source projects. Although current DrillAgent mainly works on memory-related vulnerabilities, it can be extended to logical vulnerability scenarios. We select this scope for two primary reasons. First, C/C++ programs are ubiquitously deployed in critical infrastructure systems, rendering the reproduction of their vulnerabilities highly impactful. Second, memory vulnerabilities in C/C++ leverage mature and robust validation oracles O, particularly memory sanitizers that have undergone extensive practice over time.

2.2 Motivating Example

We use CVE-2023-0760 in the GPAC multimedia framework as a motivating example to illustrate the difficulty of generating end-to-end PoVs. This vulnerability is triggered only after a long execution chain that includes MP4 container parsing, fragment-merging logic, and a comparison helper with unsafe type assumptions. The final crash happens in gf_isom_box_size (Figure 1a), yet the true root cause lies earlier in the call trace: an object originating from the sgpd parsing stage is misinterpreted as a GF_Box due to a blind pointer cast (Figure 1b). This case therefore represents a typical multi-stage semantic bug rather than a local memory error.

Vulnerability details. The type confusion occurs in the helper function shown in Figure 1b. Function gf_isom_is_identical_sgpd accepts two opaque pointers of type void*. When the parameter grouping_type == 0 (the "box mode"), the code directly casts p1 to GF_Box* and invokes box-related routines without any validation. In practice, however, the sgpd parsing logic allocates sample group entry objects whose memory layout differs from that of a real GF_Box. Passing such an group entry object to this branch violates the expected type invariant and creates a confused pointer.

Once this confused pointer reaches the function in Figure 1a, gf_isom_box_size assumes that its argument is a GF_Box and immediately dereferences nested fields such as a->registry->disabled. Because the underlying object is not a box instance, these offsets

```
GF_Err gf_isom_box_size(GF_Box *a) {
  if (!a) return GF_BAD_PARAM;
  if (a->registry->disabled) {    ← Potential heap
    a->size = 0;                     buffer overflow read!
    return GF_OK;
  }
  return gf_isom_box_size_listing(a);
}
```

(a) Potential heap-buffer-overflow read.

```
Bool gf_isom_is_identical_sgpd(void *p1, void *p2,
u32 grouping_type) {
  if (grouping_type) { // entry mode
    sgpd_write_entry(grouping_type, p1, bs1);
  } else {             // box mode
    // Assuming p1 points to a GF_Box object blindly!
    gf_isom_box_size((GF_Box*)p1);
    gf_isom_box_write((GF_Box*)p1, bs1);
  }
}
```

(b) Type confusion due to blind pointer casting.

Figure 1: gpac.cve-2023-0760: vulnerable code snippets.

correspond to memory outside the allocated region of the entry object. The access therefore escapes the object boundary and results in a heap out-of-bounds read.

Triggering this bug requires several non-local conditions to be satisfied simultaneously. The input must be a well-formed fragmented MP4 file containing boxes such as moof, traf, and sgpd so that fragment merging is exercised. During merging, the comparison helper must be invoked with grouping_type == 0, and the pointer must point to an sgpd entry rather than a genuine GF_Box. Only under this specific combination does the type confusion propagate into gf_isom_box_size and materialize as the memory error.

Challenges for general-purpose LLM agents. This case highlights fundamental limitations of general-purpose agents: generating a valid PoV requires reasoning over constraints that span across the codebase and are not explicit in local code contexts.

First, the agent must recover *non-local contextual dependencies*. Reaching the vulnerability requires combining MP4 container semantics (a fragmented file with moof, traf, and sgpd boxes), allocation behavior during sgpd parsing, and fragment-merge logic that eventually invokes the comparison helper. Simple constraint-collection approaches often miss one of these steps, producing either invalid inputs rejected early or valid MP4 files that never exercise the vulnerable path.

Second, normal execution feedback provides little semantic guidance. Common LLM agents typically can only observe program exit code and outputs, including "invalid file," "no fragments," or simply no crash. Translating these superficial signals into actionable insights—such as identifying that fragment merging failed to trigger despite successful parsing—is almost impossible. This task requires more fine-grained execution state feedback to bridge a semantic gap that most general-purpose LLM agents are unable to navigate.

Third, reaching the vulnerable function is not sufficient. To trigger the out-of-bounds read, execution must enter the "box mode"

of the comparison helper while passing a pointer that does not correspond to a real GF_Box. At the same time, the input must be a fragmented MP4 so that the merge logic is executed at all. These precise runtime predicates are difficult to infer from local code alone, making the final crash-triggering step particularly challenging.

How DrillAgent addresses it. DrillAgent successfully generated a validated PoV for this vulnerability in our experiments. During *path exploration*, it learned the required input file structure and produced well-formed fragmented MP4 files that reach the vicinity of the vulnerable code. During *crash triggering*, DrillAgent reused these near-target inputs and focused on satisfying the remaining type-confusion condition, finally triggering the crash in the third refinement round.

In contrast, an OpenHands + Claude Sonnet 4.5 baseline failed to generate a PoV after 74 iterations with unlimited budget. Its attempts oscillated between malformed inputs that fail parsing and valid MP4 files that do not satisfy the fragment-merge constraints needed to reach the unsafe cast. This behavior directly reflects the challenges above: missing non-local dependencies, limited use of execution feedback, and difficulty converging on the final vulnerability-triggering condition.

3 Design

3.1 Overview

Figure 2 shows the end-to-end workflow of DrillAgent. It takes the source code repository and the target vulnerability location as mandatory inputs, with optional auxiliary vulnerability information provided (e.g., a sanitizer report). Through a fully automated LLM-driven reasoning and analysis process, DrillAgent will ultimately output a validated PoV that can reliably trigger the vulnerability crash at the specified location.

