Agentic Orchestration for Hybrid Fuzzing

Zhengxiong Luo*‡, Huan Zhao*‡, Mehtab Zafar*, Cristian Cadar†, Abhik Roychoudhury*
*National University of Singapore †Imperial College London
{luozx, mehtab, abhik}@nus.edu.sg, zhaohuan@comp.nus.edu.sg, c.cadar@imperial.ac.uk

***Abstract*—Hybrid fuzzing, which combines the scalability of greybox fuzzing with the precision of symbolic execution, has proven effective in automated software testing. However, contemporary hybrid fuzzers adopt a fuzzer-centric paradigm, where symbolic execution is regarded merely as a reactive fallback when fuzzing plateaus. This coordination often results in biased exploration and inefficient tester interleaving, hindering effective synergy between the two techniques.**

In this paper, we propose agentic orchestration for hybrid fuzzing, a new paradigm in which fuzzing and symbolic reasoning operate as equal first-class citizens and are synergized through an LLM agent that serves as the orchestrator. By inspecting testing artifacts and accumulating a testing trajectory, the agentic orchestrator adaptively configures the appropriate tester to reach intermediate milestones. We implement our approach in AHFUZZ and evaluate it empirically on 16 real-world subjects. AHFUZZ outperforms conventional fuzzers in branch coverage by 84%–166%, and discovers 13 previously unknown vulnerabilities. Up to now, 9 have been fixed or confirmed, resulting in 2 CVE assignments and bounty rewards.

I. INTRODUCTION

Software vulnerabilities pose critical security threats, leading to service failures and exploitable weaknesses that severely undermine the integrity and reliability of software systems [1]. Automated testing techniques, particularly those based on greybox fuzzing [2], have proven remarkably effective at discovering vulnerabilities at scale [3]. However, because of its randomized and lightweight nature, greybox fuzzing alone often struggles to penetrate code regions guarded by intricate conditions that require precise input values to hold. Consequently, prolonged fuzzing campaigns exhibit diminishing returns in vulnerability discovery [4]. To address this limitation, researchers have developed **hybrid fuzzing** [5]–[8] that augments fuzzing with fine-grained constraint solving, typically through dynamic symbolic execution [9] or concolic execution [10]. These hybrid approaches combine the scalability of fuzzing with the rigor of symbolic reasoning, achieving deeper code coverage and uncovering vulnerabilities that neither technique could find in isolation [5], [11], [12].

Existing Paradigm. Contemporary hybrid fuzzers integrate fuzzing and symbolic execution through *runtime alternation* [11], as shown in Figure 1A: fuzzing drives the primary exploration by default (ⓐ), and symbolic execution is merely a fallback (ⓑ). When coverage growth stagnates, control shifts to symbolic execution to expand the testing *frontier* by solving blocking constraints, or *roadblocks* (ⓒ). Within this paradigm, substantial research has improved hybrid fuzzing along multiple dimensions, including execution throughput (ⓐ/ⓑ) [6], [8], [13], [14], constraint-solving capabilities (ⓑ) [7], [12], and frontier selection strategies (ⓒ) [15], [16].

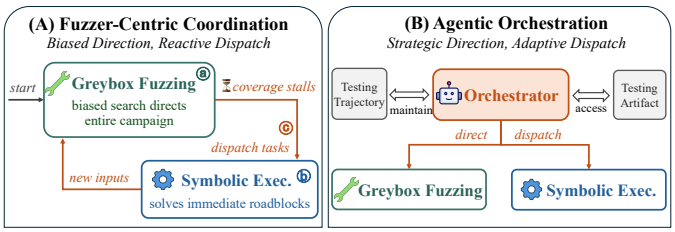


Fig. 1. Comparison of coordination modes in hybrid fuzzing: (A) *conventional* fuzzer-centric coordination *vs.* (B) *our* agentic orchestration.

Insights. Despite their success, we observe that the *fuzzer-centric* design philosophy leads to two coordination inefficiencies: (1) ***Biased Direction***. The exploration direction of the entire campaign is essentially driven by greybox fuzzing alone, since symbolic execution is only invoked to solve roadblocks at the fuzzer's frontier. However, fuzzing's coarse-grained edge coverage makes it prone to biased, greedy exploration that lingers in low-value code regions. This locality is inherent to fuzzing's lightweight design, fundamentally limiting its ability to steer the campaign toward the global optimum. (2) ***Reactive Dispatch***. Even when symbolic execution is invoked upon coverage stagnation, each switch solves only the immediate blocking constraint. Once the constraint is solved, control is eagerly returned to fuzzing, ignoring potential inter-dependencies among downstream constraints. The fuzzer's blind mutations then often break previously satisfied constraints, trapping the campaign in repeated stall-wait-solve cycles with significant budget wasted on futile fuzzing in between (as detailed in §II-B).

To better realize the *synergy* between fuzzing and symbolic execution, we identify two requirements for effective orchestration: (1) ***Strategic Direction***: At a high level, the coordinator should prioritize high-value targets by understanding the program semantics, a role that neither fuzzing's local scalar coverage signals nor symbolic execution's path-explosive exploration can fulfill; (2) ***Adaptive Dispatch***: The coordinator should proactively dispatch each exploration task to the best-

‡Joint first authors, listed in alphabetical order.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230045
www.ndss-symposium.org

suited *tester* (fuzzing or symbolic execution) based on the task nature and each tester's capabilities. Both requirements demand a coordinator capable of code comprehension and campaign-level reasoning. Recent advances in LLM-based agents have shown promise across various disciplines in software engineering and security [17]–[20]. This raises the question: *Can an LLM agent better orchestrate hybrid fuzzing by reasoning about the testing process?*

Our Approach. In this paper, we introduce a new coordination paradigm that replaces *reactive* tester alternation with intelligent, *proactive* orchestration, as shown in Figure 1B. Rather than switching to symbolic execution as a last resort, we treat both techniques as first-class citizens and bring out their complementary strengths through an orchestrator agent. This agent maintains a high-level view of both the program and the testing campaign and adaptively invokes testers to pursue the global testing objective. Specifically, through code comprehension, the agent can distinguish core program logic from peripheral regions. To reach these high-value targets, it operates in an iterative loop: at each step, it formulates a context-aware *goal* based on current progress (from testing artifacts) and past experience, then summarizes the *achievement* upon testing completion. These ⟨*goal, achievement*⟩ tuples accumulate into the *testing trajectory*—a semantic history that reveals which strategies have been productive and where diminishing returns set in—enabling strategic prioritization over greedy local exploration. Guided by this global direction, the agent adaptively dispatches testers at each step: it selects goal-aligned seeds and invokes the better-suited tester based on the nature of each task. For instance, when recognizing a cluster of inter-dependent constraints beyond fuzzing's reach, the agent delegates the *entire cluster* to symbolic execution in a single pass, eliminating the one-at-a-time stall-wait-solve cycles inherent in reactive switching. Collectively, these capabilities enable deliberate, informed orchestration that realizes the synergy between the two testers.

Evaluation. We implement the proposed paradigm in a tool called AHFUZZ and compare it against state-of-the-art hybrid fuzzers, like SymSan++ [8], MEUZZ [16], and CoFuzz [15], as well as standalone AFL++ [21] in both vanilla and advanced configurations, and the ensemble fuzzer autofz [22]. On 16 real-world subjects drawn from these baselines' evaluation suites, AHFUZZ achieves $2.26\times$, $2.66\times$, $2.60\times$, $2.08\times$, $2.01\times$, and $1.84\times$ higher branch coverage on average, respectively, and uncovers 13 previously unknown bugs, while the best-performing baseline finds at most 4 of them. We also demonstrate that AHFUZZ's agency in orchestration contributes considerably to its overall efficacy. A detailed analysis of AHFUZZ's orchestration process further reveals adaptive, subject-specific strategies that evolve with the testing trajectory, at a modest cost of $7.3 per campaign.

Contributions. Our key contributions are as follows:

- **New Paradigm.** We formulate a new hybrid coordination paradigm that treats fuzzing and symbolic reasoning as equal partners under campaign-informed agentic orchestration.
- **New Tool.** We design and build the first agentic orchestration framework for hybrid fuzzing, AHFUZZ, and open-source it to facilitate future research at

  https://github.com/ConcoLLMic/AhFuzz.
- **Evaluation.** We empirically evaluate AHFUZZ and demonstrate its significant improvement in both code coverage and vulnerability discovery over state-of-the-art tools.

II. BACKGROUND AND MOTIVATION

Greybox fuzzing [23] uses execution feedback like code coverage to guide its search. Rather than tracking complete execution paths, it decomposes each path into basic-block transitions (*edges*) coupled with bucketed *hit_counts*, where any bucket change counts as new testing progress [21]. This coarse syntactic representation enables high-throughput, scalable testing [2], [3], but this trade-off inevitably confines the fuzzer to a *local* view of progress, making it akin to a biased random search. Moreover, its mutation-based approach often struggles to satisfy complex conditions or nested branches [24] guarding deeper program logic.

Symbolic execution [9], [25]–[27], in contrast, formalizes path exploration as a constraint satisfaction problem. It treats program inputs as symbolic variables and encodes execution paths as logical constraints, which are then solved using tools like SMT solvers [28]–[31] to produce concrete inputs exercising targeted paths. This makes it effective at penetrating deeply nested code regions that are difficult for fuzzing to reach. A practical variant, concolic execution [10], [32]–[34], improves scalability by coupling concrete and symbolic execution. Despite recent efforts to assist constraint solving via fuzzing [35] or LLM agents [20], symbolic/concolic execution remains fundamentally limited by the computational cost of constraint handling and path explosion in real-world software.

The above two techniques are *complementary* in nature [2]: fuzzing offers scalable exploration but lacks precision on complex conditions, while symbolic execution provides that precision at the expense of scalability. This complementarity motivates hybrid approaches that combine both strengths.

Hybrid fuzzing, pioneered by Driller [5], is the primary example. Modern hybrid fuzzers apply a fuzzer-centric paradigm: greybox fuzzing drives the testing campaign until it encounters *roadblocks* (i.e., branches difficult to satisfy through random mutations), typically indicated by *plateaued coverage* within a temporal window. At that point, the symbolic executor takes over to bypass these *immediate* roadblocks, after which fuzzing *eagerly* resumes until its coverage stagnates again. This iterative switching enables hybrid fuzzers to reach coverage that neither tool alone can achieve.

