WHEN FUZZING BECOMES AGENTIC: SEMANTIC STATE EXPLORATION IN THE WILD

Andrew Yin[1] **Zhaoling Chen**[2] **Qian Zhang**[2] **Heng Yin**[2]
[1]University of California, San Diego [2]University of California, Riverside
a1yin@ucsd.edu zchen526@ucr.edu {qzhang,heng}@cs.ucr.edu

ABSTRACT

Agentic AI systems increasingly navigate through stateful, rule-bounded software programs, where meaningful behaviors surface only after long sequences of valid, context-aware interactions. Current evaluation methods are often outcome-focused and provide limited visibility into whether an agent has reached novel internal states. Traditional automated testing, however, frequently assumes a stateless environment and generates invalid interactions that violate consistency constraints. We propose ASA-Fuzz, a closed-loop agentic testing framework for stateful programs that treats software fuzzing as an agentic, decision making problem. This framework (1) infers and instruments state-relevant runtime signals to provide a semantic notion of progress, (2) synthesizes context-aware operators to generate valid multi-step interactions conditioned on observed state, and (3) is designed to adapt when exploration plateaus. In a case study on the rule-based, stateful chess environment, ASA-Fuzz discovered over two orders of magnitude more unique board states and achieved a $3.5\times$ to $12\times$ greater mean effective move depth than industry-standard (AFL++), state-aware (SGFuzz), and LLM-driven (Fuzz4All) baselines.

1 INTRODUCTION

Agentic AI systems increasingly operate in the wild by taking actions through *stateful* and *rule-bounded* software environments and tools, such as decision-making systems, services, and protocol-driven programs. In such environments, the most consequential failures are often inherently *long-horizon*: they emerge only after many valid moves that steer the agents into a rare internal configuration. Safe deployment of agents therefore requires systematic testing that can reach deep, dependent states under strict constraints.

Today's agent evaluation is largely outcome-focused and relies mostly on surface signals, such as rewards or coverage-like proxies. These signals do not indicate whether an agent has reached new internal situations in a stateful environment. Traditional automated testing, on the other hand, is similarly mismatched. It often assumes a stateless environment and generates action sequences without modeling the environment's rules. In practice, an agent's action success depends on hidden states that evolve across steps, and naive actions frequently violate formatting and consistency constraints, leading to early rejection. This leads to a plateau of rapid progress on shallow behaviors, followed by stalling on rare, state-dependent situations where meaningful failures occur in the wild.

To address this gap, we propose **ASA-Fuzz** (Agentic State-Aware Fuzzing), a *closed-loop agentic testing framework* for stateful programs. The key idea is to treat testing as an agent problem: learn what state matters, generate valid actions that induce meaningful state transitions, and adapt their strategies when exploration stalls. Rather than relying on fixed heuristics or static feedback signals, our system employs multiple cooperating agents that continuously reshape both their observation space and action space during testing.

- **State Abstraction**: Rather than relying on fixed observation spaces, analysis agents autonomously reason over the target source code to identify semantic state variables. The instrumentation agent explicitly modifies the environment through source code instrumentation to construct a dynamic

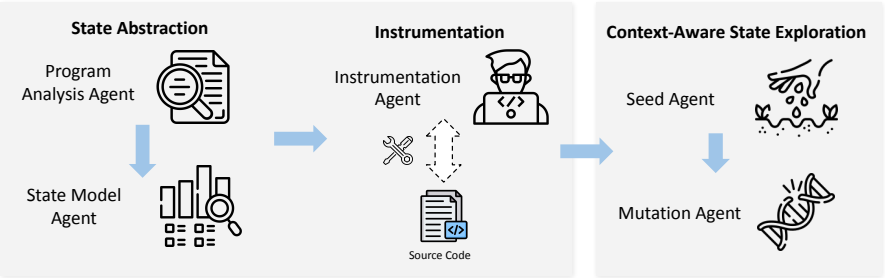

Figure 1: Overview of the ASA-Fuzz workflow. ASA-Fuzz orchestrates LLM-agents across two phases, setup (state abstraction and instrumentation) and active testing (context-aware state exploration), to support stateful fuzzing.

observation interface, allowing the system to track logical state transitions that traditional coverage metrics miss.

- **Context-Aware Action Synthesis**: Leveraging the exposed state signals, mutation agents synthesize executable Python operators that function as the fuzzer's action space. Unlike static heuristics, these agent-defined operators utilize runtime context (e.g., board state or variable values) to generate semantically valid inputs that respect complex program constraints.
- **Adaptive Plateau Recovery**: When exploration stalls, the system is designed to perform retrospective analysis over execution history and updates its strategy by injecting new seeds, refining the state abstraction, and synthesizing new operators. This feedback-driven loop enables sustained progress without manual retuning.

Key Results. We evaluated ASA-Fuzz on `CROMU_00005` GrammaTech (2016), a DARPA CGC chess-variant binary, against AFL++ Fioraldi et al. (2020), SGFuzz Ba et al. (2022), and Fuzz4All Xia et al. (2024a) under identical seeds and a two-hour budget. Our agentic fuzzer discovers over 10^5 unique board states, a $929\times$ improvement over AFL++, while all baselines plateau below 10^3. In terms of exploration depth, ASA-Fuzz achieves a median effective move depth of $\approx$25, significantly outperforming Fuzz4All ($\approx$7) and traditional fuzzers ($\leq$3). The results show that traditional mutations fail to preserve the strict move format, and LLM generation struggles to satisfy complex chess rules over multiple turns. Notably, while AFL++ attains marginally higher edge coverage, the semantic metrics reveal that this coverage is driven by shallow rejection paths rather than meaningful game logic. It demonstrates that extracting internal state variables to drive semantic-aware mutation is essential for effectively traversing state transitions and reaching deep program states where state-dependent vulnerabilities reside.

2 Related Work

Safe deployment of agents in the wild depends critically on the reliability of environments they act through, as real-world failures are trajectory-dependent and surface only after several valid interactions. Thus, testing in this setting presents unique needs of generating *realistic stateful sequences* that drive the environment into diverse internal conditions.