DrillAgent consists of four sequential phases. First, in the vulnerability analysis phase (§ 3.2), DrillAgent fully leverages the provided inputs to obtain a deep understanding of the given vulnerability, including its root cause (RC) and the harness execution command (H_{cmd}) required to trigger it. If a sanitizer report is available, DrillAgent further extracts and refines the crash backtrace to derive the V_{trace}. Then, in the code instrumentation phase (§ 3.3), two separate copies of the source code repository are compiled with coverage instrumentation and memory-sanitizer instrumentation respectively, yielding two corresponding binary programs (P_{cov} and P_{san}) for different subsequent purposes. In the path exploration phase (§ 3.4), DrillAgent iteratively attempts to generate test cases that satisfy the input format requirements, with the goal of reaching the vulnerable function at the branch level as much as possible. By executing H_{cmd} on P_{cov}, DrillAgent can extract fine-grained coverage information from the current execution, which provides insights for the next iteration. Finally, in the crash triggering phase (§ 3.5), based on test cases that have already reached or come very close to the crash location, DrillAgent aggressively attempts to generate PoV candidates aiming to trigger the vulnerability crash, and executes H_{cmd} on P_{san} to validate whether they indeed violate memory safety. The PoV generation task is considered successful if the sanitizer reports a crash with a type and location consistent with the ground truth.
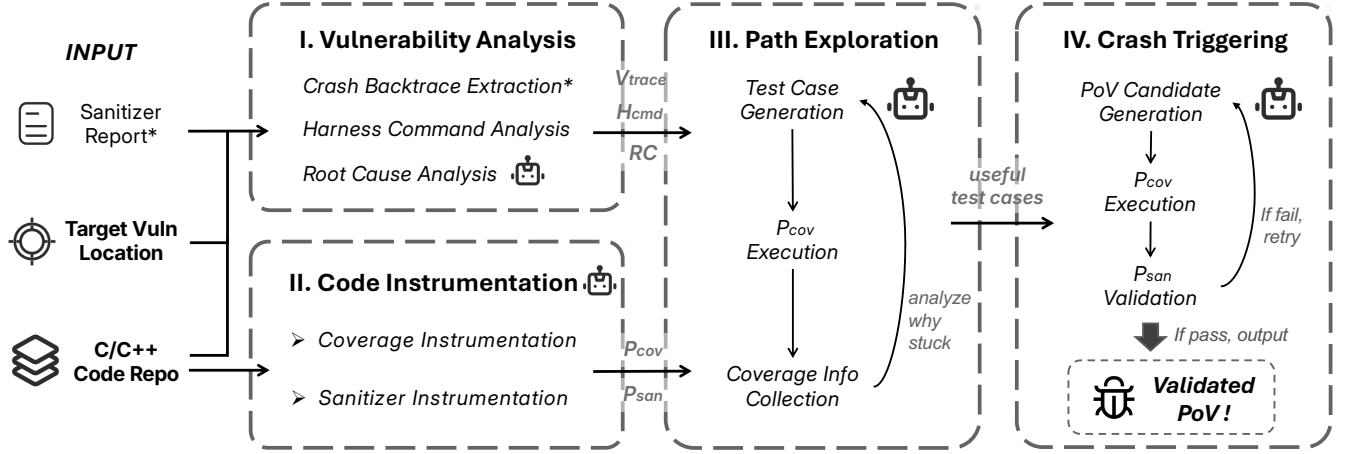
Figure 2: The Overall Workflow of DrillAgent.

Each of the above four phases is realized by a role-specialized sub-agent with a distinct objective and prompt configuration, all coordinated by DrillAgent within a unified execution workflow. For clarity, we refer to these sub-agents as the Vulnerability Analysis Agent (VAAgent), the Code Instrumentation Agent (ITAgent), the Path Exploration Agent (PEAgent), and the Crash Triggering Agent (CTAgent), respectively.

3.2 Vulnerability Analysis

In the first phase of the workflow, we use VAAgent to analyze and interpret the target vulnerability based on all provided inputs, transforming heterogeneous raw information into multiple formatted key elements for PoV generation. The resulting vulnerability-related insights will guide the subsequent contextual dependency retrieval, enabling the agent to focus its attention effectively (addressing Challenge 1). In particular, we focus on three critical facets that jointly constitute the semantic foundation for subsequent reasoning.

Crash Backtrace Extraction. Assuming the provided inputs include a sanitizer report of the target vulnerability, VAAgent extracts the crash backtrace from the program entry location to the crash location and converts it into a JSON format (as shown in the Listing 3). While sanitizer reports typically contain such information, source line numbers are often different between the LLM agent's view and the memory sanitizer's view due to compiler preprocessing and code optimizations. To address this, VAAgent refines the line number of each stack frame in the raw backtrace, ensuring that the source line of frame N accurately represents the call site of the callee function in frame $N + 1$. For heap-related vulnerabilities, we extract not only the crash backtrace but also the allocation and free backtraces, guiding the LLM to better track the execution states of heap memory objects in subsequent analysis.

Notably, the lack of a sanitizer report does not necessarily preclude VAAgent from identifying potential crash backtraces, as long as the target location is provided. Given that many off-the-shelf

```
"crash_trace": [
    {
        "sequence_num": 0,  ❗ crash site
        "function_name": "gf_isom_box_size",
        "file_path": "src/isomedia/box_funcs.c",
        "line_number": 1997
    }, {
        "sequence_num": 1,
        "function_name": "gf_isom_is_identical_sgpd",
        "file_path": "src/isomedia/isom_read.c",
        "line_number": 5865
    },
    ...
]
```

Figure 3: Simplified crash trace generated by the VAAgent.

static analysis tools [18, 24, 38, 42, 52] are proficient at source-to-sink path finding, such capabilities can be integrated into our framework. We leave this for future work.

Harness Command Analysis. Constructing an accurate harness command to trigger the target vulnerability is non-trivial. First, a single codebase may produce multiple CLI binaries, making the mapping between these binaries and the vulnerable code obscure. Second, even when the correct binary is identified, the selection of arguments significantly influences the resulting execution path. We resolves this by identifying potential entry points and locating argument parsing logic and help strings. By analyzing code snippets relevant to the crash backtrace, our agent infers the appropriate harness command for PoV testing. Furthermore, for file-parsing programs, VAAgent also extracts necessary input file extensions to ensure the test case can pass the checks.

Root Cause Analysis. Identifying the root cause (RC) of a vulnerability is fundamental to the PoV generation task, and typically serves as the initial step for manual vulnerability reproduction by security researchers. While existing literature [23, 33, 47–49] leverages symbolic execution or pattern recognition to characterize vulnerability root causes, these methods are often hampered

by limited scalability and heavy overhead. Moreover, their outputs typically consist of a set of condition constraints that lack the high-level semantics required for end-to-end PoV synthesis. In our work, VAAGENT engages in multi-round interactions with the codebase environment based on the analysis results from the preceding two sub-steps. Once all critical code fragments are incorporated into the context, it delivers a comprehensive analysis of the root cause. Specifically, our agent performs root cause analysis from three dimensions: (i) forward reasoning from entry points to derive input format prerequisites; (ii) backward tracing from crash sites to pinpoint the critical conditions that violate security properties; and (iii) deduction from generic vulnerability types to identify instance-specific sub-patterns for the current vulnerability.

After acquiring the comprehensive root cause analysis of a vulnerability, we incorporate these findings as persistent context for all subsequent PoV generation attempts. Equipped with this context, the LLM will no longer reason and solve problems blindly. Instead, it will know which critical contextual dependencies need to be retrieved, thus mitigating Challenge 1. Furthermore, vulnerability analysis insights can also help our agent more intelligently select and examine execution feedback in the latter two phases, thereby improving the task success rate.

3.3 Code Instrumentation

For a given C/C++ codebase, we perform two distinct types of instrumented compilation. The resulting binaries provide semantic-level execution feedback for the subsequent two phases of DRILLAGENT.

In this stage, ITAGENT first extract the basic build procedures and commands from the project documentation (e.g., README.md). Then, it implements the required instrumentation by injecting corresponding compilation parameters into the basic build script. This design ensures high extensibility across different instrumentation types. With prepared build script, ITAGENT iteratively attempts to compile the codebase. If compilation errors are encountered, our agent will collect and diagnose the error messages, refine the compilation commands, and retry the process until successful or a predefined attempt limit is reached, as illustrated in Figure 4.

- For coverage instrumentation, we inject "-fprofile-instr-generate" and "-fcoverage-mapping" flags to leverage LLVM's native source-based code coverage profiling. Unlike the bitmap-based coverage feedback commonly employed by fuzzers, this technique enables a more direct and precise mapping between executed code and source lines, which facilitates the reconstruction of semantic-level execution details.
- For sanitizer instrumentation, we inject "-fsanitize=<type>" and "-fno-omit-frame-pointer" flags. Leveraging the vulnerability type identified in the previous phase, ITAGENT selects the most appropriate sanitizer type (e.g., ASan or UBSan). The resulting compiled binary (P_{san}) is critical for the final validation in the end-to-end PoV generation task.