Within this paradigm, substantial subsequent research has focused on advancing the individual testers, particularly the symbolic engine (ⓑ in Figure 1A). This includes improving execution throughput [6], [8], [13], [14], [36], [37] and constraint-solving capabilities [7], [12], as well as directly solving constraints via LLMs, especially for textual inputs, as demonstrated in HLPFuzz [38] and HyllFuzz [39].

A. Coordination Strategy

Orthogonal to individual tester improvements, the coordination strategy, i.e., how the two complementary testers collaborate, also significantly affects hybrid fuzzing efficacy [15]. This necessitates **a coordinator that decides**:

(1) the *direction*—where to spend testing budget, and

(2) the *dispatch*—given the direction, how to allocate work between the two testers to bring out their synergy.

Existing Efforts. Existing hybrid fuzzers employ a *fuzzer-centric* coordination paradigm (Figure 1A), where coordination essentially revolves around the fuzzer: its local coverage view dictates the exploration *direction* (ⓐ), and its coverage stagnation triggers *dispatch* to symbolic execution (ⓒ).

Direction is largely predetermined in this paradigm: the exploration simply follows the underlying fuzzer's scalar coverage signal, greedily pursuing any positive coverage "reward" from its coarse edge-level feedback [5], [16], [21]. Pangolin [12] and later CoFuzz [15] attempt to align the two testers' directions by constraining fuzzing's mutations to preserve symbolically solved constraints, though neither proposes mechanisms for steering the campaign as a whole.

Dispatch, by contrast, has attracted more attention, primarily in refining the seed selection strategy, i.e., which seeds are forwarded to the concolic executor for further exploration. To this end, DigFuzz [11] prioritizes "harder" seeds based on estimated path difficulty from a lightweight statistical model; MEUZZ [16] generalizes this idea by incorporating a richer set of syntactic features into the prediction. SAVIOR [40] similarly falls into this class and prioritizes seeds that are more likely to lead to vulnerabilities. Subsequently, CoFuzz [15] shifts the selection granularity from individual seeds to frontier branches, targeting a subset of frontier branches with concolic execution to avoid redundant edge exploration. While these approaches progressively refine *what* to dispatch, they leave unchallenged the assumption that fuzzing alone should decide *when and where* dispatch occurs—i.e., upon the fuzzer's coverage stagnation *and* at the fuzzer's frontier.

B. Motivation

We illustrate two key inefficiencies of a fuzzer-centric paradigm with Figure 2. In this example, `run_program()` takes a command line `cmdline` (comprising a command and its arguments) along with `cmd_hash` for integrity validation. It parses `cmd` and `arg` from `cmdline` (lines 2–3), validates `cmd` through three checks (c_1/c_2/c_3, lines 4–12), and only then executes the core logic `run_command()` (line 13).

Biased Direction. The first inefficiency stems from grey-box fuzzing's inherent local view of progress. Suppose `parse_arg()` at line 3 is a branch-rich function for parsing various command arguments and combinations. A fuzzer may become trapped in this peripheral code: each new edge or hit-count bucket yields a positive "reward," continuously incentivizing the fuzzer's interest. Consequently, the fuzzer lingers in low-value regions for a long time before eventually stalling at the checks on `cmd` (lines 4–12), oblivious to the core logic

```
1  int run_program(char *cmdline, int cmd_hash) {
2    char *cmd = parse_cmd(cmdline);
3    char *arg = parse_arg(cmdline);// branch-rich
4    if (hash(cmd) != cmd_hash) {    // c1: hash check
5      return -1;
6    }
7    if (strlen(cmd) < 4) {          // c2: length check
8      return -1;
9    }
10   if (!isalpha(cmd[0])) {         // c3: alphabetic check
11     return -1;
12   }
13   run_command(cmd, arg);          // core logic
14   return 0;
15 }
```

Fig. 2. A motivating example: the function parses `cmdline` and applies three checks ($c_1 \to c_2 \to c_3$) to `cmd` before running the core logic. Under a fuzzer-centric paradigm: (1) the fuzzer's *biased search* gets trapped in the branch-rich but peripheral `parse_arg()`, oblivious to the core logic; (2) the *reactive* dispatch mechanism makes the fuzzer eagerly resume after each check is symbolically solved, but its mutations regress previously solved checks due to constraint inter-dependencies, causing wasteful alternation (see Table I).

TABLE I
REACTIVE-DISPATCH TRACE OF TRADITIONAL HYBRID FUZZING ON FIG. 2; SATISFYING c_1–c_3 REQUIRES 3 ALTERNATIONS.

Phase	Test Case (*T.C.*)		Checks Passed?		
	cmd	cmd_hash	c_1	$\to c_2$	$\to c_3$
❶ *F.* mutates $\text{Seed}_{\text{Init}}$	12	0xFFFF	✗	–	–
❶ *S.E.* solves c_1	12	0xABCD	✓	✗	–
❷ *F.* mutates ❶'s *T.C.*	1234	0xABCD	✗	–	–
❷ *S.E.* solves $c_1 \wedge c_2$	1234	0xCAFE	✓	✓	✗
❸ *F.* mutates ❷'s *T.C.*	a234	0xCAFE	✗	–	–
❸ *S.E.* solves $c_1 \wedge c_2 \wedge c_3$	abcd	0xBEEF	✓	✓	✓

In each phase, Fuzzer (*F.*) mutates to try to satisfy the next check but regresses previously solved ones due to constraint inter-dependencies, getting stuck and necessitating Symbolic Execution (*S.E.*) to advance the frontier.

in `run_command()` that lies beyond. However, such extensive exploration is *unnecessary* at an early stage of testing, since `arg` is not used at all unless the checks on `cmd` pass at line 13.

A better coordination plan for this example would be to reach `run_command()` first, after which mutations in `arg` could cover code in *both* `parse_arg()` and `run_command()` simultaneously. At its core, formulating such a plan requires a holistic, semantic understanding of the code, an ability beyond the reach of a fuzzer armed only with scalar coverage metrics.

Reactive Dispatch. The second inefficiency arises from the reactive switching mechanism, where fuzzing serves as the default and symbolic execution is triggered only when the fuzzer is stuck. Consider again Figure 2, where three checks c_1/c_2/c_3 guard the deep logic. Among them, only the hash check (c_1) is genuinely difficult for fuzzing; the length (c_2) and alphabetic (c_3) checks, considered in isolation, are manageable for fuzzers. Hence, one may expect an *ideal* (but not realizable!) workflow requiring only a *single* dispatch to the symbolic executor (S.E.):

$$\xrightarrow{\text{start}} [\textit{Fuzz}] \xrightarrow{\text{stall on } c_1} [\textit{S.E.}] \text{ solve } c_1$$
$$\xrightarrow{\text{resume}} [\textit{Fuzz}] \text{ pass } c_2 \text{ and pass } c_3 \to [\checkmark\ \textit{Done}]$$

However, this workflow *fails* to materialize in practice

due to constraint ***inter-dependencies***! As is prevalent in real-world programs, the three consecutive checks $c_1 \rightarrow c_2 \rightarrow c_3$ all constrain the same `cmd` variable, so any mutation to satisfy further checks likely invalidates the previous ones. Specifically, the actual process is illustrated in Table I. In phase ❶, the fuzzer cannot bypass the hash check (c_1) through random mutation, so after a stagnation timeout, it resorts to symbolic execution, which solves c_1 by producing a seed with a matching `cmd_hash` (e.g., `cmd="12"`, `cmd_hash=0xABCD`). Once this immediate roadblock is solved, the fuzzer eagerly resumes and mutates `cmd` to satisfy c_2 (e.g., `"12"`→`"1234"`). However, while the mutated `cmd` satisfies c_2's length requirement (line 7), it inevitably causes a `cmd_hash` mismatch, regressing c_1. With the fuzzer stuck again, symbolic execution intervenes (phase ❷). Based on the current testing frontier (i.e., phase ❶'s test case that already satisfies c_1), symbolic execution extends the frontier by additionally solving c_2—jointly satisfying $c_1 \wedge c_2$—via a new `cmd`/`cmd_hash` pair (e.g., `"1234"`/`0xCAFE`). The same stall-wait-solve cycle repeats for c_3 (phase ❸), ultimately resulting in ***three rounds of alternations***:

$\xrightarrow{\text{start}}$ [*Fuzz*] $\xrightarrow{\text{stall on } c_1}$ [*S.E.*] solve c_1
$\xrightarrow{\text{resume}}$ [*Fuzz*] $\xrightarrow{\text{stall on } c_1}$ [*S.E.*] solve $c_1 \wedge c_2$
$\xrightarrow{\text{resume}}$ [*Fuzz*] $\xrightarrow{\text{stall on } c_1}$ [*S.E.*] solve $c_1 \wedge c_2 \wedge c_3$
$\longrightarrow$ [✓ *Done*]

This repeated alternation wastes considerable testing budget. In each phase, the fuzzer makes *no* actual progress—all advancement comes from symbolic execution—yet consumes significant time waiting for the stagnation timeout before each handoff. The root cause is the *reactive* dispatch mechanism: symbolic execution solves only the *immediate* roadblock upon each handoff and then the fuzzer eagerly resumes, blind to the inter-dependencies among downstream constraints, leaving the complementary strengths of both testers underutilized.

Instead, an effective orchestrator should *proactively* match tasks to testers based on their nature: recognizing that c_1–c_3 form an inter-dependent cluster, it would *directly* dispatch *all* of them to symbolic execution in one go, thereby avoiding the costly stall-wait-solve cycles and clearing the roadblocks for fuzzing to explore `run_command()`.

C. An Agentic Paradigm

To address the aforementioned inefficiencies, we propose *agentic orchestration for hybrid fuzzing*: rather than treating fuzzing as the primary coordination driver, this paradigm elevates both techniques to equal partners under a standalone orchestrator that proactively coordinates them. Realizing this paradigm presents two key challenges.

C1: Understanding the Testing Campaign. Informed *direction* decisions require a semantic, holistic understanding of the testing campaign, yet existing testing artifacts provide only syntactic, snapshot-level evidence. First, the artifacts capturing the current state are voluminous yet undifferentiated. A typical campaign produces thousands of test inputs, each covering hundreds of code lines: presenting all artifacts at once would overwhelm any decision-maker with no sense of priority on what matters most, while aggregating them into a scalar (e.g., a coverage number, as in existing tools) discards the very detail on which direction decisions hinge. Second, the artifacts are snapshots, not narratives. Coverage and seeds record only what currently exists, not the intricate patterns of how the campaign arrived there. This temporal context, essential for adapting strategy over time, is simply not preserved.