Outcome-Based Agent Evaluation. A growing body of work evaluates LLM agents in interactive environments, including web navigation and UI manipulation Zhou et al. (2024); Yao et al. (2022), computer/OS control Xie et al. (2024), tool-use with real APIs Qin et al. (2023), and repository-level software engineering Jimenez et al. (2023); Yang et al. (2024); Zhang et al. (2024). Most benchmarks nevertheless operationalize success using *endpoint* signals, task completion rates, goal-state matching, or tests passing for a final patch Jimenez et al. (2023); Yao et al. (2025). These outcome-centric metrics are necessary but often insufficient for safety, as they provide limited visibility into whether an agent has reached *diverse internal situations* of the underlying tool environment (e.g., distinct DB states, filesystem configurations, cache/mode toggles, permission regimes, or protocol phases). As a result, evaluations can systematically miss rare, trajectory-dependent failures that only manifest after long, valid interaction sequences, precisely the regimes encountered in deployment. Recent efforts begin to incorporate more process-level signals (e.g., progress-style metrics or tra-

jectory diagnostics) Chang et al. (2024); Kuang et al. (2025), but still typically do not target *state diversity* of the environment itself as a first-class coverage objective.

Classical Stateful Testing. Fuzzing is one of the de facto approaches in testing real-world software projects. It automatically generates inputs or input sequences by mutating seeds, executes the target, and uses runtime feedback such as crashes or coverage to guide further generation Zalewski (2014); Fioraldi et al. (2020); Sutton et al. (2007); Xia et al. (2024b).

On stateful programs, fuzzing often plateaus Lemieux et al. (2023). Coverage-guided loops (e.g., AFL++ Fioraldi et al. (2020)) can miss semantically distinct internal situations Ba et al. (2022); Zhao et al. (2022), while mutation tends to be constraint-blind, producing many invalid interactions that are rejected early Lemieux et al. (2023). Prior work on stateful fuzzing improves this typically by prioritizing message sequences and protocol states (AFLNet Pham et al. (2020)), modeling protocol state machines (StateAFL Natella (2022)), using response-based prioritization (SATFuzz Pan et al. (2022)), or incorporating additional state signals/annotations (IJON Aschermann et al. (2020), SGFuzz Ba et al. (2022)). In contrast, our approach makes stateful testing closed-loop and agentic: it automates state-signal selection and instrumentation, synthesizes constraint-aware operators, and adapts when exploration stalls.

LLMs and Agents for Program Analysis and Testing. Recent work uses LLMs to assist program analysis and automated testing, especially by generating more structured, format-aware inputs for fuzzing. ChatFuzz integrates GPT models into a fuzzing loop to produce structured variants of seeds Hu et al. (2023); Fioraldi et al. (2020). Fuzz4All further explores LLM-driven fuzzing with auto-prompting and iterative prompt refinement to generate diverse inputs across systems under test Xia et al. (2024a). CodaMosa invokes LLMs when fuzzing plateaus Lemieux et al. (2023). They often treat LLMs as better input generators within a fixed fuzzing loop. We instead use agents to infer state signals, synthesize state-conditioned input mutations for valid multi-step interactions, and adapt strategy to break plateaus.

3 Approach

3.1 Overview of ASA-Fuzz

Figure 1 illustrates the overall workflow of ASA-Fuzz for testing stateful programs. To address the aforementioned (1) gap between coverage plateaus and meaningful stages and (2) blindness in mutations, ASA-Fuzz employs a set of specialized agents working in concert in 2 phases.

- **State Abstraction and Instrumentation**: In this foundational phase, the *Program Analysis Agent* and *State Modeling Agent* perform static analysis to formally define the target program's state context. The *Instrumentation Agent* then instruments the source code, injecting logging logic to realize these proposed state definitions within the binary.
- **Context-Aware State Exploration**: Transitioning to active testing, the *Seed Agent* generates a corpus of diverse initial inputs tailored to the identified state space. The *Mutation Agent* synthesizes state-aware input transformation functions (operators), constructing a dynamic action space for the fuzzing process.

Instead of relying on generic heuristics, our design leverages agents to perform semantic program analysis. This allows the use of (1) State Feedback: deriving program-specific state feedback that tracks semantically meaningful variables rather than raw code paths; and (2) Context-Aware Mutations: constructing a mutation engine that utilizes runtime feedback to generate mutations more likely to conform to program semantics.

3.2 State Abstraction and Instrumentation

Traditional fuzzing tools depend on expert knowledge to determine which program variables to log, how to represent execution states, and what mutations best explore the input space. This manual setup is not only time-consuming but also limits scalability across different binaries or domains. Our workflow addresses this bottleneck by using LLM-driven reasoning to automatically extract program semantics, identify relevant instrumentation points, and design mutation and seed strategies.

State Representation. The core of our architecture relies on transforming execution logs into usable abstractions for fuzzing. We define an `execution_value` as a discrete data point captured during execution, expressed as a key-value pair:

$$v = (\textit{name}, \textit{type}, \textit{val})$$

Where *name* is the semantic identifier, *type* is the primitive data type ($T \in \{\text{int, float, bool, string}\}$), and *val* is the observed data.

Execution State ($\mathcal{S}$). The Execution State acts as the fuzzer's feedback mechanism, determining if an input has explored a unique logical path. A state $\mathcal{S}$ is an aggregation of values transformed by the following operators:

- *Predicate*: A boolean expression $\phi(v_1, v_2, \ldots, v_n)$ that evaluates logical conditions over execution values.
- *Value*: The most recent observation of v.
- *Sum*: The scalar accumulation $\sum v_i$ for numeric types.
- *Counter*: An incremental tally of how many times a specific predicate ϕ evaluates to `true`.
- *Set*: The set of all unique values $\mathcal{U} = \{v_1, v_2, \ldots\}$ observed for a given identifier.

Agentic Design. In this phase, the extraction of program semantics and identification of relevant instrumentation points is automated by 3 agents.