It is worth noting that a successful compilation does not always guarantee successful instrumentation. For instance, an LLM agent may insert instrumentation flags into the build scripts; however, these flags may later be overridden by subsequently specified other compilation parameters. To resolve this, we introduce lightweight
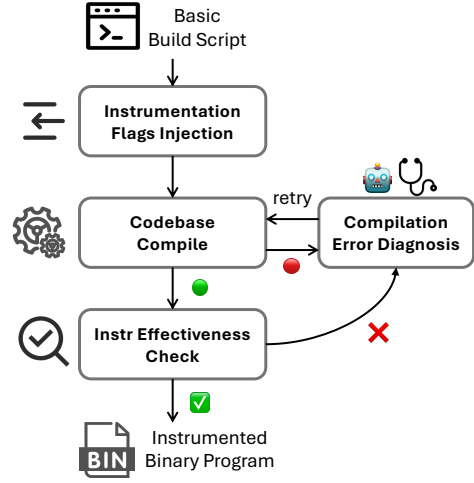

Figure 4: Automated code instrumentation pipeline.

instrumentation effectiveness checks to examine the compiled binary programs. Specifically, we locate the target binary and inspect it for characteristic strings (e.g., "__llvm" or "__sanitizer") that confirm the presence of the intended instrumentation. This design significantly enhances the robustness of the entire analysis process and avoids wasting tokens when no effective execution feedback is available. For the projects whose entry point is a shell script rather than a binary, we also identify the actual binary path to perform this validation.

3.4 Path Exploration

Different from existing LLM-based agents that rely on simplistic "generate-and-test" iterations, DRILLAGENT decouples the PoV generation process into two specialized phases: path exploration and crash triggering. The complete workflow of this collaborative process is formalized in Algorithm 1.

In this phase, PEAGENT adopts an iterative loop with fine-grained execution feedback, together with a well-designed context maintenance mechanism, to enable execution-state-aware LLM reasoning. Specifically, as illustrated in the algorithm 1, PEAGENT first attempts to generate a test case based on the basic vulnerability information and root cause. Then, the generated test case is executed on P_{cov}, from which PEAGENT collects program coverage information (using CollectCov). After updating the context, PEAGENT reviews both the test case generated in the previous iteration and the program's internal execution state before synthesizing the next test case, allowing it to understand the bottlenecks that hinder exploration.

During this iterative process, we additionally check whether each generated test case already covers the last three stack frames of the crash backtrace (using ReachesVulnFunc). If so, the test case is considered useful, as it has reached the vulnerable function with a largely valid execution context, and is added to a set of useful test cases (i.e., useful_tcs). These test cases represent inputs that already pass the program's parsing and basic format checks. As a result, CTAGENT can reuse them as a starting point and concentrate on refining inputs to satisfy vulnerability-specific runtime predicates, rather than repeatedly fixing input well-formedness.

Although the `coverage_query` agent tool built upon llvm-cov is capable of retrieving coverage information at various granularities (e.g., line-level and region-level), guiding an LLM to utilize this capability both stably and effectively remains a significant challenge. To address this, we propose a *hybrid feedback strategy* to balance pre-defined contextual information with the autonomy of the LLM agent. On the one hand, during `CollectCov`, PEAgent deterministically collects and injects coarse-grained, function-level coverage information along the crash backtrace into the context. This step does not require the LLM to generate any tool calls. On the other hand, PEAGent can autonomously query fine-grained, line-level coverage information on demand by explicitly invoking the `coverage_query` tool and inspecting its results. This design allows the LLM to identify where path exploration becomes stalled using a appropriate number of tool calls, enabling more targeted test-case refinement. Ultimately, this hybrid feedback strategy bridges the semantic gap between low-level execution states and high-level LLM reasoning, thereby addressing Challenge 2.

Algorithm 1 Collaboration between PEAgent and CTAgent

Require: Vulnerability info $\mathcal{V}$, Crash backtrace $\mathcal{T}$, Root cause $\mathcal{R}$, Instrumented binary programs P_{cov} and P_{san}
Ensure: Proof-of-Vulnerability $\mathcal{P}$

1: /* **Path Exploration Phase** */
2: $useful\_tcs \leftarrow \emptyset$
3: $context \leftarrow \textsc{UpdateContext}(\mathcal{V}, \mathcal{R})$
4: **for** $i = 1$ **to** N_1 **do**
5:    $tc \leftarrow \textsc{LLM-Generate-Testcase}(context)$
6:    $raw\_profile \leftarrow \textsc{Execute}(P_{cov}, tc)$
7:    $cov\_info \leftarrow \textsc{CollectCov}(raw\_profile)$
8:    **if** $\textsc{ReachesVulnFunc}(cov\_info, \mathcal{T})$ **then**
9:      $useful\_tcs \leftarrow useful\_tcs \cup \{tc\}$
10:    **end if**
11:    $context \leftarrow \textsc{UpdateContext}(tc, cov\_info)$
12: **end for**

13: /* **Crash Triggering Phase** */
14: $vuln\_type, guidance \leftarrow \textsc{SampleVulnTypeHints}(\mathcal{R})$
15: $context \leftarrow \textsc{UpdateContext}(\mathcal{V}, \mathcal{R}, guidance, useful\_tcs)$
16: **for** $j = 1$ **to** N_2 **do**
17:    $pov \leftarrow \textsc{LLM-Generate-PoV}(context)$
18:    $exec\_info \leftarrow \textsc{Execute}(P_{cov}, pov)$
19:    **if** $\textsc{ValidateCrash}(P_{san}, pov) = vuln\_type$ **then**
20:      **return** pov
21:    **end if**
22:    $context \leftarrow \textsc{UpdateContext}(pov, exec\_info)$
23: **end for**
24: **return** null

3.5 Crash Triggering

In the crash triggering phase, CTAgent takes as input the accumulated useful test cases from path exploration and iteratively attempts to construct a PoV that can trigger the target crash. As formalized in Algorithm 1, this phase follows a generate–execute–validate loop. In each iteration, CTAgent synthesizes a PoV candidate from the current context, execute it to observe runtime behavior, and

then validated against the sanitizer-instrumented binary (using `ValidateCrash`). If a PoV candidate produces a sanitizer error consistent with the ground-truth crash, the process terminates successfully. Otherwise, execution feedback is incorporated into the context to guide subsequent refinement, allowing CTAgent to progressively move from near-crashing executions toward a valid PoV.

A key challenge is that the content of a PoV often differs substantially from that of regular input files, making it difficult for the LLM to deviate from its pretraining data and safety-oriented post-training. To mitigate this issue (Challenge 3), CTAgent leverages a *lightweight prompt sampler* that derives vulnerability-type-specific guidance from the root cause, explicitly steering the LLM toward inputs that are more likely to satisfy the vulnerability-triggering runtime predicates. Concretely, `SampleVulnTypeHints` triages the root cause to identify the vulnerability type and then provides corresponding high-level hints that bias the LLM's reasoning toward inputs that are more likely to violate the associated security properties. This enables our agent to adapt its generation strategy across different vulnerability classes without relying on hand-crafted exploit templates.