C2: Dispatching Testers Smartly. Existing testers have no native awareness of strategic intent and are inherently greedy—fuzzers optimize for local coverage gains while symbolic executors focus on immediate constraint solving. Besides, their dynamics are equally opaque: random mutations and greedy branch selections leave no traceable rationale, making it challenging to assess whether they adhered to the intended direction or drifted away. Naively interleaving them may cause their efforts to overlap or even conflict. Effective coordination thus requires careful dispatching to achieve the global objective.

Our Approach. To address **C1**, we design a layered information architecture that balances detail with manageability: high-level summaries prevent information overload, while specialized tools enable on-demand access to fine-grained details of the testing artifacts (§III-A). To capture the campaign trajectory, we maintain a structured representation as a sequence of ⟨*goal, achievement*⟩ tuples, enabling the agent to recognize which attempts succeeded, which failed, whether progress is accelerating or diminishing, and whether certain testers have been effective on particular types of tasks (§III-B). For **C2**, we design a seed selection mechanism that translates strategic intent into concrete starting points that testers can act upon, and collect structured feedback to surface tester dynamics for the orchestrator (§III-C). In the next section, we explain our system design in detail.

III. System Design

Figure 3 presents the system overview of AHFuzz. Similar to conventional hybrid fuzzers, AHFuzz takes in the source code of the system under test (SUT) and an initial seed, and outputs generated test cases and triggered bugs. However, unlike traditional fuzzing-driven approaches, AHFuzz introduces an intelligent orchestration layer that actively and strategically steers the testing campaign. The framework integrates two heterogeneous testers, namely **AFL++** [21], the *de facto* standard greybox fuzzer, and **ConcoLLMic** [20], a state-of-the-art concolic execution engine applicable to both textual and binary inputs. ConcoLLMic follows the standard summarize-and-solve pipeline with LLM assistance to synthesize inputs reaching alternative paths from a given seed.

Underpinning the testing process is the global state structure shared across all components (§III-A). Beyond the source code itself, this structure maintains two critical data repositories: (1) a coverage map that tracks explored code regions, and (2) a seed pool of promising test inputs. Both repositories can be updated by either tester, with an actively maintained *mapping* between per-line coverage statistics and retained seeds

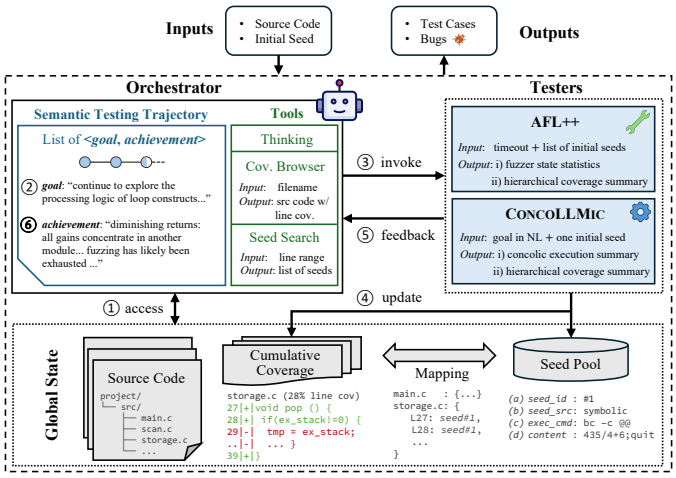


Fig. 3. System overview of AHFUZZ, consisting of (1) an agentic orchestrator, (2) two heterogeneous testers, AFL++ and CONCOLLMIC, and (3) a shared global state. Based on past experience and supporting tools to ① *access* the global state, the orchestrator iteratively formulates a testing ② *goal* and ③ *invokes* a suitable tester to pursue it. The invoked tester ④ *updates* the global state and provides structured ⑤ *feedback*. The orchestrator then reflects on the ⑥ *achievement* and appends it to the *testing trajectory*, a growing semantic history of the campaign that informs subsequent decisions.

throughout the campaign. The agentic orchestrator accesses the global state through a suite of supporting tools that provide structured interfaces for code/coverage browsing and seed retrieval, allowing it to make informed decisions based on comprehensive testing state understanding.

Workflow. The agentic orchestrator operates through iterative goal-directed tester invocations (§III-C). In each cycle, upon examining the current testing progress (①), it formulates a *goal* in natural language to document the intended testing outcome (②). These goals typically specify high-level exploration targets such as code locations (e.g., under-explored modules) or structural constraints on inputs (e.g., specific field values). It then invokes a suitable tester to pursue the goal (③), by supplying initial seeds that best align with the testing objective. By carefully selecting which seeds to provide, the orchestrator effectively steers individual testers along a coherent global trajectory rather than allowing greedy local exploration. The invoked tester starts from the provided seeds and runs until it finishes or its allotted testing budget is exhausted, updating the global coverage and seed pool during its execution (④). Critically, after each invocation, the agent then assesses the extent to which the goal has been *achieved* (§III-B). This reasoning draws on the tester's feedback (⑤), distilled from its output and status summaries, as well as the updated global state accessed via supporting tools. The assessment characterizes the testing phase in concise natural language, capturing what was attempted and accomplished in this round of testing. Both the goal and its corresponding achievement are appended to the testing trajectory as a tuple of ⟨*goal, achievement*⟩ (⑥), enabling the agent to gradually build a distilled summary of the entire campaign.

A. Global State Tracking

Key Data Structures. The global state is the central repository of testing artifacts, collectively capturing the outcomes of the testing campaign in its fullest detail. It contains the *source code* and *three* interconnected data structures:

(1) Seed Pool. First, it maintains a seed corpus consisting of all retained test inputs, each stored alongside its execution command. This captures the usage of the seed and any other contextual parameters necessary to reproduce the exact execution, such as command-line options and environment variables. We note that this execution command remains unchanged during each invocation of AFL++, but may be distinct for each symbolically generated seed thanks to CONCOLLMIC's capability in environment manipulation [20].

(2) Cumulative Coverage. Second, the global state tracks fine-grained line coverage accumulated throughout the testing campaign. Unlike the edge-based bitmap used internally by AFL++ or the block-level tracking employed by CONCOLLMIC, AHFUZZ maintains unified, source-line-level coverage by replaying generated test inputs with GCov [41] instrumentation enabled. This design choice trades some runtime overhead for heightened visibility into code exploration, enabling the orchestrator to reason about coverage at the granularity of individual statements.

(3) Mapping. Finally, the system establishes an explicit mapping between seeds and their coverage contributions. During the GCov-instrumented replay, each seed is mapped to the specific source lines it executes. In particular, to mitigate the slowdown from coverage tracking for voluminous fuzzer-generated seeds, AHFUZZ tracks only the *delta coverage* for each seed, i.e., the set of executed lines that have not been previously covered by any earlier seed. This mapping represents a compact summary of whether each seed is valuable, and it enables efficient seed retrieval to facilitate targeted exploration of specific program regions.

Supporting Tools. While these data structures capture a comprehensive testing state, the sheer volume of artifacts poses a significant challenge for agentic reasoning. A typical fuzzing campaign may accumulate thousands of test cases with each covering thousands of lines across multiple source files. Even with the delta-coverage optimization, this would still overwhelm the conversation context of the agent. AHFUZZ addresses this challenge through a two-tier information architecture: high-level summaries are automatically provided to the orchestrator after each cycle (detailed in §III-C), while fine-grained details are accessed on demand through specialized supporting tools.

(1) Coverage Browser. To bridge the gap between covered source lines and covered behaviors, the agent may invoke a coverage browser to examine source code annotated with coverage statistics. This tool allows the orchestrator to inspect a specific file to understand what code has been explored rather than merely tracking numerical coverage percentages.

(2) Seed Search. To navigate the potentially vast seed corpus and extract relevant inputs for goal-directed testing, the orchestrator is equipped with a seed search tool that provides

structured, query-based access to the seed pool. This tool accepts a line range specification (e.g., lines 150–200 of `scan.c`) and retrieves seeds whose execution reached code within that range. To balance diversity with efficiency, the tool performs randomized sampling when multiple candidates exist. Each retrieved seed is enriched with metadata to support informed decision-making. This includes (a) the seed identifier, (b) the source tester, (c) its associated execution environment, (d) the actual input content, and finally (e) its coverage contribution breakdown by file, calculated from its delta-coverage mapping. Collectively, these agentic access capabilities enable the agent to reconstruct the testing trajectory from low-level artifacts.

B. Strategic Goal Direction

Beyond code comprehension assisted by the tools, effective tester orchestration requires the agent to understand the testing trajectory holistically and work toward *high-value* code regions that are of functional significance and/or high future coverage potential. Operationally, this is achieved via iterative cycles of tester invocations, each guided by a strategic goal formulation (② in Figure 3) and reflection process (⑥).

Goal Formulation. In each cycle, the agent first formulates a *goal* based on the global state (§III-A) and historical progress before actually invoking a tester. This goal, recorded in natural language, identifies promising exploration directions—for example, under-explored modules, regions guarded by complex conditional branches, or paths dependent on key environment constraints. For instance, when testing a language interpreter, the orchestrator might set a goal "*continue to explore the processing logic of loop constructs*" to more thoroughly explore the newly reached logic, as shown in Figure 3. We note that the shape and value of such a goal can be subject- and testing-state-specific; as such, before proposing the goal (②), AHFUZZ is provided with agentic access (①) to program sources, testing artifacts, and the campaign trajectory to ground its assessment.

The goal serves two critical purposes for intelligent dispatch (§III-C). First, it provides a contextual anchor that guides the agent's reasoning about which tester is best suited for the current objective. For instance, a goal targeting intricate program logic or requiring manipulation of environment variables naturally suggests invoking CONCOLLMIC. Conversely, a goal seeking to explore close neighborhoods of an interesting execution path may favor AFL++'s mutation-based approach. Second, the goal informs the selection of initial seeds to kickstart the preferred tester. Consider, for example, a targeted program behavior that requires a precise sequence of events to trigger. A seed whose execution produces a prefix of this sequence would be invaluable to the goal at hand. However, that same seed would be counter-productive for reaching behaviors that expect an entirely different event sequence.

Reflection and Assessment. After each tester invocation concludes, the orchestrator assesses the extent to which the stated goal was achieved, distilling the outcome into a concise natural language summary. This *achievement* not merely captures whether new coverage was discovered, but also characterizes what was accomplished in relation to the intended goal. Crucially, the outcome is expressed at a high level of abstraction—not as a binary reachability verdict, but as nuanced judgments such as "partial success", "diminishing returns", or "blocked by complex constraints"—enabling the orchestrator to reason about progress holistically. If the target was not (fully) reached, the assessment evaluates whether continued pursuit of this goal remains worthwhile and identifies the main obstacles, informing the next course of action.