- The **Program Analysis Agent** first parses the source tree to generate two foundational documents that subsequent agents use as a source of "ground truth" about the program: (1) `workspace_overview.md` a document describing how the workspace is structured and any additional notes about the source files. (2) `program_analysis.md` a document describing the program's behavior and how inputs are parsed within the program.
- The **State Modeling Agent**, based on such program analysis, derives a program-specific logical model. This agent identifies the execution state (used for novelty detection) and the mutation context (used for guiding mutations in mutation operator functions), documenting the reasoning for these choices in `state_model.md`. The agent also produces a document for implementing instrumentation (`instrumentation_blueprint.md`), which identifies the precise source code locations for injecting logging statements to capture the required execution values.
- The **Instrumentation Agent** follows this blueprint, performing the source-to-source transformation and binary compilation necessary to realize the state model within the target executable.

3.3 Context-Aware State Exploration

Mutation Context ($\mathcal{C}$). To support context-aware state exploration, we now define the higher-level structure of **Mutation Context** based on the state representation described in Section 3.2. While the Execution State is optimized for distinguishing novelty, the Mutation Context ($\mathcal{C}$) is a set of state-values specifically curated to guide the LLM-based mutation engine. The distinction is critical: state values chosen for Execution State ($\mathcal{S}$) may not be best used in the mutation engine and vice versa.

Consider a fuzzer targeting a file parser. A `counter` tracking the number of processed blocks is an ideal **Execution State** component, as it allows the fuzzer to identify seeds that explore larger files. However, this counter provides little utility to the mutation engine. Instead, a **Mutation Context** containing a `set` of observed magic bytes provides the necessary semantic signal for the agent to synthesize operators that generate valid file headers. By decoupling these two structures, we provide the agent with the flexibility to define how specific program properties should influence the search strategy: either as a metric for novelty or as a grounding context for semantic transformations.

Context-Aware Mutation Synthesis. Unlike traditional mutation engines that rely on hardcoded, bit-level transformations, our architecture employs an LLM-based agent to synthesize a program-aware mutation engine. This engine consists of two layers: (1) *Semantic Operators* and (2) *Conditional Strategies*, allowing the agent to "exploit" specific program states.

- **Semantic Mutation Operators:** The agent generates a set of Python-based mutation functions, $\mathcal{O}$. Each operator $o \in \mathcal{O}$ is defined as a function of the current input string and the observed mutation context:

$$o : (\text{input_str}, \mathcal{C}) \rightarrow \text{mutated_str}$$

  By incorporating $\mathcal{C}$ into the operator logic, the agent can implement "context-aware" mutations.
- **Hierarchical Mutation Strategy:** To manage the selection of these operators, the agent defines a *Mutation Strategy* composed of ordered rules. Each rule consists of a predicate ϕ (evaluated against $\mathcal{C}$) and a weighted distribution of operators. The engine selects an operator through a two-step process: (1) *Rule Matching:* The engine evaluates rules in sequential order. The first rule whose predicate $\phi(\mathcal{C})$ evaluates to `true` is selected. A terminal `null` condition serves as the default fallback; and (2) *Probabilistic Selection:* Within the matched rule, an operator o_i is chosen based on its assigned weight w_i, where the probability $P(o_i) = \frac{w_i}{\sum w_j}$.

Agentic Design. For this purpose, the **Seed Agent** utilizes the input specifications and state definitions to generate an initial corpus of seeds (or program inputs in the fuzzing system) designed to maximize the initial state-space coverage. The agent can effectively test initial seeds by observing the state definition of any chosen inputs and iteratively choosing inputs until diverse states are achieved. The **Mutation Agent** creates a set of Python-based semantic mutation operators and a hierarchical mutation strategy as described above.

3.4 Agentic Coverage Plateau Remediation

To support generalization beyond tractable targets, ASA-Fuzz includes a plateau remediation protocol. While not exercised in the case study (Section 4), this mechanism is designed for targets where the state space is larger or operator efficacy degrades over time.

Stagnation in state discovery implies convergence to a local optimum. We classify the causes of such plateaus into three failure modes:

- **Seed Exhaustion:** The current corpus lacks the inputs required to traverse transitional boundaries.
- **Operator Inefficacy:** Mutation operators generate inputs that are statistically redundant or irrelevant to the current program context.
- **State Aliasing:** The novelty detection metric fails to distinguish between logically distinct program states.

To resolve these bottlenecks, the system employs a hierarchical remediation protocol managed by a reasoning supervisor.

Adaptive Plateau Recovery. A plateau is identified when the time elapsed since the last unique state discovery exceeds a threshold τ. Upon detection, the fuzzer serializes its internal telemetry (including coverage sets and operator statistics) into a diagnostic summary JSON.

The **Coverage Plateau Supervisor Agent** ingests this summary alongside the static analysis artifacts generated in Section 3.2. Acting as a meta-controller, the supervisor diagnoses the primary failure mode and activates one of three remediation sub-agents:

- **Seed Injection Subagent:** Synthesizes and injects novel inputs into the queue to expand the search frontier.
- **Mutation Engine Subagent:** Reconfigures the mutation strategy or synthesizes new operators to align with unexplored semantic constraints.
- **State Subagent:** Refines the instrumentation logic to increase the granularity of the novelty detection system, reducing state aliasing.

Empirical evaluation of plateau remediation on targets requiring sustained state recovery is left to future work.