It is worth noting that the role of execution feedback during crash triggering is different from the path exploration phase. Although the `coverage_query` tool remains available, coverage information is primarily used to help analyze why a PoV candidate fails to trigger the crash, rather than to expand exploration coverage. By selectively querying fine-grained coverage data and correlating it with sanitizer feedback, CTAgent can focus its reasoning on the final obstacles to crash triggering, resulting in more directed and efficient PoV refinement.

4 Implementation

We built DrillAgent from scratch, resulting in a codebase of approximately 7,700 lines of Python code. While several off-the-shelf CLI-based coding agents (such as Cursor CLI [12] and OpenAI Codex [32]) allow for customization, recent research [39, 44, 45, 51] suggests that agents relying solely on LLM-driven autonomous planning often suffer from prompt drift and performance instability when tackling complex tasks. In our implementation, we adopt a pre-structured planning approach, where the four primary phases are executed sequentially in a fixed order. The overall workflow is designed to emulate the common operating procedures followed by human security experts when constructing a PoV. Within each phase, we deploy a dedicated sub-agent that maintains a high degree of autonomy over decision-making and tool invocation within a stabilized work cycle. This hybrid design strikes a strategic balance between the reliability of structured workflows and the generalization capabilities inherent in autonomous agents.

Agent Tools. To facilitate interaction with the target codebase and execution environment, DrillAgent provides a comprehensive suite of tools, namely `read_file`, `write_file`, `execute_bash`, `finish`, `ast_grep_search_function`, `ast_grep_search_pattern`, and `coverage_query`. Specifically, the coverage query tool is built upon the "llvm-cov" and "llvm-profdata" commands. It parses the "default.profraw" files generated after each execution to extract fine-grained, source-mapped coverage information and presents it in an agent-friendly format. Furthermore, our agent's capability to

effectively construct structured binary input data (such as MP4 video streams) is derived from a two-step process: first, it utilizes `write_file` to generate a Python script containing the logic for synthesizing structured binary bytes; subsequently, it invokes `execute_bash` to run the script, thereby producing the final test case or PoV candidate file.

Runtime Cost Control. Many prior studies [14, 22, 55] report that directly using LLM-based agents for test case generation incurs substantial monetary overhead, particularly when generating complex PoVs. In the implementation of DrillAgent, we adopt several techniques to mitigate the runtime cost. First, DrillAgent supports flexible configuration of different LLMs for different sub-tasks. For example, root-cause analysis is performed using a more capable model (e.g., Claude-Sonnet), while crash trace refinement is delegated to a more lightweight and cost-efficient model (e.g., Claude-Haiku). This design optimizes the trade-off between analysis quality and API cost. Second, we impose explicit length limits on outputs returned by each tool invocation, preventing context explosion and reducing unnecessary token consumption. Finally, DrillAgent continuously monitors the cumulative cost during analysis and terminates the process once a predefined budget threshold is exceeded. Together, these design choices make our purely LLM-agent-based approach practical and cost-effective in real-world settings.

5 Evaluation

In this section, we evaluate DrillAgent through the following research questions:

- **RQ1:** How effective is DrillAgent in generating validated PoVs for real-world C/C++ vulnerabilities?
- **RQ2:** What is DrillAgent's runtime overhead?
- **RQ3:** Is root cause analysis critical for our agent to retrieve contextual dependencies efficiently?
- **RQ4:** How does fine-grained execution coverage feedback affect DrillAgent's performance?
- **RQ5:** Is it necessary to decompose PoV generation into two phases (i.e., path exploration and crash triggering)?

5.1 Experimental Setup

Benchmarks. We evaluate our approach on SEC-bench [27], a recent large-scale benchmark of real-world C/C++ CVEs designed for software security tasks. Compared with Magma [19], SEC-bench covers a more diverse set of projects (29 versus 7), which enables a better assessment of the generalization capability of our approach. Besides, SEC-bench provides ready-to-use basic build scripts and raw harness commands, allowing the agent to focus more on the PoV generation task itself.

The original SEC-bench PoV-generation suite contains 200 CVE tasks. We exclude 10 tasks whose sanitizer reports provide only raw binary addresses without source-line information in the crash backtraces. This results in two benchmark sets used throughout this paper:

- **SEC-bench-Full (190 tasks).** The full evaluation set after filtering the mentioned 10 tasks. We report DrillAgent's end-to-end PoV generation performance on this set to characterize overall effectiveness in a comprehensive manner.

- **SEC-bench-60 (60 tasks).** A uniformly random subset sampled from SEC-bench-Full using a fixed random seed (42), whose corresponding task IDs will be released as part of our artifact. We use this subset for head-to-head comparisons with baselines to control evaluation cost while preserving a fair, fixed task set for reproducibility.

Baselines. We compare DrillAgent against **OpenHands** [43], a state-of-the-art LLM-based agent framework that demonstrates strong performance on SEC-bench. OpenHands represents a general-purpose agentic system capable of repository navigation, code inspection, tool invocation, and iterative reasoning, and has been shown to be effective on a wide range of software engineering and security tasks. We select OpenHands as our primary baseline as it embodies the dominant end-to-end paradigm for PoV generation, serving as a representative contrast to our task-specialized, feedback-driven design.

Model Configuration. To ensure a fair comparison, we evaluate both DrillAgent and the baseline agent using the same underlying LLM whenever applicable. Specifically, unless otherwise noted, we run both systems with **Claude Sonnet 4.5**, replacing the originally reported model used in OpenHands. This setup isolates the effect of agent design from differences in model capability. Within DrillAgent, we additionally allow different sub-agents to employ different LLM variants for cost–quality trade-offs (Section 4), but this configuration is fixed across all evaluation runs and does not affect baseline comparisons.

Metrics. Our evaluation follows the PoV validation protocol defined by SEC-bench, which determines PoV validity based on sanitizer-reported crash outputs under the benchmark-provided execution configuration. Instead of treating PoV generation as a binary success-or-failure task, we explicitly distinguish between different classes of all sanitizer-triggering test cases. Given an input that produces a sanitizer-reported crash, we categorize it as one of the following:

- **Validated PoV.** An input whose execution triggers a sanitizer crash that matches the *ground-truth vulnerability specification* provided by SEC-bench, including both the expected crash type and the target crash location.
- **Variant PoV.** An input that triggers a sanitizer-reported crash but does *not* match the ground-truth vulnerability type or location. Such inputs may expose other related memory safety violations or alternative crash sites, which also carry substantial implications for real-world security practices.

We record both validated and variant PoVs generated by DrillAgent. Validated PoVs represent faithful reproductions of the target vulnerabilities, while variant PoVs serve as a complementary signal of the agent's ability to explore and trigger diverse sanitizer-reported failure modes.