This reflection process is facilitated not only by structured feedback from individual testers (§III-C), but also by autonomous access to the global state through coverage browsing (§III-A), allowing the agent to contextualize raw coverage/progress metrics within the actual code structure. Consider the example in Figure 3 again. After a few rounds of AFL++ invocations, the orchestrator reviews the feedback and consults the coverage report for details, summarizing the achievement as: "*diminishing returns: +12 lines covered (vs. +143 lines last round), all coverage gains concentrated in the execution (instead of processing) module; fuzzing has likely exhausted variations around current syntax constructs.*"

The tuple of ⟨*goal, achievement*⟩ generated in this iteration is eventually appended to construct a structured history of the testing campaign. This history allows the agent to extract insights needed to inform subsequent testing decisions effectively, by revealing useful patterns such as which types of goals have been productive, which program modules remain under-explored, and whether recent testing has been making steady progress or encountering diminishing returns.

C. Agentic Dispatch

With strategic goals formulated, the orchestrator works toward achieving them progressively by dispatching to the best-suited tester. In each iteration, it first selects seeds that align with the objective (① in Figure 3), then configures and invokes the appropriate tester (③–④), and finally collects structured feedback upon completion (⑤). The agentic orchestrator can freely choose between AFL++ and CONCOLLMIC with carefully constructed arguments, arming each tester with maximum leverage for achieving the stated objective.

Seed Selection. For both testers, the orchestrator employs a unified seed selection procedure to identify inputs best positioned to achieve the current goal. To do so, it iteratively invokes the seed search tool described in §III-A. Typically, it first queries for seeds covering the target code region; if none exist, it may cross-reference the current coverage status via the coverage browser, and then broaden its search to adjacent or upstream code regions based on its understanding. This iterative process enables the orchestrator to reason about the code semantics and coverage patterns of each candidate seed, and to contextualize the *difficulty* of targeted constraints instead of relying on a syntactic *count* (e.g., CFG distance, see Appendix D) often used in prior work [16], [42]. In particular, for CONCOLLMIC, the orchestrator provides a *single* seed as the basis for concolic exploration to maximize the gain from its slow and expensive symbolic reasoning. For AFL++, *multiple* initial seeds are provided, allowing the fuzzer to explore from

diverse starting points, and potentially combine these inputs through genetic mutations (e.g., splicing). AFL++ adopts its execution command from the primary (first) seed in the list.

Invoking the Symbolic Executor. CONCOLLMIC [20] is a powerful symbolic executor built with LLM-based agents. While its agentic architecture enables sophisticated reasoning about complex paths, it operates at significantly lower throughput (a few seconds per input) and incurs substantial costs ($0.2 per input) due to API calls to underlying language models [20]. To maximize the gain from slow and expensive symbolic reasoning, the orchestrator issues carefully directed, single-iteration invocations to CONCOLLMIC.

In addition to the seed selection procedure described above, the orchestrator also provides *explicit* guidance to the symbolic executor by passing the testing goal. This is possible since CONCOLLMIC natively understands natural language. During execution, CONCOLLMIC executes the provided seed concretely to collect its dynamic slice with coverage annotations. We pass the orchestrator's goal to CONCOLLMIC as an optional directive, so that it prioritizes target lines conducive to the stated objective when selecting which branches to target. Multiple such target lines are identified, and for each target, CONCOLLMIC negates the necessary branch conditions along the path, summarizes the resulting path constraints, and solves for concrete test inputs that satisfy these constraints. Following the default configuration of CONCOLLMIC [20], multiple inputs are generated in parallel per invocation, amortizing both the time and cost overhead of LLM calls.

Invoking the Fuzzer. The orchestrator guides AFL++ sessions less explicitly, in particular through *on-demand extension* and *cumulative coverage tracking*. In addition to a list of curated seeds, each AFL++ session is allotted a time budget by the orchestrator and operates unmodified in its standard mutation-execution-retention cycle.

When the specified time budget expires, the orchestrator can not only invoke a new round of fuzzing or symbolic execution, but also *extend* the last fuzzing session if it has been fruitful. To support session extension, an expired AFL++ session is *suspended* rather than terminated, preserving the complete fuzzing state, including its input queue, coverage bitmap, and scheduling metadata. The orchestrator can then provide an additional time budget to resume AFL++ from exactly where it paused, without needing to supply initial seeds again. This extension can be applied repeatedly—even with interleaved invocations of the other tester—allowing the agent to maintain continuity in productive fuzzing sessions while remaining responsive to emerging opportunities elsewhere.

Eventually, the orchestrator may determine that the current AFL++ session has exhausted its potential or that a different search direction warrants exploration. At this point, it initiates a fresh fuzzing session with a new list of initial seeds, effectively pivoting the search along a more promising trajectory. Critically, this new AFL++ instance begins with a *pre-populated* coverage bitmap, inherited from the previous session's output directory, that accumulates *all* coverage across historical fuzzing sessions. This design ensures that the fresh session does not waste effort rediscovering explored code regions, but instead expends its budget on pushing the coverage frontier along the newly provided seeds, allowing the orchestrator to steer an otherwise greedy fuzzer to pursue global objectives.

Feedback Collection. Upon tester completion, the orchestrator collects structured feedback that characterizes the exploration outcome. CONCOLLMIC's feedback to the agent is centered around its goal-directed execution: for each target branch it attempted to reach, the summary specifies the selected branch location, whether the target was actually reached during validation, and the new coverage achieved. It also reports the total LLM API cost incurred. On the other hand, AFL++ returns statistics extracted from its internal fuzzer state, including its corpus size, bitmap coverage percentage, and the elapsed time/cycles since the last new path (à la AFL++) was found.

In addition, both testers conclude their feedback with a coverage summary, showing newly covered lines organized *hierarchically* by source file and directory structure. More specifically, the summary follows a top-down tree format, with each line representing the coverage update for either a source file or the entire directory/module. Each line is formatted as "`(+new)[covered]/[total](old%→new%)`", displaying the count of covered and total lines along with the coverage increment in both absolute and percentage terms. For individual source files, newly covered line numbers are reported as compressed ranges (e.g., "`[1,3-5,10]`"), allowing the orchestrator to identify precisely which code regions were newly explored. This hierarchical presentation enables the orchestrator to quickly assess how progress is distributed across the codebase, feeding into the reflection process that determines subsequent testing directions.

IV. IMPLEMENTATION

AHFUZZ is implemented in Python 3 and comprises three main components. **(1) Orchestrator Layer.** We implement the agentic orchestrator atop claude-sonnet-4.5 [43] with temperature set to zero and incremental prompt caching. **(2) Global State.** We implement the global state and its supporting tools as described in §III-A. **(3) Tester Integration.** We implement uniform interfaces for invoking both testers, collecting their structured feedback, and converting their outputs into a unified seed format for the global state. A coverage monitor [44] continuously replays each newly generated seed against a GCov-instrumented [41] binary to track cumulative coverage. For **CONCOLLMIC** [20], we make two minor modifications: (a) since AFL++ is already C/C++-specific, we replace CONCOLLMIC's LLM-based instrumentation with GCov for concrete execution tracking; (b) we extend its prompt with an optional *goal* argument, allowing it to prioritize branches aligned with the orchestrator's objective. Neither modification alters CONCOLLMIC's symbolization or solving core. All other configurations stay at their defaults. **AFL++** [21] is integrated *without* modifications; AHFUZZ configures each fuzzing session via the unified tester interface by providing

a list of initial seeds and a time budget (5–50 min). We note that this design is a deliberate decision to facilitate the *usability* of our tool. Compared to existing hybrid fuzzers that apply custom-built compiler passes, AHFUZZ relies on two stock instrumentation builds, namely AFL++ (for fuzzing) and GCov (for CONCOLLMIC and global coverage tracking). The engineering effort of applying AHFUZZ to new subjects is thus comparable to that of a standard fuzzing setup (Appendix D).

V. EVALUATION

In this section, we aim to understand the effectiveness of AHFUZZ through an extensive evaluation. After presenting our **evaluation setup** (§V-A), we demonstrate AHFUZZ's effectiveness by comparing the achieved **code coverage** (§V-B) and **bug discovery** (§V-C) against state-of-the-art fuzzers. We then examine the **orchestration's effectiveness** (§V-D) by creating four ablative variants with reduced scheduling intelligence. Finally, we present **AHFUZZ's key statistics** (§V-E) to better understand its remarkable performance.

A. Evaluation Setup

Comparison Tools. We compare AHFUZZ against six leading fuzzers, including the latest versions of:

1) **SymSan++** [8], the hybrid version of SymSan [8] and AFL++, where SymSan is a state-of-the-art concolic executor with optimized constraint collection. Both testers run in parallel and collaborate through periodic seed syncing.
2) **MEUZZ** [16], a hybrid fuzzer using QSYM [6], a specialized concolic executor for hybrid fuzzing. It employs machine-learned heuristics to select seeds and prioritize which roadblocks to delegate to symbolic execution.
3) **CoFuzz** [15], also a hybrid fuzzer using QSYM as the concolic executor. It employs edge-oriented scheduling to minimize the exploration overlap between the two testers.
4) **AFL++** [21], the underlying greybox fuzzer in AHFUZZ, running in its default configuration. Since hybrid fuzzers typically run a fuzzer and a concolic executor in parallel, we run *two* AFL++ instances in parallel mode to provide comparable computational resources.
5) **Advanced AFL++**, an advanced AFL++ configuration enabling two additional features: (1) DICT [45]: automatic dictionary generation that extracts string comparison tokens at compile time; and (2) CMPLOG [46]: Redqueen-style comparison logging [47] designed to bypass complex comparisons and "magic bytes" that traditional random mutations struggle to solve. As with AFL++, we run *two* such instances in parallel mode. We include this baseline to assess whether advanced features alone can replace a hybrid approach with symbolic reasoning.
6) **autofz** [22], a meta-fuzzer that dynamically allocates CPU resources to an ensemble of baseline testers based on each tester's runtime coverage trend (see §VII). We use its default configuration in the evaluation.

These baselines are recent representative work with established efficacy. Notably, they span distinct coordination strategies: static syncing, machine-learned scheduling, edge-oriented scheduling, and dynamic resource allocation. We further isolate the effect of orchestration with ablative variants in §V-D.

TABLE II
DETAILS OF REAL-WORLD BENCHMARKS. THESE SUBJECTS EXERCISE DIVERSE CONSTRAINT CHARACTERISTICS: BINARY STRUCTURE/DATA ENCODING, STRUCTURED-FORMAT PARSING, ARITHMETIC/SYMBOLIC EXPRESSIONS, AND PATTERN MATCHING.