Algorithm 1 State-Based Evolutionary Fuzzing Loop

Input: Initial seed corpus $\mathcal{C}_{in}$, Max iterations N
Output: Set of discovered unique states $\mathcal{H}$, Error logs $\mathcal{E}$
$\mathcal{Q} \leftarrow$ InitializeLIFOQueue($\emptyset$)
$\mathcal{H} \leftarrow \emptyset$
for $seed \in \mathcal{C}_{in}$ **do**
  $res, \mathcal{S}, \mathcal{C} \leftarrow$ ExecuteBinary($seed$)
  $\mathcal{Q}$.push($\langle seed, \mathcal{C} \rangle$)
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathcal{S}\}$
end for
while iterations $< N$ **do**
  $\langle s_{inp}, s_{ctx} \rangle \leftarrow \mathcal{Q}$.pop()
  for $i \leftarrow 1$ **to** n **do**
    $rule \leftarrow$ AgentStrategy.MatchRule(s_{ctx})
    $op \leftarrow$ WeightedSelect($rule$.operators)
    $m_{inp} \leftarrow op(s_{inp}, s_{ctx})$
    $res, \mathcal{S}_{new}, \mathcal{C}_{new} \leftarrow$ ExecuteBinary(m_{inp})
    if $\mathcal{S}_{new} \notin \mathcal{H}$ **then**
      $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathcal{S}_{new}\}$
      $\mathcal{Q}$.push($\langle m_{inp}, \mathcal{C}_{new} \rangle$)
    end if
  end for
end while

3.5 Evolutionary Search Loop

The agentic fuzzer operates in a continuous evolutionary cycle, as formalized in Algorithm 1. The novelty of a mutation is determined by its ability to reach a new execution state ($\mathcal{S}$). The loop is structured to maximize deep exploration of the program's state space:

(a) **Seed Selection**: The engine selects a seed from the corpus using a Last-In-First-Out (LIFO) priority queue. This depth-first approach biases the search toward the most recently discovered states, encouraging the exploration of long, sequential state transitions rather than shallow branching.

(b) **Batch Generation**: For a given seed, the Mutation Engine generates a batch of n candidate inputs. Notably, the seed is popped as a tuple $\langle s_{inp}, s_{ctx} \rangle$, allowing the agent-defined `mutation_strategy` to be evaluated immediately without re-execution.

(c) **Execution and State Extraction**: Each candidate is executed within the target binary. The system captures raw logs and transforms them into an execution state tuple $\mathcal{S}$ and a corresponding mutation context $\mathcal{C}$.

(d) **Novelty Evaluation**: Unlike coverage-guided fuzzers that look for new code paths, our system performs a State-Uniqueness Check. A candidate is added to the seed queue if and only if $\mathcal{S} \notin \mathcal{H}$, where $\mathcal{H}$ is the global history of observed states.

(e) **Feedback and Re-queuing**: If the queue is exhausted, the fuzzer re-populates $\mathcal{Q}$ with seeds from the global history $\mathcal{H}$ to explore alternate mutation paths, preventing the search from stalling.

3.6 Implementation

ASA-Fuzz is implemented as a distributed system in Python 3.12. The feedback loop extracts state variables directly from the program's standard output. During the execution phase, the system captures the target's `stdout` stream and utilizes regular expression pattern recognition to identify the specific `name: value` patterns injected by the Instrumentation Agent. These captured values are automatically parsed and cast into their appropriate data types to update the execution state for novelty detection and the mutation context for the agent-defined mutation engine.

The setup workflow (3.2) is managed via the LangChain Developers (2023) library.

- GPT-5.1-Codex is used for the Program Analysis, State Modeling, Mutation, and Seed agents, where reasoning over program semantics is required.
- GPT-5.1-Codex-Mini is used for the Instrumentation Agent to perform the more structured, deterministic task of source-to-source logging injection based on the provided blueprint.

The architecture is deployed across a dual-container Docker environment on Ubuntu 22.04, comprising a "Target Sandbox" containing the program source code and a "Fuzzer Controller." To facilitate agent interaction with this distributed environment, we implemented a custom LangChain Backend Protocol. This protocol enables the LangChain Filesystem Middleware to perform cross-container operations, allowing agents to read, write, and execute files within the target environment as if they were local. This shared workspace serves as the "memory" of the system, storing markdown artifacts and binary logs that coordinate the multi-agent handoff.

4 Case Study

4.1 Target Overview: CROMU_00005

We evaluate ASA-Fuzz on `CROMU_00005` GrammaTech (2016), a stateful, DARPA CGC DARPA (2014) binary implementing a Chess variant. The program interface accepts coordinate-based moves ($x_1, y_1 \rightarrow x_2, y_2$) and numeric codes (e.g., `9` to display the board). The vulnerability is a classic spatial safety violation. Moving a white knight to specific coordinates bypasses boundary checks. This failure leads to an out-of-bounds (OOB) write, overwriting the pointer responsible for rendering the board. Triggering this bug requires a particular sequence of valid moves to clear obstacles and position the knight at the target coordinate. See Appendix A for details.

`CROMU_00005` represents a "semantic wall" for traditional fuzzing architectures for three reasons: (1) *Syntax Constraints*, as moves must adhere to a strict format (e.g., `digit,digit space digit,digit`) which standard bit-flipping mutations rarely preserve; (2) *Rule-Based Validity*, where even syntactically correct inputs are rejected if they violate chess rules or reference empty squares; and (3) *State Sequencing*, where the vulnerability is not reachable via a single input but requires a multi-move trajectory through the game's logic.

4.2 Agentic Synthesis

We demonstrate the pipeline (Section 3.2) on `CROMU_00005`.

State Abstraction and Instrumentation. To address the program's strict syntax and rule-based constraints, the **Program Analysis Agent** produces two primary artifacts: `workspace_overview.md`, which maps the project structure (Listing 1), and `program_analysis.md` (Listing 2), which defines the input grammar (e.g., coordinate-to-coordinate moves) and semantic invariants. The **State Modeling Agent** identifies critical variables to serve as execution state ($\mathcal{S}$) and mutation context ($\mathcal{C}$). As shown in Listing 3, it synthesizes a "Minimum Viable State" by serializing the 8×8 board into a 64-character string. This allows the fuzzer to track chess positions rather than raw code coverage. For the mutation engine, the agent further extends the context to include turn info and piece counts (Listing 4) to guide the synthesis of semantically valid moves.

The **Instrumentation Agent** then realizes these definitions by following a generated `instrumentation_blueprint.md`. This blueprint specifies the custom printing mechanism (Listing 5) and the precise hook points (Listing 6) required to extract values within the unique environment of the program. Notably, as shown in Listing 7, the agent synthesizes an instrumentation block within the game loop that converts the two-dimensional board array into the required string buffer. Finally, the agent validates the instrumentation by compiling the binary and verifying the output against automated test cases (Listing 8).