5.2 RQ1: Effectiveness

Table 1 summarizes the end-to-end PoV generation effectiveness of DrillAgent and the baseline agent OpenHands on SEC-bench. Overall, DrillAgent demonstrates a clear advantage in generating validated Proofs-of-Vulnerability under a fixed budget constraint. **Overall PoV Generation Performance.** On the full benchmark (SEC-bench-Full), DrillAgent successfully generates validated

Table 1: End-to-End PoV Generation Effectiveness (RQ1)

Method	Benchmark	Budget	Validated PoVs	Variant PoVs	Total Crashes	Resolved Rate[†]	Crash Rate
DrillAgent	SEC-bench-Full	$1.5/task	**55**	12	**67**	28.9%	**35.3%**
DrillAgent	SEC-bench-60	$1.5/task	15	2	17	25.0%	28.3%
OpenHands	SEC-bench-60	$1.5/task	6	–	6	10.0%	10.0%
OpenHands	SEC-bench-60	unlimited	18	–	18	30.0%	30.0%

[†] Resolved Rate is calculated as *Validated PoVs* divided by the total number of tasks in the current benchmark.
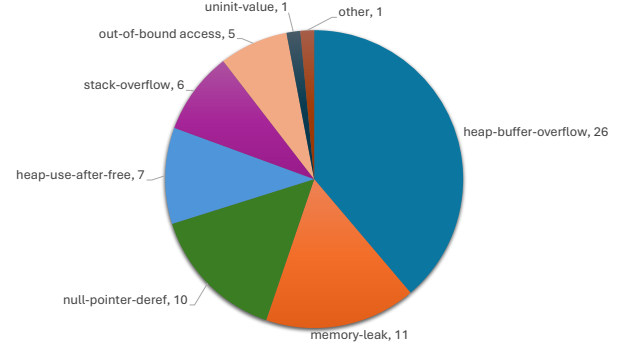Our number of *Validated PoVs* is 52.8% higher than the current state-of-the-art result [36].

PoVs for 55 out of 190 tasks, achieving a success rate of 28.9%. As of now, the highest reported resolved rate on the SEC-bench leaderboard [36] is 18% by OpenHands + Claude Sonnet 3.7, corresponding to 36 solved tasks. In comparison, **DRILLAGENT improves upon this state-of-the-art result by 52.8%**. Moreover, among the 55 tasks that DRILLAGENT successfully solves and validates, 29 are uniquely solved (i.e., they have not been successfully addressed by any prior methods or evaluations). This result highlights the distinct advantages and complementary strength of our approach.

In addition to the 55 validated PoVs, DRILLAGENT also generated 12 variant PoVs, which mainly trigger sanitizer-detected crashes with different error types (e.g., producing a heap buffer overflow instead of the originally expected memory leak). Although these variant PoVs are not counted as successful reproductions under the SEC-bench evaluation protocol, they demonstrate the agent's ability to reach semantically relevant program states and uncover related failure patterns. To capture this capability beyond strict vulnerability resolution, we additionally report the *Crash Rate*, on which DRILLAGENT achieves an impressive 35.3%. Such PoV variants also carry practical security value: in the AIxCC competition [13], variants that trigger the target vulnerability via different crash paths can receive additional credit, as they may bypass specific patches.

Comparison with OpenHands under Controlled Budget. On the SEC-bench-60 subset, DRILLAGENT consistently outperforms the OpenHands baseline under the same Claude Sonnet 4.5 model and budget constraint of $1.5 per task. Specifically, our approach generates more than twice as many validated PoVs as OpenHands (15 vs. 6), highlighting the effectiveness of its execution state feedback design. Notably, when the budget constraint is removed, OpenHands achieves a higher absolute success rate; however, this comes at the cost of unbounded runtime and monetary overhead, which will be discussed in the next section.

Breakdown of Solved Tasks and Execution Behavior. To gain a deeper understanding of the success of DRILLAGENT, we conduct a more fine-grained statistical analysis over the evaluation results.

Among the 67 tasks on which DRILLAGENT successfully triggered crashes, these tasks span 13 unique C/C++ projects, nearly half of the 29 projects included in SEC-bench, demonstrating that the success is not concentrated on a few specific codebases. The generated PoVs cover a wide spectrum of input formats: some require syntactically valid programming language scripts (e.g., mruby and njs), while others involve highly structured binary multimedia files (e.g., gpac and imagemagick). This diversity demonstrates the strong generalization capability of our approach across heterogeneous application domains and input formats.

As shown in Figure 5, the types of vulnerabilities successfully handled by DRILLAGENT also broadly cover those present in the benchmark, including stack overflow, heap buffer overflow, use-after-free, null pointer dereference, and memory leak. This indicates that the effectiveness of DRILLAGENT is not limited to a specific vulnerability category, but extends to a wide range of memory safety issues.

Figure 5: Vulnerability-Type Distribution of Solved Tasks.

Furthermore, to better understand the execution behavior of DRILLAGENT, we analyze its tool usage over all 190 tasks. On average, each end-to-end task requires 60 LLM interactions and 100.4 tool invocations. The detailed distribution of tool call types is illustrated in Figure 6. We observe that the majority of tool calls are devoted to source code reading, which is consistent with the fact that the input tokens consumed in the PoV generation task far exceed the output tokens (Table 3). Although the coverage_query tool is invoked only 7.8 times per task on average, this number is biased downward by tasks that generate a PoV in a single attempt or fail during instrumented compilation. In practice, coverage feedback is crucial for the success of many tasks, as will be further demonstrated in Section 5.4.2.

Summary Findings. Overall, DRILLAGENT substantially improves end-to-end PoV generation effectiveness under fixed budget constraints. The results on SEC-bench demonstrate that our approach not only outperforms a strong LLM-agent baseline, but also advances the current best resolved rate while reproducing a considerable number of previously unresolved vulnerabilities. These gains highlight the importance of execution-state-aware feedback for reliable and scalable PoV reproduction.
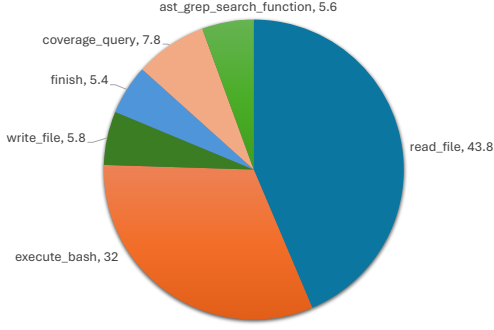
Figure 6: Average Tool Call Count per Task.

5.3 RQ2: Efficiency

We first analyze the monetary cost of our approach, as shown in Table 2. On SEC-bench-Full, DrillAgent incurs an average cost of $1.79 per task under a nominal budget of $1.5/task. This slight overrun is expected, as DrillAgent does not enforce a hard cutoff at the budget limit; instead, it guarantees the completion of at least one full test-case generation and validation cycle to avoid premature termination. Despite this conservative design choice, the overall cost remains tightly controlled. On the SEC-bench-60 subset, DrillAgent exhibits comparable behavior, with an average cost of $1.93 per task, remaining close to the intended budget while substantially outperforming the OpenHands baseline in terms of validated PoV generation.

A more revealing comparison emerges by calculating the average cost spent on each successfully resolved task. On SEC-bench-60, DrillAgent achieves a cost of $7.72 per validated PoV, which is significantly lower than OpenHands under both budgeted ($15.30) and unlimited ($20.13) settings. Although OpenHands solves slightly more tasks when allowed unlimited budget, this improvement comes at a disproportionate monetary cost. In contrast, DrillAgent delivers a substantially better cost–effectiveness trade-off, demonstrating that execution-state-aware reasoning enables more efficient convergence toward valid PoVs rather than relying on unconstrained exploration.