Subject	Version	LoC	Input Type (Format)	Source
`objdump`	2.45	31,516	*Binary* (executable)	[8], [15], [16]
`readelf`	2.45	32,907	*Binary* (ELF file)	[8], [15], [16]
`libjpeg`	#117795	138,052	*Binary* (JPEG file)	[8], [15], [16]
`woff2`†	#f83a17	56,279	*Binary* (font data)	[8], [20]
`lcms`	#23994d	104,301	*Binary* (color data)	[8]
`oggenc`	#abdfdf	69,303	*Binary* (audio data)	[20]
`cisv`*	#3103b5	11,572	*Textual* (CSV file)	–
`tomlc17`*	#f6c98d	4,180	*Textual* (TOML file)	–
`libxml2`	v2.15.1	281,961	*Textual* (XML file)	[8], [15], [16]
`libyaml`	#840b65	11,311	*Textual* (YAML file)	[20]
`confetti`*	v1.0.0	5,310	*Textual* (config. file)	[20]
`proj4`	v9.8.1	327,262	*Textual* (map projection)	[8]
`libsoup`	#54f945	71,165	*Network* (HTTP msg.)	[20]
`libmatheval`	1.1.12	11,920	*Textual* (math expr.)	[20]
`bc`	1.08.2	29,348	*Textual* (math script)	[20]
`krep`*	#d3e5e6	6,329	*Textual* (regex)	[20]

* indicates subjects created after claude-sonnet-4.5's training cutoff [48].
† `woff2` is compiled with `-DBROTLI_BUILD_PORTABLE` to disable intentionally unsafe memory optimizations that would otherwise trigger UBSan [49].

Benchmarks. As shown in Table II, we evaluate AHFUZZ on 16 real-world subjects, with varying codebase sizes and characteristics. All subjects (except two post-training ones) are drawn from the evaluation suites of comparison tools [8], [15], [16] or base testers [20], as detailed in the last column. We reuse their existing fuzzing configurations for comparability. In particular, `proj4`'s harness is sourced from [3] since the original harness used in [8] has been retired. Many of these subjects are mature open-source projects undergoing continuous fuzzing in OSS-Fuzz [3]. Beyond spanning diverse input formats (*binary*, *textual*, and *network*), these subjects exercise *four* distinct ***constraint characteristics*** that demand different testing strategies: (1) *binary structure and data encoding*, where inputs conform to either rigid binary layouts (e.g., magic bytes and section offset tables in ELF files) or encoded/compressed data formats (e.g., ICC color profiles and audio codecs); (2) *structured-format parsing*, where inputs follow format-specific grammars such as nested delimiters (e.g., XML tags), indentation-sensitive syntax (e.g., YAML), key-value hierarchies (e.g., TOML), or protocol state machines (e.g., HTTP messages); (3) *arithmetic and symbolic expressions*, where inputs represent mathematical expressions involving operator precedence, variable scoping, and user-defined function calls; and (4) *pattern matching*, where inputs are patterns that undergo compilation and constrained search. These collectively evaluate the orchestrator's ability to recognize the nature of constraints and dispatch the better-suited tester under diverse scenarios. To mitigate data leakage risks arising from LLM use, we follow [20] and additionally include four well-maintained projects created *after* claude-sonnet-4.5's training cutoff [48], i.e., `cisv`, `confetti`, `krep`, and `tomlc17`.

Evaluation Environment. Each campaign runs in a Docker

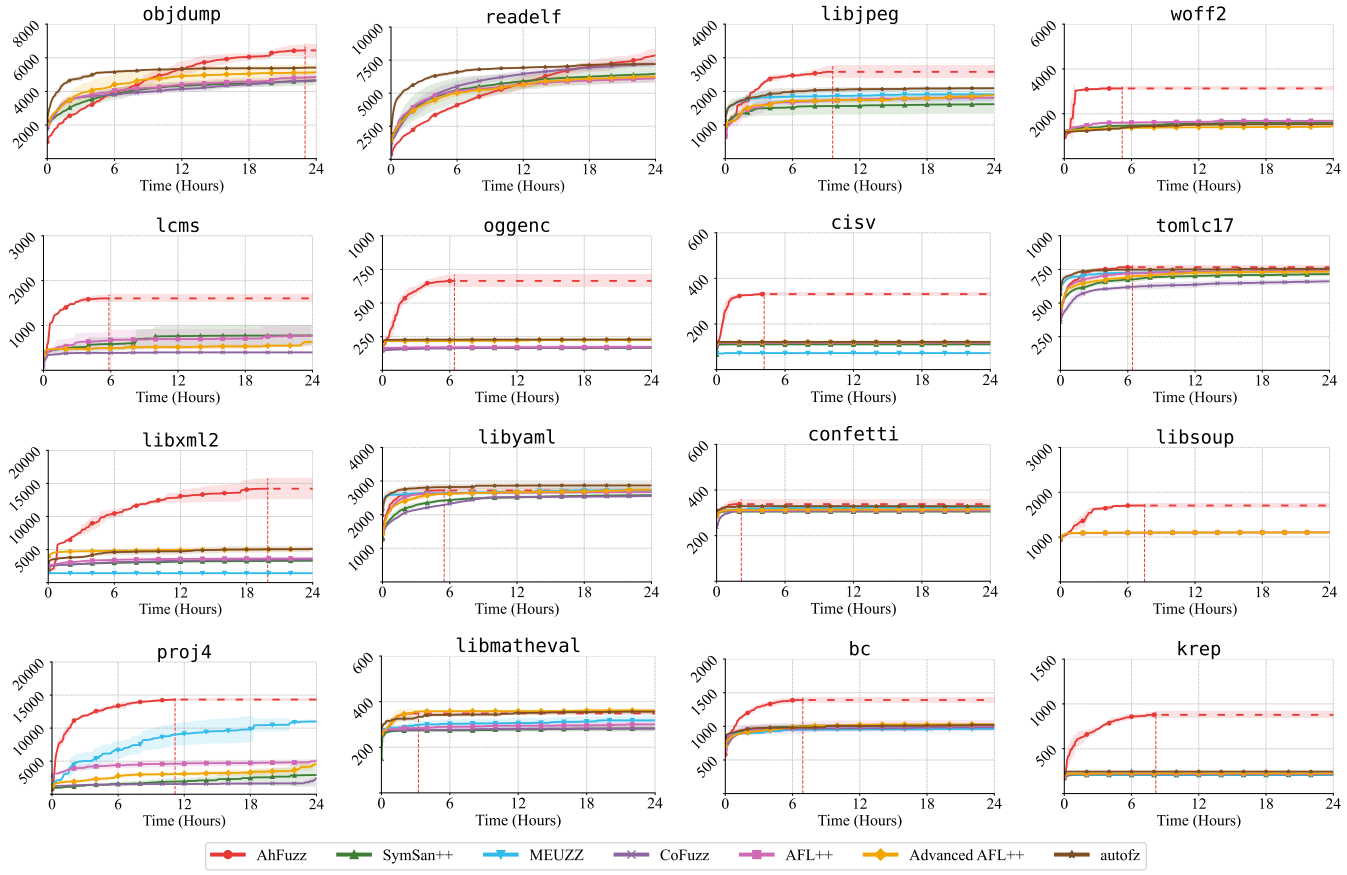


Fig. 4. GCov branch coverage growth over time. Solid lines and shaded areas show the mean and standard deviation across repetitions. AHFUZZ automatically exits after 30 min of coverage stagnation, averaging only 7.93 h per run; *dashed segments* thereafter indicate AHFUZZ is *no longer running*. Even with this shorter runtime, AHFUZZ attains higher final coverage than *all* competing tools on 14 subjects with statistical significance (p-value < 0.05).

container with 2 CPUs and 8 GB RAM. Each tool is run 10 times per subject to ensure statistical significance. Each campaign is capped at 24 h, except that AHFUZZ *may terminate early*—we follow [20] and configure it to exit automatically once coverage has stagnated for over 30 min, as AHFUZZ is more costly to run for an extended period due to LLM API calls. In total, our evaluation amounts to 4 years of CPU time.

Metric. While each tool internally uses its own instrumentation, we measure all tools under one consistent metric: branch coverage on the same GCov-instrumented binary [41], obtained by replaying each tool's generated test cases.

B. Efficacy of Test Generation

We first compare AHFUZZ against baselines to measure the efficacy of test generation by visualizing the GCov [41] branch coverage growth over time in Figure 4. We also report the raw coverage results in Appendix A. Here, missing curves indicate tool failures on specific subjects, and dashed lines indicate where AHFUZZ exited early due to 30 min of coverage stagnation. Overall, AHFUZZ significantly outperforms all comparison tools. On average, AHFUZZ's final branch coverage is 2.26×, 2.66×, 2.60×, 2.08×, 2.01×, and 1.84× that of the 24 h runs of SymSan++, MEUZZ, CoFuzz, two instances of AFL++, Advanced AFL++, and autofz, respectively. This average is computed as the arithmetic mean of per-benchmark final branch coverage ratios (see Appendix A). With *statistical significance* (Welch's t-test, p-value < 0.05), AHFUZZ's final coverage is higher than that of *all* baselines on 14 subjects. The two exceptions are `libmatheval`, where AHFUZZ is comparable to Advanced AFL++ and autofz while outperforming the other baselines, and `libyaml`, where all fuzzers saturate at a similar level. We attribute the impressive performance of AHFUZZ partly to its symbolic executor [20] and partly to its orchestration decisions on *which* tester to invoke, *how* to configure it, and *when* to pivot. The two factors are disentangled by the ablation study (§V-D), and the orchestration decisions are further examined per subject in §V-E. On the other hand, the variation in AHFUZZ's relative margin correlates with the complexity of the target subject. For smaller subjects like `libyaml` and `libmatheval`, fuzzing alone quickly saturates coverage, leaving limited room for improvement. Conversely, large and complex subjects such as `proj4` and `objdump` feature deep regions that are challenging for individual engines, and thus orchestration pays off most.