Context-Aware State Exploration. The system configures the active exploration components using the previously established semantic models. The **Seed Agent** first generates an initial corpus (Listing 9) by iteratively proposing inputs and verifying them with tools like `test_seeds`. This ensures that the starting seeds are not only syntactically valid but also exercise unique execution

states, providing semantic diversity from the outset. Subsequently, the **Mutation Agent** synthesizes a program-aware mutation engine. Unlike standard bit-level mutators, the agent generates specialized Python operators that utilize the mutation context ($\mathcal{C}$) to perform complex logic. For example, the agent implements helper functions to identify pieces on the board (Listing 10) and wraps them into operators like `knight_jump_operator` (Listing 11), which calculates valid L-shape trajectories based on the current board layout. Finally, the agent defines a weighted strategy in `mutation_strategy.json` (Listing 12). This strategy adjusts operator selection probabilities based on real-time feedback; for instance, it can prioritize aggressive piece captures when the mutation context indicates a numerical advantage in piece counts.

4.3 Baselines and Setup

We use three baselines that cover classic grey-box fuzzing, LLM-based input generation, and state-aware fuzzing: (1) **AFL++** Fioraldi et al. (2020) is a widely used coverage-guided greybox fuzzer and an enhanced version of the original AFL, focusing on practical improvements such as faster and better instrumentation, advanced mutation strategies, and broader platform support. (2) **Fuzz4All** Xia et al. (2024a) is an LLM-powered universal fuzzer for systems that take programming or formal languages as input. It uses autoprompting to derive effective prompts from specs or examples, then runs an LLM-driven fuzzing loop to iteratively generate and mutate diverse, realistic inputs across targets. (3) **SGFuzz** Ba et al. (2022) targets stateful programs, such as protocol implementations. It tracks state variables during execution to construct a State Transition Tree (STT). SGFuzz leverages STT coverage instead of just code coverage to guide seed selection and mutation, enabling the exploration of state spaces often missed by stateless fuzzers. All fuzzers start with the same initial seeds (Appendix C) and run for 2 hours.

4.4 Evaluation

Evaluation Metrics. We record the generated seed corpus to compare the edge coverage over time. We propose two additional metrics to compare the semantics of the program:

- **Unique Board States Over Time**: The essential "state" object in the program is the chessboard. Mutations targeting changes in the chessboard are more likely to trigger the program vulnerability that is gated behind a series of chess moves. Consequently, a corpus exercising more unique board states increases the probability of uncovering the vulnerability.
- **Effective Move Depth Distribution**: The effective move depth is the number of moves from the input that caused a change on the board. Mutations that exercise a greater number of effective moves will exercise a deeper board state. A corpus with a larger move depth can be expected to reach deeper states within the program.

We manually instrument the source code to collect logs of the board hash and processed valid move strings for our additional metrics.

Corpus Metric	ASA-Fuzz	AFL++	SGFuzz	Fuzz4All
Corpus Size (Files)	34,697	732	15,031	2,120
Average File Size	3,348 B	1,201 B	94 B	61 B
Standard Deviation	1,147 B	1,876 B	117 B	7 B
Min File Size	6 B	1 B	1 B	0 B
Max File Size	4,600 B	9,074 B	4,096 B	93 B

Table 1: Corpus statistics after the two-hour fuzzing budget across all four fuzzers.

Corpus Summary. The ASA-Fuzz corpus is significantly larger than the baselines, containing approximately 47 times more files than AFL++ and over double the count of the second largest SGFuzz. The AFL++ and ASA-Fuzz corpus have higher average file sizes and standard deviations in contrast to SGFuzz and Fuzz4All, suggesting that they may explore a more constrained input space compared to the ASA-Fuzz and AFL++ approaches.

Unique Board State Discovery. As shown in Figure 2, ASA-Fuzz exhibits a dramatic advantage in discovering valid semantic states. Plotting unique board states on a logarithmic scale, our approach rapidly ascends to over 10^5 unique states within the two-hour window.

- Compared with traditional fuzzers: Both AFL++ and SGFuzz plateau early, discovering fewer than 10^3 states. This confirms that coverage-guided and heuristic-based state fuzzers struggle to penetrate the "semantic wall" of valid chess moves.
- Compared with LLM Fuzzers: While Fuzz4All outperforms the traditional baselines (reaching $\approx 10^3$ states), it still falls orders of magnitude short of our agentic system. This gap suggests that LLM generation alone is insufficient without the closed-loop feedback and state abstraction provided by our architecture.

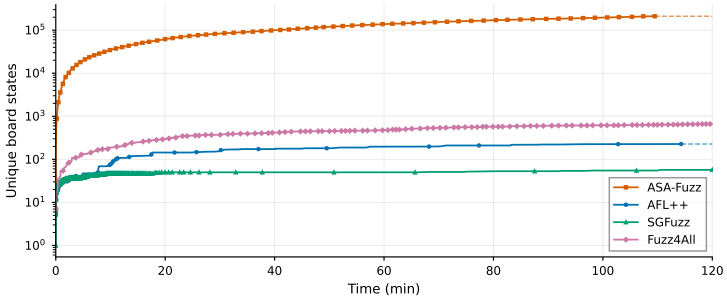

Figure 2: Unique board states discovered over time (log scale). ASA-Fuzz reaches over 10^5 unique states within the two-hour window while all baselines plateau below 10^3.

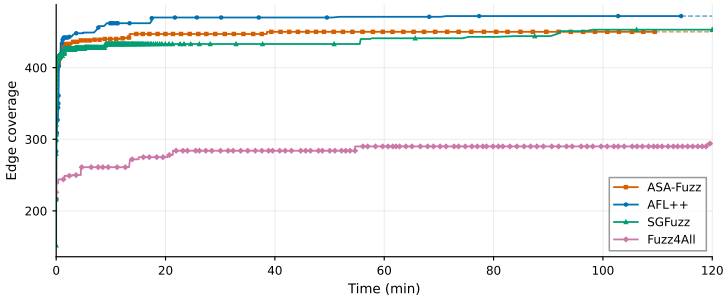

Figure 3: Edge coverage over time measured using `afl-showmap`. AFL++ achieves the highest edge coverage (≈ 470), marginally outperforming ASA-Fuzz (≈ 450).