We next examine execution time overhead. As shown in Table 2, DrillAgent requires approximately 11–12 minutes per task, which is higher than OpenHands but remains within a practical range for automated vulnerability reproduction. Table 3 further indicates that this time is distributed across multiple reasoning-intensive phases. Importantly, this overhead is modest compared to traditional fuzzing-based approaches, which often require hours of execution to reproduce deep, logic-heavy vulnerabilities. For such cases, our method offers a more time-efficient alternative while maintaining deterministic PoV validation.

5.4 Ablation Study

To understand which components are critical to DrillAgent's effectiveness, we conduct an ablation study that systematically disables individual mechanisms in the full system. Considering the cost constraints, we perform ablation experiments on **SEC-bench-Ablation-30**, a carefully selected 30-task subset of SEC-bench-Full containing only tasks successfully solved by DrillAgent. To

maximize the diagnostic value of the ablation results, we prioritize tasks that are uniquely solved by DrillAgent, as these cases better highlight the contributions of individual design components.

5.4.1 RQ3: Impact of Root Cause Analysis. To evaluate the role of root cause analysis (RCA), we remove all root-cause-related context from DrillAgent and only retain the crash location and basic vulnerability information. As shown in Figure 7, disabling RCA leads to a clear reduction in the number of validated PoVs, indicating that RCA is critical for effective PoV generation.

This result suggests that, for PoV synthesis, high-level root cause descriptions are more effective than rigidly collecting all low-level program constraints. RCA enables the agent to focus on semantically relevant contextual dependencies and guides efficient context retrieval, thus mitigating Challenge 1. This also partially explains why prior approaches such as FaultLine [31] exhibit weaker performance on C/C++ programs, where constraint-heavy reasoning without high-level abstraction becomes brittle and impractical.

5.4.2 RQ4: Impact of Execution State Feedback. To study the impact of execution state feedback, we disable all coverage query capabilities in PEAgent and CTAgent, making the agent execution-state-unaware. As illustrated in Figure 7 (refer to `w/o ExecFB`), this ablation also causes severe performance drop.

Without execution feedback, the agent actually reasons "blindly" and lacks actionable signals to understand why a generated input fails. As a result, it becomes difficult to simultaneously satisfy path reachability and vulnerability-triggering constraints, confirming that execution-state-aware feedback is essential for grounding LLM reasoning in concrete program behavior.

5.4.3 RQ5: Impact of Crash-Triggering Guidance. For this research question, we remove the dedicated crash-triggering guidance by disabling CTAgent and integrating part of its functionality into PEAgent. As shown in Figure 7, the resolved rate decreases, but the degradation is less severe compared to other ablations.

This indicates that while explicit crash-triggering guidance is not strictly required for reachability, it plays an important role in the final vulnerability-triggering stage. In particular, the prompt sampler mentioned in Section 3.5 helps bias the LLM toward security-violating behaviors, alleviating the difficulty of triggering crashes beyond mere path exploration.
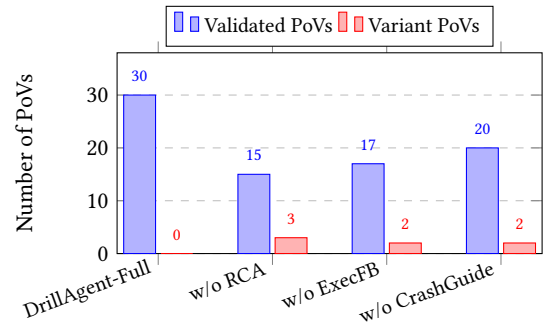

Figure 7: Ablation study on key DrillAgent design components. Each variant disables one component while keeping all other settings fixed.

Table 2: End-to-End Task Cost and Execution Time (RQ2)

Method	Benchmark	Budget	Cost/Task ($)	Cost/Success* ($)	Exec Time (min)
DrillAgent	SEC-bench-Full	$1.5/task	1.79	**6.18**	11.5
DrillAgent	SEC-bench-60	$1.5/task	1.93	**7.72**	11.8
OpenHands	SEC-bench-60	$1.5/task	1.53	15.30	3.2
OpenHands	SEC-bench-60	unlimited	6.04	20.13	7.7

*Cost / Success is computed as the total monetary cost divided by the number of validated PoVs. And the three above metrics are averaged over all tasks.

Table 3: Phase-wise Time and Token Usage Breakdown of DRILLAGENT on SEC-bench-Full

Phase	Avg. Time (min)	Input Tokens	Output Tokens
Vulnerability Analysis	2.5	*228,145*	*8,147*
Code Instrumentation	1.9	*-**	*-**
Path Exploration	3.7	*195,908*	*12,748*
Crash Triggering	3.3	*171,438*	*11,584*

*Token usage during the code instrumentation phase is omitted, as it involves few LLM interactions and no tool calls.

6 Discussion

6.1 Comparison with Directed Fuzzing

Directed greybox fuzzing (DGF) is fundamentally designed to optimize reachability objectives, typically by minimizing a distance metric between runtime states and a target program location, and has proven effective for coverage-oriented vulnerability discovery. However, PoV reproduction is a more goal-directed task that requires not only reaching a vulnerable location, but also satisfying a conjunction of semantic feasibility constraints—often involving non-trivial data dependencies, protocol invariants, or stateful checks—that are not explicitly encoded in the control-flow graph. Importantly, our method does not attempt to replace fuzzing nor to enhance it with more sophisticated heuristics. Instead, it reformulates PoV generation as an interactive hypothesis–verification process: the agent iteratively proposes candidate inputs, observes concrete execution feedback and failures, and refines its hypotheses based on semantically grounded feedback. This process is closer to program debugging than to stochastic exploration, and therefore addresses a complementary problem setting to fuzzing.

6.2 Threats to Validity

Internal Validity. *System implementation correctness* is a potential threat, as DRILLAGENT integrates multiple sub-agents and interacts with a complex execution environment (e.g., coverage queries and Python scripts). However, the correctness of our reported results ultimately depends on whether a generated PoV triggers a sanitizer-confirmed crash. This final validation is performed using a well-tested sanitizer-instrumented binary (P_{san}) and guarded by the lightweight "Instr Effectiveness Check" (Figure 4). Consequently, implementation bugs in earlier components would at worst lead to PoV generation failures rather than false positives. We further manually inspected most generated PoVs and their execution logs.

Uncontrolled randomness may also introduce variability in outcomes. We mitigate this by using a low temperature (0.1) for most LLM interactions, and a higher temperature (0.7) only during test

case and PoV generation in Phases III and IV to encourage diversity. However, these stages typically involve N iterative rounds that generate multiple input candidates, which helps amortize the impact of randomness across iterations. Overall, DRILLAGENT's pre-structured workflow and timely intermediate validation reduce the impact of nondeterministic behavior in the agentic system.

External Validity. A potential threat to external validity is *LLM data leakage*, since SEC-bench consists of public CVEs for which PoVs may already exist online, raising the risk that the backbone model reproduces memorized solutions rather than synthesizing new ones. We provide two qualitative indications that this effect is limited. First, under the same backbone model, our agent solves substantially more vulnerabilities than the OpenHands baseline, suggesting gains from the agent design rather than latent recall. Second, as shown in Table 4, our dual-metric similarity analysis shows low overlap between generated and ground-truth PoVs (average score 0.0296, max 0.1320 across 24 successfully solved CVEs), indicating that our system produces structurally and lexically distinct PoVs rather than reproducing reference implementations.