Remarkably, AHFUZZ achieves these results with a shorter

runtime. The duration of AHFUZZ's testing campaigns averages only 7.93 h, less than 1/3 of the baselines' 24 h budget. This demonstrates its efficacy in covering more code regions faster. We present a detailed analysis of key statistics, such as time, costs, and tester invocation patterns, in §V-E. The coverage curves also reveal that AHFUZZ maintains a more sustained growth rate throughout the campaign. While competing tools often exhibit rapid initial progress that quickly decelerates, AHFUZZ continues to discover new code regions at a steady pace. This is particularly evident on long-running subjects such as `readelf`, `objdump`, and `libxml2`, where baseline tools plateau early yet AHFUZZ sustains coverage gains until campaign termination. In particular, we highlight that AHFUZZ retains its efficacy for benchmarks expecting binary inputs (e.g., `libjpeg`), which tend to be more challenging for LLMs to reason about. We also observe no degradation in performance for newly created projects whose codebases are absent from claude-sonnet-4.5's training corpus [48].

C. Bug Discovery

Beyond code coverage, we evaluate AHFUZZ's bug detection capabilities against comparison tools. To enable systematic and fair bug detection, we instrument all benchmarks in Table II with ASan [50] and UBSan [49] to capture memory safety and undefined behavior violations. Critically, *all* tools are evaluated on the *same* sanitizer-instrumented binaries to ensure fair comparison. The primary detection signals are program crashes and hangs. Each discovered bug is validated through concrete execution using the generated input and associated environment context, ensuring no false positives. We follow standard procedures to deduplicate bugs by analyzing sanitizer or GDB stack traces.

Results. Table III summarizes the previously unknown vulnerabilities discovered by AHFUZZ. In total, AHFUZZ uncovered 13 bugs across 7 well-tested projects, spanning a diverse range of vulnerability types. At the time of writing, 9 vulnerabilities have been confirmed or fixed by the developers. Bugs #5 and #6 in `libsoup` have already been assigned CVE identifiers, while the remaining bugs are undergoing the CVE application process. These subjects represent mature, production-grade software: `bc` is a POSIX-standard calculator shipped in most Unix-like systems, `libmatheval` is a widely used GNU library for symbolic expression evaluation, and `libsoup` is GNOME's default networking library that underpins many applications. Notably, AHFUZZ has also discovered bugs in projects created *after* claude-sonnet-4.5's knowledge cutoff (i.e., `cisv` and `krep`), demonstrating that the effectiveness of agentic orchestration does *not* depend on the LLM's prior exposure to the codebase. Despite being a recent project, `krep` has garnered over 450 GitHub stars and is actively used in production environments, yet AHFUZZ uncovered 3 new bugs, all of which have been fixed.

Comparison. The 13 bugs discovered by AHFUZZ are difficult to trigger with other tools. SymSan++, MEUZZ, CoFuzz, AFL++, Advanced AFL++, and autofz exposed only 2, 1, 0, 2, 3, and 4 of them, respectively, and *none* discovered additional bugs beyond AHFUZZ's findings. Through manual inspection, we find that AHFUZZ's unique findings often require stringent inputs that are (1) *well-formed* enough to drive execution into a rarely exercised path, e.g., a multi-range header in bug #6, a valid function definition in bugs #8/#9, or a deep protocol construct in bug #5; yet (2) *adversarial* due to a counter-intuitive perturbation, e.g., a type mismatch in bug #1, an underflowing value in bug #2, or a resource-exhaustion condition in bug #10. AHFUZZ curates these inputs through intelligent, proactive task-tester matchmaking, which we illustrate further with the case study below.

Case Study: Bug #5. We present a detailed analysis of bug #5 in `libsoup`. Due to its severity, its disclosure earned a €5,000 bounty under the GNOME Bug Bounty Program after the bug was confirmed by the upstream developers. The bug resides on libsoup's HTTP/2 request handling path, which is not activated by the initial seed or shallow exploration. Upon observing from the coverage browser that the entire HTTP/2 server I/O module remained uncovered, the orchestrator formulated the goal of reaching this critical logic and directed CONCOLLMIC toward this goal, leveraging its rigorous path reasoning capability beyond what random mutation could provide. However, reaching the HTTP/2 path was only the first step. The bug itself was subtle, surfacing only under counter-intuitive input mutations that semantic reasoning may miss. With this high-quality seed as a starting point, AHFUZZ then strategically dispatched AFL++ to explore variations along this path, eventually triggering the use-after-free. Notably, neither AFL++ nor CONCOLLMIC alone discovered this bug: AFL++ lacks constraint reasoning, while CONCOLLMIC lacks high-throughput exploration of extreme values. Moreover, the other baselines and AHFUZZ's ablative variants (see Table VI) also failed to discover this bug, demonstrating that realizing such synergy requires intelligent orchestration that delegates tasks based on their nature and the strengths of the testers.

D. Ablation Study

Setup. As discussed at length in §II-C, AHFUZZ deviates from conventional approaches in two ways: (1) avoiding a fuzzer-centric design and (2) realizing smart orchestration with an LLM agent. To isolate the effect of both aspects, we implement four ablative variants of AHFUZZ:

(1) **STATIC** mimics fuzzer-centric approaches following the same design as previous hybrid fuzzers [6], [15], [16]. Specifically, AFL++ runs as a persistent session, while STATIC maintains a FIFO seed queue and a global coverage state by periodically syncing seeds from AFL++. When coverage plateaus for 5 minutes (following [11]), STATIC invokes CONCOLLMIC for concolic execution: it fetches the next seed from the FIFO queue (following [8]), executes it symbolically, and generates new inputs. If the generated inputs cover new code, they are shared with AFL++ to mutate further.

(2) **VELOCITY** keeps STATIC's reactive, fuzzer-centric design but replaces its rigid stall timer with a velocity-based trigger. It monitors the *coverage derivative* over a sliding window and preempts the fuzzer when the coverage velocity

TABLE III
PREVIOUSLY UNKNOWN VULNERABILITIES EXPOSED BY AHFUZZ. BASELINES DETECTED AT MOST 4 OF THEM WITH NO ADDITIONAL FINDINGS.

#	Subject	Bug Description	Status
1	`lcms`	NULL pointer dereference in `cmsopt.c` due to the unchecked return value during pipeline optimization	Fixed
2	`oggenc`	Undefined behavior in `audio.c` caused by the left shift of negative values when processing malformed WAV files	Confirmed
3	`cisv`	Memory corruption in `parse_scalar()` due to illegal pointer cast when parsing CSV files with malformed quotes	Fixed
4	`cisv`	NULL pointer deref. in `ensure_quote_buffer()` due to unchecked `realloc()` when processing large quoted fields	Confirmed
5	`libsoup`	Heap use-after-free due to `io` freed by `got-headers` signal handler before callback returns during HTTP/2 request handling	Confirmed
6	`libsoup`	Integer overflow in `sort_ranges()` comparator silently drops ranges from multi-range HTTP responses	Confirmed
7	`libmatheval`	Segmentation fault in `node.c` due to unhandled two-argument function node when computing min/max derivatives	Reported
8	`bc`	Heap buffer overflow in `process_params()` when calling functions with scalar arguments where arrays are expected	Reported
9	`bc`	NULL pointer deref. in `process_params()` when calling functions with array arguments where scalars are expected	Reported
10	`bc`	Infinite loop in `scan.l` due to `input()` blocking instead of returning EOF on unclosed multi-line comment	Reported
11	`krep`	Heap buffer overflow in `memchr_short_search()` when searching short patterns with whole-word mode	Fixed
12	`krep`	NULL pointer deref. in `main()` due to missing NULL check before `strcmp()` when processing redirected stdin input	Fixed
13	`krep`	Heap buffer overflow in `regex_search()` when `regexec` reads beyond `mmap` boundary on files exceeding 64KB	Fixed

TABLE IV
PER-SUBJECT STATISTICS OF AHFUZZ'S TESTING CAMPAIGNS.

Subject	Time (h)*	# Invocations (New Cov.%)†		Cost ($/h)‡	
		AFL++	CONCOLLMIC	Orch.	S.E.
`objdump`	20.21	65.3 (95.5%)	18.0 (75.4%)	1.24	1.01
`readelf`	23.97	71.2 (98.2%)	11.4 (70.3%)	0.73	0.49
`libjpeg`	7.56	27.3 (61.9%)	11.3 (100%)	0.82	1.56
`woff2`	2.99	12.3 (83.9%)	6.0 (34.2%)	0.41	3.25
`lcms`	4.88	20.7 (72.2%)	13.3 (56.1%)	0.62	3.40
`oggenc`	4.80	15.3 (30.2%)	28.3 (84.3%)	0.92	3.25
`cisv`	2.88	13.3 (15.4%)	18.7 (83.4%)	1.20	2.79
`tomlc17`	4.96	24.8 (60.1%)	15.5 (77.6%)	0.78	1.61
`libxml2`	16.03	46.3 (88.6%)	23.7 (93.2%)	2.26	1.96
`libyaml`	4.50	16.7 (67.1%)	13.0 (51.5%)	0.79	1.80
`confetti`	1.78	7.2 (43.7%)	7.3 (53.1%)	0.94	2.06
`proj4`	11.17	8.0 (5.6%)	45.7 (96.4%)	0.91	11.63
`libsoup`	5.10	21.0 (77.7%)	13.5 (75.0%)	0.60	2.51
`libmatheval`	2.59	10.7 (58.3%)	12.1 (69.4%)	0.72	2.33
`bc`	6.41	25.3 (73.8%)	19.3 (92.9%)	0.60	1.61
`krep`	6.99	15.7 (25.8%)	69.3 (58.4%)	1.13	4.99
Average	**7.93**	**25.1 (59.9%)**	**20.4 (73.2%)**	**0.92**	**2.89**

* **Time (h)**: campaign duration, up to 24 h; AHFUZZ may terminate early after 30 min of coverage stagnation (following [20]'s setting).
† **# Invocations (New Cov.%)**: number of invocations of each tester, with the percentage of invocations that produced *new* coverage.
‡ **Cost ($/h)**: hourly cost breakdown for the orchestration agent (*Orch.*) and the symbolic execution agent (*S.E.*) with CONCOLLMIC.
All values are averaged across repetitions.

drops below the round's own running average (i.e., a self-adaptive threshold). All other settings follow those of STATIC.

(3) **RANDOM** shares AHFUZZ's higher-level orchestrator design and all tester operations, but schedules them randomly [51] rather than intelligently. At each round, it randomly draws seeds from the union of both testers' outputs and chooses one of the following operations: (1) start a fresh AFL++ session, (2) resume the previous AFL++ session, or (3) invoke CONCOLLMIC. The fuzzing duration *per invocation* is chosen uniformly at random in the same 5–$50\,\mathrm{min}$ range.

(4) **RANDTESTER** retains AHFUZZ's orchestrator agent *only* for seed selection and randomizes tester choice. At each round, the agent identifies the most promising seed for future testing. This seed is then forwarded to a randomly selected tester. All the other settings follow those of RANDOM. This *decouples* the effect of seed selection from tester scheduling.