Effective Move Depth. The distinction in semantic exploration is further visualized in Figure 4. ASA-Fuzz achieves a median effective move depth of ≈ 25, with an interquartile range extending from 18 to 33 and maximum depths reaching nearly 60 moves. This indicates the agent is successfully playing full, valid games. In contrast, AFL++ and SGFuzz possess median depths of ≈ 3 and ≈ 2 respectively, indicating they rarely progress beyond the opening moves before generating invalid inputs. Fuzz4All shows a tight distribution around a median of 7 moves, suggesting it can generate short valid sequences but lacks the state-aware adaptability to sustain long-horizon play.

Edge Coverage vs. Semantic Depth. Figure 3 reveals that higher edge coverage does not imply semantic success. AFL++ achieves the highest edge coverage (≈ 470), marginally outperforming ASA-Fuzz (≈ 450). However, comparing this with Figure 2 and Figure 4 reveals that AFL++ is likely exploring error-handling paths and shallow rejection states rather than valid game logic. Notably, Fuzz4All exhibits significantly lower edge coverage (≈ 300), potentially due to its generation of repetitive or syntactically limited patterns that fail to exercise diverse control flows.

Cost. ASA-Fuzz incurs an initial setup cost. We used 838k/77.7k input/output tokens for GPT-5.1-Codex and 411k/27k input/output tokens for GPT-5.1-Codex-mini.

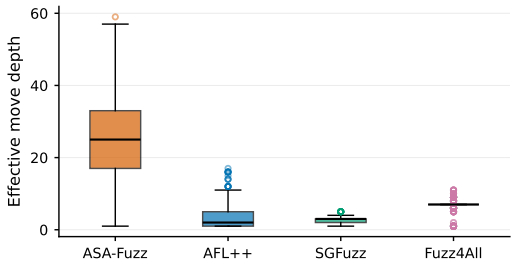

Figure 4: Distribution of effective move depth across the seed corpus. ASA-Fuzz achieves a median depth of ≈25 moves with an interquartile range of 18 to 33, while AFL++, SGFuzz, and Fuzz4All achieve medians of ≈3, ≈2, and ≈7 respectively.

5 Discussion

Orchestration and Multi-Agent Coordination. The complexity of vulnerability discovery in stateful programs necessitates a decomposition of responsibilities. We observe that a monolithic model struggles to maintain the context required to simultaneously reason about high-level program semantics and low-level instrumentation details. ASA-Fuzz addresses this by orchestrating specialized agents within a distributed Docker environment. By enforcing a structured artifact-based handoff, the system effectively modularizes the fuzzing pipeline. This design suggests that robust agentic systems benefit significantly from isolating reasoning tasks (e.g., state modeling) from deterministic execution tasks (e.g., compiling instrumentation).

The Decoupling of Coverage and Semantic Progress. A critical finding of this study is the decoupling of code coverage from semantic state exploration. As detailed in Figure 2, AFL++ achieved the highest edge coverage (≈ 470) yet discovered 929$\times$ fewer unique board states than ASA-Fuzz. This inversion suggests that in rule-bounded environments, traditional coverage metrics often reward shallow exploration rather than deep state transitions. The performance of SGFuzz highlights that observing state is insufficient without the ability to act upon it. While SGFuzz tracks the state, it lacks the semantic understanding to generate the specific, syntax-heavy inputs required to traverse that state graph, resulting in a median effective move depth of only ≈ 2.

Closed-Loop vs. Open-Loop Generative Testing. The comparison with Fuzz4All illuminates the limitations of open-loop LLM generation. Although Fuzz4All outperformed traditional baselines by leveraging an LLM to generate valid chess syntax (reaching $\approx 10^3$ states), it still fell orders of magnitude short of ASA-Fuzz. We attribute this to the lack of runtime grounding; Fuzz4All operates without direct visibility into the program's evolving internal state.

The agents transform the fuzzing process into a partially observable Markov decision process by dynamically instrumenting the target to expose variables such as `board_flat` and `piece_count`. This allows the Mutation Agent to condition its synthesis on the actual current state. This grounded reasoning allowed the agent to maintain a median game depth of ≈ 25 moves, validating that semantic exploration requires both state perception and state-conditioned action.

Resource Allocation and Initialization Costs. ASA-Fuzz introduces initialization overhead, consuming approximately 838k input tokens for the GPT-5.1-Codex model during the setup phase. However, this cost represents the automation of manual reverse engineering and instrumentation, tasks that typically require significant human effort. For targets where deep state exploration is blocked by "semantic walls", the trade-off favors the high initial token cost over the heuristic limitations of bit-level mutation engines.

Acknowledgments

The authors of this work are in part supported by National Science Foundation (NSF) grants CCF-2426161 and OAC-2530911, Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590041, Cisco Research Award, Google Research Credit Award, and UCR Senate Awards. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and DARPA.

REFERENCES

Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1597–1612, 2020. doi: 10.1109/SP40000.2020.00117.

Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3255–3272, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

Ma Chang, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362, 2024.

DARPA. Cyber grand challenge (cgc). https://www.darpa.mil/research/programs/cyber-grand-challenge, 2014. Accessed: 2025-09-16.

LangChain Developers. Langchain. https://python.langchain.com, 2023. Version 0.3.27, accessed 2026-01-07.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

GrammaTech. Cgc challenge binary: Cromu_00001. https://github.com/GrammaTech/cgc-cbs/tree/master/cqe-challenges/CROMU_00005, 2016. Accessed: 2025-09-16.

Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai, 2023. URL https://arxiv.org/abs/2306.06782.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Jiayi Kuang, Yinghui Li, Xin Zhang, Yangning Li, Di Yin, Xing Sun, Ying Shen, and Philip S Yu. Process-level trajectory evaluation for environment configuration in software engineering agents. *arXiv preprint arXiv:2510.25694*, 2025.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 919–931. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00085. URL https://doi.org/10.1109/ICSE48619.2023.00085.

Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(191), 2022. URL https://doi.org/10.1007/s10664-022-10233-3.

Zulie Pan, Liqun Zhang, Zhihao Hu, Yang Li, and Yuanchao Chen. Satfuzz: A stateful network protocol fuzzing framework from a novel perspective. *Applied Sciences*, 12(15):7459, 2022.

Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, Boston, MA, USA, 2007. ISBN 978-0321446116.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering*, ICSE '24, 2024a.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*, pp. 1–13, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 979-8-4007-0217-4. doi: 10.1145/3597503.3639121. URL https://doi.org/10.1145/3597503.3639121.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

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

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. τ-bench: A benchmark forTool-Agent-User interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025.

Michał Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2014. Accessed: 2025-09-16.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.

Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3273–3289, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

A TARGET PROGRAM: CROMU_00005

CROMU_00005 is a challenge binary from the DARPA Cyber Grand Challenge (CGC), authored by Cromulence. The program implements a text-based chess engine with modified rules: collisions between pieces result in a *swap* of positions rather than a capture. The user controls both sides (white and black), alternating turns.

Board and Input Format. The game displays an 8×8 board using single-character piece encodings:

The user interacts by entering source and destination coordinates in the format `x1,y1 x2,y2`. Each move is validated against standard chess movement rules (pawn, rook, knight, bishop, king, queen) and path clearance. The user may also print the current board layout (command `9`) or quit (command `666`).

Vulnerability. The vulnerability is a spatial safety violation (CWE-787: Out-of-Bounds Write). In `service.c:performMove`, the white knight's move validation occurs *before* the board boundary check. This allows the knight to be moved to the off-board coordinate $(7, 8)$, one row beyond

```
jhilkihj
gggggggg
........
........
........
........
aaaaaaaa
dbcfecbd
```

Figure 5: Initial board layout of CROMU_00005. Rows from top to bottom: black pieces (y=7), black pawns (y=6), empty rows (y=5..2), white pawns (y=1), white pieces (y=0).

the board array. This out-of-bounds write overwrites a function pointer on the stack that controls the board display routine. If the user subsequently issues the display command (`9`), execution jumps to the corrupted pointer.

Fuzzing Challenge. Triggering the vulnerability requires a specific sequence of *legal* chess moves: the player must clear a path for the white knight, maneuver it to the target coordinate using valid L-shaped jumps, and then issue the display command. The combinatorial explosion of board states—combined with strict syntactic and semantic input constraints—makes this target particularly difficult for traditional coverage-guided fuzzers that rely on bit-level mutations.

B Case Study Artifacts

Listing 1 Excerpt from `workspace_overview.md`

```
## Top-Level Structure
- `src/`: Contains the main service implementation (`service.c`).
- `include/`: Public header files, including `service.h` defining data structures, constants, and function prototypes.
- `lib/`: Custom standard library implementations and math utilities used by the service (e.g., `stdlib.c`, `printf.c`, `mymath.c`).
- Shell configuration files (`.bashrc`, `.profile`, etc.) are present but irrelevant to the service.
```

Listing 2 Excerpt from `program_analysis.md`

```
## 3. Semantics / State Machine
- **States**: Implicit state tracked by `current_team` (`WHITE_TEAM` or `BLACK_TEAM`). No additional modes beyond loop.
- **Transitions**: ...
- **Command Preconditions**:
  - Move source must contain piece of `current_team`; otherwise `performMove` returns 0.
  - Destination must be in bounds and satisfy specific piece movement rules.
  ...
- **Command Effects**:
  ...
- **Dependencies / Ordering**:
  - Only current team's pieces may be moved; a successful move is required to hand over turn.
  - Moves must obey standard chess-style piece movements: pawns move forward one square (no captures/diagonals), rooks straight lines, bishops move diagonally, knights L-shape, queen combination, king adjacent squares.
- **Accumulators/Counters**: ...
```

C Initial Seed Corpus

All baselines and the agentic fuzzer share the same 7 initial seeds (Listing 13). For Fuzz4All these seeds are embedded as examples in the LLM prompt; for AFL++ and SGFuzz they are placed in the seed corpus directory.

Listing 3 Excerpt from `state_model.md`

```
## Execution States (Minimum Viable State)

1. **board_flat**
   - **Type**: value of execution value `board_flat`
   - **Rationale**: Represents the entire 8x8 board serialized into a deterministic
   ↪  64-character string (y = 0..7, x = 0..7). This is the primary domain object the program
   ↪  manipulates; every legal/illegal move, capture, or display command depends on the
   ↪  precise arrangement of characters here. Tracking the exact board contents guarantees
   ↪  that two distinct game positions always map to distinct execution states.

2. **turn_team**
   - **Type**: value of execution value `turn_team`
   - **Rationale**: Captures whose turn it is (`WHITE_TEAM` or `BLACK_TEAM`). Even with an
   ↪  identical board layout, behavior differs depending on which team must move next (legal
   ↪  sources, move validity, and prompts all depend on this flag). Thus it is essential
   ↪  domain/session state.
```

Listing 4 Excerpt from `state_model.md`

```
## Mutation Context

1. **board_flat (value)**
   - **Use**: Gives mutation operators direct access to the actual board layout. They can
   ↪  reason about which coordinates currently hold movable pieces, where empty squares are,
   ↪  and where opposing targets exist. This enables generation of syntactically|and more
   ↪  importantly, semantically|valid move candidates (e.g., only proposing moves from
   ↪  squares containing the side-to-move pieces).

2. **turn_team (value)**
   - **Use**: Tells the mutator which side must act when crafting the next command. Without
   ↪  this, it could issue moves for the wrong color, leading to systematic `NO` responses.
   ↪  Paired with `board_flat`, the operator can filter to the appropriate source squares.

3. **white_piece_count (value)**
   - **Use**: Helps the mutator gauge White's remaining material and prioritize captures or
   ↪  evasive maneuvers when the count is low. It also acts as an easy sanity check that
   ↪  board parsing succeeded (counts should never exceed 16).

4. **black_piece_count (value)**
   - **Use**: ...

...
```