Another potential threat arises from *benchmark quality*. For example, the sanitizer report for imagemagick.cve-2019-13301 contains multiple types of crashes with several distinct backtraces, which may complicate the final triage and validation process. In such cases, validated PoVs may be confused with variant PoVs that trigger different but related crashes. However, this issue does not affect the core experimental outcomes, as all reported successes are still grounded in sanitizer-confirmed crashes consistent with the benchmark specification.

Regarding supported *vulnerability types*, our evaluation currently focuses primarily on memory vulnerabilities. Nevertheless, DRILLAGENT is not inherently limited to this class. It can be extended to other C/C++ vulnerability types, such as path traversal or command injection, by adapting the crash-triggering prompts and replacing the final PoV validation environment accordingly, following the approach used in FaultLine [31].

Table 4: PoV Similarity Between DRILLAGENT Generated and Ground-Truth PoVs

Statistic	GramSim	ChunkSim	Score
Average	0.0413	0.0022	0.0296
Max	0.1886	0.0479	0.1320
Min	0.0000	0.0000	0.0000

Note: Higher values indicate a greater degree of similarity between the generated and ground-truth PoVs.

7 Related Work

LLM-agent-based Methods. Recent studies have explored the use of LLM-based agentic systems for various PoV generation tasks. For vulnerability reproduction in C/C++ programs, PBFuzz [55] guides LLMs to generate parameterized input generators and randomly samples the constrained input space to produce PoVs. However, the generated input generators suffer from construct validity bias. They also struggle to craft malformed inputs in certain cases, a task that pure LLM-agent approaches can accomplish with much greater flexibility. Another recent work [35] proposes a similar framework that provides coverage feedback to LLM agents to better address the reachability gap. Nevertheless, its agent focuses solely on reaching target location in function-level, while subsequent crash triggering still relies on existing fuzzers [17]. In contrast, DRILLAGENT also integrates coverage feedback into LLM reasoning to resolve vulnerability-triggering constraints and supports end-to-end PoV generation, making the two methodologies fundamentally different. FaultLine [31] does not target memory vulnerabilities; instead, it proposes an agentic workflow to generate PoVs for taint-style vulnerabilities in C/C++ programs (e.g., command injection). However, it merely collects low-level path constraints and directly delegates them to LLMs for solving, which overlooks high-level vulnerability semantics.

Several other agent-based works have also explored the more generalized PoV generation task, but their focus differs from the research scope of this paper. EnIGMA [1] builds upon the SWE-agent [50] to solve Capture The Flag (CTF) challenges, enabling interaction with both a debugger and a server connection tool. PwnGPT [34] is an LLM-based framework for automatic exploit generation targeting CTF pwn challenges, which partially overlaps with the PoV generation task. However, synthetic CTF tasks are generally less complex than real-world CVEs, and other categories of CTF challenges (e.g., reverse and crypto) have limited relevance to PoV generation for C/C++ vulnerabilities. PoCGen [40] can generate and validate PoC exploits for vulnerabilities in npm packages with the help of LLM, while we focus on different programming language. There are also research efforts [30, 56, 57] that investigate the reproduction and exploitation of web vulnerabilities, which are substantially different from memory vulnerabilities.

Fuzzing-based Methods. Fuzzing-based methods[16, 54], especially directed greybox fuzzing[5, 7, 9, 10, 20, 26, 29, 53], have also been widely adopted to address PoV generation tasks. SDFuzz [29] introduces a target states-driven directed fuzzing approach that derives high-value program states from vulnerability descriptions to guide seed exploration, combined with selective instrumentation to prune irrelevant execution and accelerate reaching targets.

Lyso [5] leverages correlations among static-analysis alarms and decomposes each target into multiple semantic steps to guide fuzzing toward true positives across multiple related vulnerability sites. However, recent studies [5, 21, 55] reveal that state-of-the-art undirected general fuzzers (e.g., AFL++) can even outperform directed fuzzers on the given vulnerability benchmarks such as Magma [19]. This is partly because the distance metrics adopted by current fuzzing methods cannot perfectly capture the actual distance to vulnerable states, due to inherent limitations of static program analysis. For example, AFLGopher [4] points out that the distance calculation in most existing approaches is feasibility-unaware, which may even mislead the fuzzer with inaccurate feedback. In contrast, DRILLAGENT mimics the behavior of human vulnerability analysts by reasoning solely based on faithful execution feedback rather than potentially imprecise distance metrics, thereby enabling effective PoV construction.

Several works [3, 28, 41] combine fuzzing with symbolic execution to satisfy the constraints required to reach specific vulnerable locations. However, inherent issues of symbolic execution (e.g., state explosion) may limit its scalability. Recent work [6, 15, 55] has also leveraged large language models to enable more semantics-aware directed fuzzing, for example by using LLMs to guide distance metrics or to synthesize reachable seeds and mutators. In future work, we plan to explore the integration of DRILLAGENT with traditional AFL-style fuzzing techniques, such as performing further fuzzing based on the test cases generated by our LLM agent.

AIxCC CRSs DARPA's Artificial Intelligence Cyber Challenge (AIxCC) [13] has attracted significant attention and participation from both academia and industry. In the final round, seven teams were required to each submit a cyber reasoning system (CRS) that autonomously performs the end-to-end pipeline of vulnerability discovery, triggering, and patching to earn scores. These CRSs appear closely related to the PoV generation task studied in this paper. However, directly applying them to our problem setting is nontrivial. First, in the AIxCC competition, vulnerability locations are not provided in advance, and most finalist teams [8, 25, 37] rely on undirected fuzzing techniques such as AFL++. In these CRSs, vulnerability discovery and PoV generation are tightly coupled and performed simultaneously by the fuzzing engine. This design makes them unsuitable for specialized PoV generation tasks that require triggering vulnerabilities of specific types at designated locations. Second, the cost overhead significantly limits the practicality of AIxCC CRSs in real-world scenarios. In the final competition, organizers provided each team with $50,000 in LLM API credits and $85,000 in Azure compute resources [25]. Therefore, although Team Theori's CRS appears to be directly applicable to the vulnerability reproduction task studied in this paper, as it adopts a pipeline of static analysis to locate potential vulnerabilities followed by LLM-based PoV generation [46], it remains difficult to deploy in practice due to cost constraints.

Consequently, the exploration of effective yet cost-efficient solutions still remains an open and worthwhile research direction.

8 Conclusion

This paper presents DRILLAGENT, an agentic framework that addresses the challenge of automated PoV generation by bridging the

gap between static semantic reasoning and dynamic execution. Unlike prior approaches that rely on blind mutation or open-loop generation, DRILLAGENT formulates the task as an iterative hypothesis–verification–refinement process. By systematically translating low-level execution failures into source-level semantic constraints, our approach enables LLMs to effectively reason over both path reachability and complex vulnerability-triggering conditions. Our evaluation on the SEC-bench dataset demonstrates that this execution-grounded design substantially outperforms existing LLM-based baselines under fixed budget constraints, improving validated PoV reproduction rates by up to 52.8% and successfully reproducing many vulnerabilities that were previously unresolved.

References

[1] Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberly Milner, Sofija Jancheska, John Yang, Carlos E Jimenez, Farshad Khorrami, et al. 2025. EnIGMA: Interactive Tools Substantially Assist LM Agents in Finding Security Vulnerabilities. In *Forty-second International Conference on Machine Learning*.