We note that all the aforementioned variants share the exact same AFL++ and CONCOLLMIC configurations as AHFUZZ. STATIC and VELOCITY represent a fuzzer-driven paradigm, whereas RANDOM and RANDTESTER precisely isolate the effect of the orchestration layer's decisions on seed selection and/or tester dispatch. Besides, we include **CONCOLLMIC** as the standalone concolic executor baseline to contextualize performance gains from hybrid approaches. We conduct the comparison on *all* targets listed in Table II, and each variant is run for the same duration as AHFUZZ for fair comparison.

Results. Overall, AHFUZZ demonstrates consistent and substantial performance gains in both coverage and bug discovery. We present a short summary of key findings here and the full results in Appendix B. On average, AHFUZZ achieves 34.9%, 24.1%, 48.5%, 27.1%, and 28.4% higher branch coverage than STATIC, VELOCITY, RANDOM, RANDTESTER, and standalone CONCOLLMIC, respectively. These five variants discovered 4, 4, 4, 4, and 7 bugs, respectively, out of AHFUZZ's 13 total findings in Table III, and *none* of them discovered additional bugs. These results represent qualitative differences in the ability to penetrate deep program logic and discover latent vulnerabilities. Beyond efficacy gains, AHFUZZ also incurs 55.2% lower LLM API costs than standalone CONCOLLMIC. This is because neighborhood exploration is often dispatched to fuzzing rather than expensive per-seed symbolic reasoning.

E. Understanding the Orchestration Process

To gain deeper insight into agentic orchestration, we present key statistics of AHFUZZ's testing campaigns in Table IV, averaged across all repetitions for each subject. For a more fine-grained view, Figure 5 visualizes the invocation timeline, showing the best-performing run (in terms of coverage) for each subject. We highlight AHFUZZ's adaptiveness and efficiency along four dimensions.

(1) Adaptive Tester Selection. First, we observe AHFUZZ's adaptive strategy in determining ***which*** tester to invoke. In Table IV, the tester invocation proportion $\frac{\text{AFL++}}{\text{CONCOLLMIC}}$ varies substantially across subjects: `woff2` is fuzzing-dominant $\left(\frac{12.3}{6.0} \approx 2.1\right)$, whereas `proj4` predominantly involves symbolic execution $\left(\frac{8.0}{45.7} \approx 0.17\right)$. Such variability across subjects is a result of how the underlying testers perform under a hybrid fuzzing

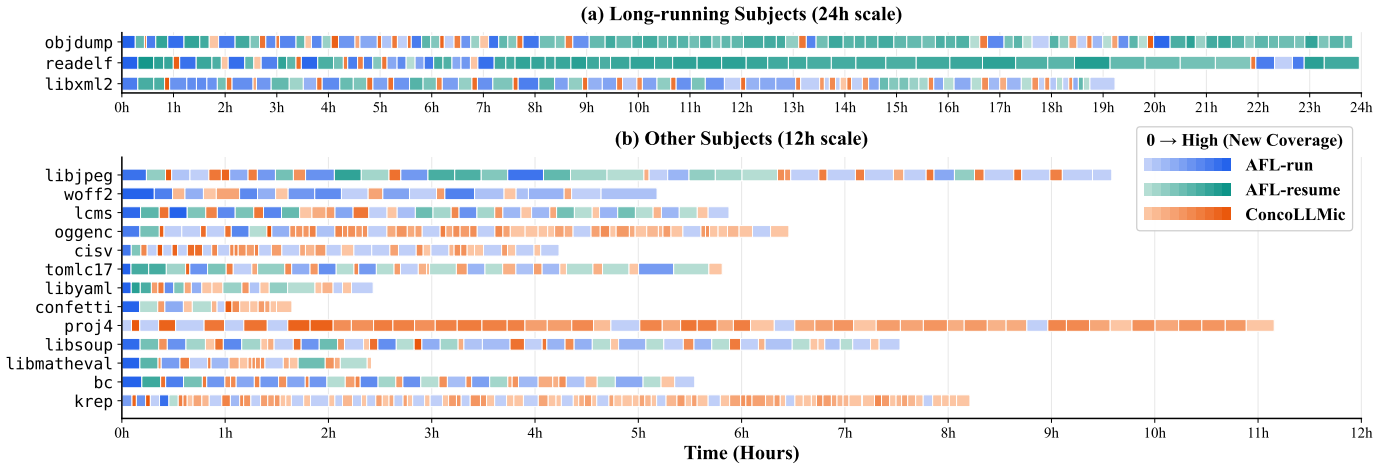


Fig. 5. Timeline of AHFUZZ's best-performing campaign (by final coverage) for each subject, presented in two panels by campaign duration. Each block represents a single tester invocation: *width* encodes its duration, *hue* encodes the tester, and *color saturation* encodes how much new coverage is achieved. `AFL-run`: a fresh AFL++ session; `AFL-resume`: the resumption of a prior one. The distinct invocation patterns across subjects exemplify how AHFUZZ adaptively decides *which* tester to invoke, *how* to configure each invocation, and *when* to switch, informed by the testing trajectory and tester performance.

setup: fuzzing is highly effective on `woff2` with 83.9% of invocations producing new coverage; in contrast, `proj4` poses challenges for AFL++ to reach new code, with only 5.6% of fuzzing invocations proving fruitful. This is further reflected by the ablation study in §V-D: the ablative variants that blindly favor fuzzing (STATIC and VELOCITY) or randomly allocate testers (RANDOM and RANDTESTER) cannot perform well *consistently* across different subjects (as shown in Table VI). Instead of following a mechanical alternation pattern (e.g., "fuzz → symbolic → fuzz"), AHFUZZ *adaptively* allocates testers based on each subject's characteristics.

(2) Effective Tester Configuration. Second, using Figure 5, we study ***how*** AHFUZZ configures each invocation. While fresh AFL++ sessions (`AFL-run`) are typically allocated short time slots, the orchestrator extends them with progressively longer timeouts (`AFL-resume`) if they prove effective, as observed in `objdump` and `readelf`. For both subjects, AHFUZZ strategically configures AFL++ sessions with well-selected seed corpora around hour 9, enabling *sustained* coverage growth for the subsequent *15 hours* (via `AFL-resume`) without frequent restarts or tool switches. These stand in contrast to `woff2`, where AHFUZZ consistently chooses *new* initial seeds to launch *new* AFL++ sessions to explore along diverse directions. On the other hand, CONCOLLMIC is carefully deployed to unravel new code regions for fuzzing, as evidenced by *spikes* of coverage gains from fuzzing (indicated by more saturated colors) that immediately follow symbolic execution invocations. This suggests that the orchestrator configures individual testers effectively to exploit their respective strengths and harvest their synergy, contrasting with the ablative variants' rigid seed retrieval (e.g., FIFO seed queue) that may make testers' efforts overlap (see Appendix B).

(3) Context-Aware Step Decision. Third, the orchestrator leverages feedback from prior invocations to continuously determine ***when*** to switch testers or change configurations. When coverage growth slows, the orchestrator quickly "cuts its losses" by switching seed corpora for fuzzing (via `AFL-run`) or invoking symbolic execution. Moreover, when historical attempts reveal a tester's ineffectiveness on a particular subject, such as on `proj4` where fuzzing struggles to yield new branch coverage (see Table IV), the orchestrator decisively redirects the campaign toward those unexplored regions with *consecutive* CONCOLLMIC invocations. This campaign-informed approach ensures that expensive symbolic reasoning is deployed precisely when it is most valuable. Further examination of the logs reveals that only 8.7% of the formulated goals were *unproductive*, for example, targeting nearly saturated code regions. We also observe that the orchestrator pivots quickly upon finding the previous goal unproductive.

(4) Cost Efficiency. The last two columns of Table IV report the cost breakdown of AHFUZZ's testing campaigns. Most of the cost comes from the symbolic execution (S.E.) component, with the orchestrator (Orch.) accounting for only $\frac{0.92}{0.92+2.89} \approx 24\%$ of the total cost. On average, the orchestration agent costs about $\$0.92/\text{h} \times 7.93\,\text{h} \approx$ **$7.3 per campaign**. This translates to roughly $\frac{7.3}{25.1+20.4} \approx \$0.16$ per orchestration decision, which we deem reasonable given the substantial coverage gains over ad-hoc coordination strategies (§V-D). We present end-to-end monetary costs of AHFUZZ in Appendix A.

VI. DISCUSSION

Cost, Context Window, and Unsoundness of LLM Calls. Since our approach relies on LLMs for decision-making, it inherits certain limitations of LLMs. First, AHFUZZ incurs higher costs due to LLM invocations. Our results show that these expenses are dominated by symbolic reasoning rather than orchestration (§V-E). Notably, by offloading neighborhood exploration to AFL++, AHFUZZ on average halves the cost of standalone CONCOLLMIC while simultaneously achieving higher coverage (§V-D). In Appendix C, we demonstrate significant cost reduction by employing a cheaper, open-

weight model (deepseek-v4-pro [52]), without sacrificing efficacy. Second, our approach is bounded by the context window size of current LLMs, which could potentially be exhausted in prolonged campaigns. In practice, even the longest-running campaign, `readelf`, accumulated at most 269K tokens over its full 24 h duration and remained well within the 1M token limit of claude-sonnet-4.5 [43]. Finally, our agentic orchestrator is inherently unsound, as LLMs may produce factually incorrect outputs. Nevertheless, this unsoundness does not compromise the soundness of generated test cases, since input synthesis and execution are delegated to individual testers.

Data Leakage. Our agentic orchestrator and the underlying concolic engine both leverage LLMs trained on many public code repositories. This raises concerns that evaluation benchmarks may overlap with training data, potentially inflating our results. We address this concern from three perspectives. First, we include four projects created after claude-sonnet-4.5's knowledge cutoff [48], and observe no qualitative performance degradation on these subjects (§V-B). Second, AHFUZZ discovers 13 zero-day vulnerabilities (§V-C), which could not have appeared in any training corpus. This demonstrates that AHFUZZ's effectiveness stems from intelligent tester orchestration rather than memorization. Third, in our ablation study (§V-D), all variants employ the same LLM backend. Familiarity with older projects or common vulnerability patterns, if any, is thus shared across all variants rather than benefiting AHFUZZ alone. Since these variants *only* differ in their coordination strategies, the substantial performance gaps observed can be directly attributed to the orchestration layer.