Listing 5 Excerpt from `instrumentation_blueprint.md`

```
## Printing Mechanism
- **Description**: Use the challenge's custom `printf` that treats `@` as the format
↪  introducer. Every execution value must be printed as `"<name>: <value>\n"`. For raw
↪  buffers (e.g., the flattened board), first print the label with `printf`, then stream the
↪  buffer bytes via `transmit` to avoid intermediate allocations, and terminate with
↪  `printf("\n")`.
- **Example**: `printf("turn_team: @d\n", (int)current_team);`
```

Listing 6 Excerpt from `instrumentation_blueprint.md`

```
### 1. `turn_team`
- **File**: `src/service.c`
- **Line Range**: 560-575 (within `main`'s game loop)
- **Hook Point**: `main` | immediately after the `if (performMove(move)) { ... } else { ... }`
↪  block, before the end of the `while (1)` loop.
- **Access Path**: `current_team`
- **Type Cast**: `(int)current_team`
- **Example Print Output**: `turn_team: 1`
- **Additional Notes**: Value flips only after successful moves, so logging here reflects the
↪  side to act for the next command.
```

Listing 7 Excerpt from `src/service.c`

```c
    else
    {
            if(performMove(move))
            {
                    printf(OK);
                    if (current_team == WHITE_TEAM)
                            current_team = BLACK_TEAM;
                    else
                            current_team = WHITE_TEAM;
            }
            else
            {
                    printf(NO);
            }
    }
    // Added instrumentation block from agent
    {
            size_t tx_count = 0;
            int idx = 0;
            int white_count = 0;
            int black_count = 0;

            for (int y = 0; y < HEIGHT; y++)
            {
                    for (int x = 0; x < WIDTH; x++)
                    {
                            char cell = (*ptr_array)[x][y];
                            board_flat_buf[idx++] = cell;
                            int color = getColor(cell);
                            if (color == WHITE_PIECE)
                                    white_count++;
                            else if (color == BLACK_PIECE)
                                    black_count++;
                    }
            }

            board_flat_buf[idx] = '\0';

            printf("board_flat: ");
            transmit(STDOUT, board_flat_buf, WIDTH * HEIGHT, &tx_count);
            printf("\n");

            printf("turn_team: @d\n", (int)current_team);
            printf("white_piece_count: @d\n", white_count);
            printf("black_piece_count: @d\n", black_count);
    }
```

Listing 8 Excerpt from `instrumentation_blueprint.md`

```md
2. **Input**:
   ```
   0,1 0,2
   0,6 0,5
   ```
   - **Expected Execution State Output**:
     - After first move: pawn at (0,1) moves to (0,2); `turn_team: 0`; counts stay 16/16.
     - After second move: black pawn (0,6) to (0,5); `turn_team: 1`; counts still 16/16.
   - **Test Case Description**: Two legal pawn pushes to confirm board serialization updates
   ↪  and `turn_team` toggles only on successful moves.
```

Listing 9 Excerpt from `initial_seeds.md`

```md
## Seed List
1. `9\n666\n` { Minimal command coverage: show board then exit.
2. `0,1 0,2\n666\n` { Single white pawn advance to flip turn flag, then exit.
3. `0,1 0,2\n1,6 1,5\n666\n` { Back-to-back legal pawn moves for both teams, exercising
↪  alternating turns.
4. `0,1 0,2\n0,6 0,5\n0,0 0,1\n6,6 6,5\n666\n` { Pawn trades followed by rook advances,
↪  altering board layout near both edges.
...
```

Listing 10 Excerpt from `operators.py`

```python
def _generate_knight_move(board: List[List[str]], team: int) -> Optional[str]:
    data = TEAM_DATA.get(team, TEAM_DATA[0])
    knights: List[Tuple[int, int]] = []
    for y in range(BOARD_SIZE):
        for x in range(BOARD_SIZE):
            if board[y][x] == data["knight"]:
                knights.append((x, y))
    random.shuffle(knights)
    for (x, y) in knights:
        deltas = list(KNIGHT_DELTAS)
        random.shuffle(deltas)
        for dx, dy in deltas:
            tx, ty = x + dx, y + dy
            if 0 <= tx < BOARD_SIZE and 0 <= ty < BOARD_SIZE:
                occupant = board[ty][tx]
                if occupant == "." or occupant in data["enemy"]:
                    return _format_move((x, y), (tx, ty))
    return None
```

Listing 11 Excerpt from `operators.py`

```python
def knight_jump_operator(input_str: str, mutation_context: dict) -> str:
    """Move a knight in an L-shape, preferring legal board-aware hops."""
    board = _board_to_grid(mutation_context.get("board_flat", ""))
    team = mutation_context.get("turn_team", 0)
    move = _generate_knight_move(board, team)
    if move is None:
        move = _fallback_random_move()
    return _inject_formatted_command(input_str, move)
```

Listing 12 Excerpt from `mutation_strategy.json`

```json
{
  "name": "white_advantage_aggression",
  "condition": "turn_team == 0 && black_piece_count < white_piece_count",
  "operators": [
    ["pawn_capture_operator", 3],
    ["knight_jump_operator", 2],
    ["line_slide_operator", 3],
    ["coordinate_nudge_operator", 1]
  ]
}
```

Listing 13 Initial seeds shared by all fuzzers

```
// Display board, then quit.
1. `9\n666\n`
// Single white pawn advance.
2. `0,1 0,2\n666\n`
// One move per side, exercising alternating turns.
3. `0,1 0,2\n1,6 1,5\n666\n`
// Pawn and rook moves on both sides.
4. `0,1 0,2\n0,6 0,5\n0,0 0,1\n6,6 6,5\n666\n`
// White knight development (b1 path).
5. `1,0 2,2\n1,6 1,5\n2,2 3,4\n5,6 5,5\n3,4 4,6\n666\n`
// Queen-side pawn and piece moves.
6. `3,1 3,2\n4,6 4,5\n3,0 3,1\n666\n`
// King-side knight development (g1 path).
7. `6,0 5,2\n2,6 2,5\n5,2 4,4\n2,5 2,4\n666\n`
```