[2] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. *arXiv preprint arXiv:2502.05368* (2025).

[3] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 641–652.

[4] Weiheng Bai, Kefu Wu, Qiushi Wu, and Kangjie Lu. 2025. AFLGopher: Accelerating Directed Fuzzing via Feasibility-Aware Guidance. *arXiv preprint arXiv:2511.10828* (2025).

[5] 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 *34th USENIX Security Symposium (USENIX Security 25)*. 6977–6997.

[6] Wang Bin, Ao Yang, Kedan Li, Aofan Liu, Hui Li, Guibo Luo, Weixiang Huang, and Yan Zhuang. 2025. Attention Distance: A Novel Metric for Directed Fuzzing with Large Language Models. *arXiv preprint arXiv:2512.19758* (2025).

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[8] Trent Brunson. 2025. *AIxCC finals: Tale of the tape.* https://blog.trailofbits.com/2025/08/07/aixcc-finals-tale-of-the-tape/ Blog post on the Trail of Bits blog covering DARPA's AI Cyber Challenge (AIxCC) finals and the diverse approaches of its finalists.

[9] 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 *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2095–2108.

[10] Yiyang Chen, Chao Zhang, Long Wang, Wenyu Zhu, Changhua Luo, Nuoqi Gui, Zheyu Ma, Xingjian Zhang, and Bingkai Su. 2025. {IDFuzz}: Intelligent Directed Grey-box Fuzzing. In *34th USENIX Security Symposium (USENIX Security 25)*. 6219–6238.

[11] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic bug reproduction for effective automated program repair at google. *arXiv preprint arXiv:2502.01821* (2025).

[12] Cursor AI. 2024. Cursor CLI: An LLM-Powered Coding Agent. https://cursor.com/cli. Accessed: 2026-01.

[13] DARPA AI Cyber Challenge. 2023–2025. *AI Cyber Challenge.* https://aicyberchallenge.com/ Official website of the Artificial Intelligence Cyber Challenge (AIxCC), a two-year competition organized by DARPA and ARPA-H to advance autonomous AI systems for discovering and patching software vulnerabilities.

[14] 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 *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery.

[15] 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).

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[17] Octavio Galland and Marcel Böhme. 2025. Invivo Fuzzing by Amplifying Actual Executions. In *Proceedings of the 47th International Conference on Software Engineering (ICSE'25)*.

[18] GitHub. 2026. CodeQL: The static analysis engine used by developers to identify security vulnerabilities and by security researchers to perform variant analysis. https://codeql.github.com/.

[19] 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.

[20] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.

[21] Madonna Huang and Caroline Lemieux. 2024. Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop*. 33–41.

[22] Kush Jain and Claire Le Goues. 2025. TestForge: Feedback-Driven, Agentic Test Suite Generation. *arXiv preprint arXiv:2503.14713* (2025).

[23] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash deduplication through root-cause clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3318–3336.

[24] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1695–1708.

[25] Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R Jeong, Dohyeok Kim, Dongkwan Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, et al. 2025. ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System. *arXiv preprint arXiv:2509.14589* (2025).

[26] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. 3559–3576.

[27] Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. 2025. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. *arXiv preprint arXiv:2506.11791* (2025).

[28] Jia Li, Jiacheng Shen, Yuxin Su, and Michael R Lyu. 2025. ColorGo: Directed Concolic Execution. *arXiv preprint arXiv:2505.21130* (2025).

[29] Penghui Li, Wei Meng, and Chao Zhang. 2024. {SDFuzz}: Target States Driven Directed Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2441–2457.

[30] Bin Liu, Yanjie Zhao, Guoai Xu, and Haoyu Wang. 2025. LLM Agents for Automated Web Vulnerability Reproduction: Are We There Yet? *arXiv preprint arXiv:2510.14700* (2025).

[31] Vikram Nitin, Baishakhi Ray, and Roshanak Zilouchian Moghaddam. 2025. FaultLine: Automated Proof-of-Vulnerability Generation Using LLM Agents. *arXiv preprint arXiv:2507.15241* (2025).

[32] OpenAI. 2021. OpenAI Codex. https://openai.com/codex/. Accessed: 2026-01.

[33] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. 2024. Benzene: A practical root cause analysis system with an under-constrained state mutation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1865–1883.

[34] Wanzong Peng, Lin Ye, Xuetao Du, Hongli Zhang, Dongyang Zhan, Yunting Zhang, Yicheng Guo, and Chen Zhang. 2025. PwnGPT: Automatic Exploit Generation Based on Large Language Models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 11481–11494.

[35] Gaetano Sapia and Marcel Böhme. 2026. Scaling Security Testing by Addressing the Reachability Gap. In *International Conference on Software Engineering (ICSE)*.

[36] SEC-Bench Team. 2025. SEC-bench LeaderBoard. https://sec-bench.github.io/. Accessed: 2026-01-15.

[37] 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).

[38] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706.

[39] Yuchen Shi, Siqi Cai, Zihan Xu, Yuei Qin, Gang Li, Hang Shao, Jiawei Chen, Deqing Yang, Ke Li, and Xing Sun. 2025. Flowagent: Achieving compliance and flexibility for workflow agents. *arXiv preprint arXiv:2502.14345* (2025).

[40] Deniz Simsek, Aryaz Eghbali, and Michael Pradel. 2025. PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages. *arXiv preprint arXiv:2506.04962* (2025).

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[42] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[43] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[44] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).

[45] Ruixuan Xiao, Wentao Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. 2024. Flowbench: Revisiting and benchmarking workflow-guided planning for llm-based agents. *arXiv preprint arXiv:2406.14884* (2024).

[46] Xint. 2025. *AI Cyber Challenge and Theori's RoboDuck.* https://theori.io/blog/aixcc-and-roboduck-63447 Theori blog post, an introduction to DARPA's AI Cyber Challenge (AIxCC) and Theori's third place cyber reasoning system RoboDuck.

[47] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. 2024. Racing on the Negative Force: Efficient Vulnerability {Root-Cause} Analysis through Reinforcement Learning on Counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4229–4246.

[48] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 320–336.

[49] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. {ARCUS}: symbolic root cause analysis

[50] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[51] Kang Yang, Yunhang Zhang, Zichuan Li, Guanhong Tao, Jun Xu, and Xiaojing Liao. 2025. HarnessAgent: Scaling Automatic Fuzzing Harness Construction with Tool-Augmented LLM Pipelines. *arXiv preprint arXiv:2512.03420* (2025).

[52] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A fused approach to path-sensitive sparse data dependence analysis. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 567–592.

[53] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2139–2154.

[54] Michal Zalewski. 2014. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/.

[55] Haochen Zeng, Andrew Bao, Jiajun Cheng, and Chengyu Song. 2025. PBFuzz: Agentic Directed Fuzzing for PoV Generation. *arXiv preprint arXiv:2512.04611* (2025).

[56] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, et al. 2025. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities. *arXiv preprint arXiv:2503.17332* (2025).

[57] Yuxuan Zhu, Antony Kellermann, Akul Gupta, Philip Li, Richard Fang, Rohan Bindu, and Daniel Kang. 2024. Teams of llm agents can exploit zero-day vulnerabilities. *arXiv preprint arXiv:2406.01637* (2024).

of exploits in production systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 1989–2006.