Reliability of Results. LLMs' decision-making is opaque and non-deterministic in nature, raising reliability concerns about the reported results. To combat opacity, we replicate the main experiment with an open-weight model (Appendix C). We further quantify the stability of AHFUZZ's orchestration decisions across repetitions, at three levels of granularity:

- **L1** *End-to-end coverage*: final branch coverage (Figure 4) has a low Coefficient of Variation (CV) of 4.2% on average.
- **L2** *Decision pattern*: the tester-dispatch proportion (fuzzing vs. S.E., in Table IV) has an average CV of 13.8%.
- **L3** *Single dispatch decision*: for the three subjects with the highest CV at L2, namely `libmatheval`, `confetti`, and `oggenc`, we replay each decision three times using its *exact* context. Results show that, out of all the decision points, goal semantics disagree on only 0.8% (i.e., 99.2% consistency), and dispatch decisions (including both tester selection and arguments) disagree on only 4.3%.

Collectively, these numbers suggest that the individual orchestration decisions remain stable (L3); therefore, run-to-run variations (L2) are likely attributable to the underlying testers' randomness. Still, these divergent trajectories often converge in terms of the final coverage (L1).

VII. Related Work

Dynamic Test Generation. Our work directly builds on the established practices of using dynamic test generation to automatically detect software vulnerabilities, most notably greybox fuzzing [2], [21], [42], [53] and symbolic execution [6], [8], [10], [26], [31], [32], [54]–[56]. However, either class of technique alone often remains insufficient to comprehensively validate complex software, necessitating hybrid approaches to bring out their complementary power.

Hybrid Approaches in Test Generation. There have been multiple avenues to combine heterogeneous test generation techniques. Earlier attempts, such as [51], randomly alternate between black-box and white-box testing until convergence. One line of work employs one-time symbolic reasoning to provide high-quality initial seeds prior to the fuzzing campaign [57]–[60]. Another promising direction is to embed lighter-weight symbolic analyses *into* fuzzing to reach deeper code regions. For example, Matryoshka [24] identifies nested branches and collapses them for fuzzing; Redqueen [47] bypasses magic bytes and complex checksum tests by tracking how inputs propagate to deep program states. The specific direction studied in this work is hybrid fuzzing, which alternates between fuzzing and symbolic execution *at runtime* and typically follows the fuzzer-centric paradigm [5]. Subsequent research advances this paradigm by enhancing the symbolic engine [6]–[8], [12]–[14], [38], [39], improving coordination through seed selection [11], [15], [16], [40], and applying it to diverse setups [61]–[63]. In this work, we argue that fuzzer-centricity leads to reactive coordination, and propose an agentic orchestration process on two equal testers.

LLM-Assisted Test Generation. Recent advances in LLMs have opened new possibilities for software testing. In fuzzing, LLMs have been leveraged to synthesize test harnesses [64], [65], produce structured inputs [59], [66], [67], overcome coverage plateaus [59], and more [68]. Recent work has also explored using LLMs to replace explicit symbolic modeling and SMT solvers in standalone symbolic/concolic execution engines [20], [69], [70] and the symbolic component in hybrid fuzzing [38], [39]. Notably, CONCOLLMIC [20] shows the promise of employing an LLM agent to summarize and solve path constraints with a flexible constraint representation. We integrate CONCOLLMIC as the symbolic reasoning engine in AHFUZZ due to its efficacy and generality [20], but focus our core contribution on harnessing agentic reasoning for high-level shepherding instead of precise test case construction.

Collaborative Testing. Collaborative testing is a broader testing scheme that combines multiple testers to achieve synergistic exploration, with hybrid fuzzing being one important instantiation of this paradigm. Another prominent example is ensemble fuzzing [71], which launches a set of diverse fuzzers in parallel to maximize collective coverage. Here, coordination is achieved by selecting the active fuzzer either statically [72] or dynamically [73]. More nuanced resource allocation strategies, such as autofz [22], decide at runtime how much CPU each fuzzer gets by monitoring its coverage trend. AHFUZZ instead targets the more fine-grained problem of *task allocation*, deciding *what* task to dispatch to *which* tester.

VIII. Perspectives

Software vulnerability discovery usually benefits from both (i) a high-throughput brute-force search in the form of fuzzing, and (ii) a rigorous path reasoning in the form of symbolic execution. Hybrid fuzzing is thus an important topic that combines the strengths of both to achieve effective bug detection.

In this work, we propose agentic orchestration to *explicitly* coordinate between symbolic execution and fuzzing, rather than have the fuzzer do so *reactively* as in traditional hybrid fuzzers [5]–[8]. Our work also stands in contrast to approaches like ConcoLLMic [20] that use LLM agents directly for end-to-end test generation, which, as shown in §V-D, can discover notably fewer bugs despite twice the budget. This suggests that *reaching* a code region via agentic reasoning and *thoroughly exercising* it using unexpected extreme values still pose distinct challenges. Fuzzing, through performant input mutations, excels at such thorough exploration and uncovers subtle bugs that elude agentic reasoning alone. The role of LLM agents is thus likely *not* in exhaustive search, but in helping to reach specific branches and smartly orchestrating security tools like fuzzers.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, under its Artificial Intelligence (AI)-for-Science (AI4S) Challenge Grant (Award No. NRFAI4SCH-2025-0003, "AI for Program Reasoning"); by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>); and by the National Research Foundation, Singapore, under its Singapore National Research Foundation (NRF) Postdoctoral Award (NRF-PA2025-NUS). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

Ethical Considerations

There may be a concern that malicious parties could also use AHFuzz to exploit software systems. However, developers can use AHFuzz to remove any bugs during software development, *before* potential attackers exploit vulnerabilities.

All security issues identified by AHFuzz have been responsibly disclosed to the developers of the respective projects. For each bug, we provided a detailed report and a Proof of Concept (PoC) to facilitate reproduction. Throughout this process, we have followed the responsible disclosure process and have received 2 CVE identifiers.

References

[1] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, 2017.

[2] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, 2021.

[3] K. Serebryany, "OSS-Fuzz: Google's continuous fuzzing for open-source software." Vancouver, BC: USENIX Association, Aug 2017.

[4] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 713–724.

[5] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*, Aug. 2018.

[7] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 515–530.

[8] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SymSan: Time and space efficient concolic execution via dynamic data-flow analysis," in *31st USENIX Security Symposium (USENIX Security 2022)*, 2022, pp. 2531–2548.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the Association for Computing Machinery (CACM)*, vol. 56, no. 2, pp. 82–90, 2013.

[10] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 13th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2005, pp. 263—272.

[11] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[12] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020, pp. 1613–1627.

[13] S. Poeplau and A. Francillon, "SymQEMU: Compilation-based symbolic execution for binaries," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2021.

[14] Z. Qi, J. Hu, Z. Xiao, and H. Yin, "SymFit: Making the common (concrete) case fast for Binary-Code concolic execution," in *33rd USENIX Security Symposium (USENIX Security 2024)*, 2024, pp. 415–432.

[15] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and improving hybrid fuzzing," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[16] Y. Chen, M. Ahmadi, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 77–92.

[17] A. Roychoudhury, C. Pasareanu, M. Pradel, and B. Ray, "Agentic AI software engineers: Programming with trust," *Communications of the Association for Computing Machinery (CACM)*, 2026.

[18] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1592–1604.

[19] R. Fang, R. Bindu, A. Gupta, and D. Kang, "LLM agents can autonomously exploit one-day vulnerabilities," *arXiv preprint arXiv:2404.08144*, 2024.

[20] Z. Luo, H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury, "Agentic concolic execution," in *Proceedings of the 47th IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 37–55.

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 2020)*, 2020.

[22] Y.-F. Fu, J. Lee, and T. Kim, "autofz: Automated fuzzer composition at runtime," in *32nd USENIX Security Symposium (USENIX Security 2023)*, 2023, pp. 1901–1918.

[23] Google, "American fuzzy lop (AFL)," 2024. [Online]. Available: https://github.com/google/AFL

[24] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 499–513.

[25] J. C. King, "Symbolic execution and program testing," *Communications of the Association for Computing Machinery (CACM)*, vol. 19, no. 7, pp. 385–394, 1976.

[26] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, Dec. 2008.

[27] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 6, pp. 867–870, 2020.

[28] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[29] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2011, pp. 171–177.

[30] O. S. Dustmann, K. Wehrle, and C. Cadar, "PARTI: A multi-interval theory solver for symbolic execution," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 9 2018, pp. 430–440.

[31] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *International Conference on Computer Aided Verification (CAV)*, 7 2013, pp. 53–68.

[32] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the Conference on Programing Language Design and Implementation (PLDI)*, Jun. 2005.

[33] K. Sen, "Concolic testing," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007, pp. 571–572.

[34] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, 2012.

[35] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 521–532.

[36] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceeding of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[37] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 736–747.

[38] Y. Yang, S. Yao, J. Chen, and W. Lee, "Hybrid language processor fuzzing via LLM-Based constraint solving," in *34th USENIX Security Symposium (USENIX Security 2025)*, 2025, pp. 6299–6318.

[39] R. Meng, G. J. Duck, and A. Roychoudhury, "Large Language Model assisted hybrid fuzzing," *IEEE Transactions on Software Engineering*, 2026.

[40] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards bug-driven hybrid testing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1580–1596.

[41] GNU, "Gcov Test Coverage," 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[42] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.

[43] Anthropic. (2026) Claude sonnet 4.5. [Online]. Available: https://platform.claude.com/docs/en/about-claude/models/overview

[44] AFLplusplus, "afl-cov: Produce code coverage results with gcov from afl-fuzz test cases," 2024. [Online]. Available: https://github.com/AFLplusplus/afl-cov

[45] ——. AFL++ documentation: LLVM instrumentation and dict2file pass. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.llvm.md

[46] ——. AFL++ documentation: CmpLog instrumentation and Redqueen. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md

[47] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019.

[48] Anthropic. (2026) Claude support: How up-to-date is claude's training data? [Online]. Available: https://support.anthropic.com/en/articles/8114494-how-up-to-date-is-claude-s-training-data

[49] LLVM, "Undefined Behavior Sanitizer," 2026. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[50] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," *2012 USENIX Annual Technical Conference*, 2012.

[51] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 416–426.

[52] DeepSeek-AI, "Deepseek-v4: Towards highly efficient million-token context intelligence," 2026. [Online]. Available: https://arxiv.org/abs/2606.19348

[53] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceeding of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[54] T. Kapus, F. Busse, and C. Cadar, "Pending constraints in symbolic execution for better exploration and seeding," in *Proceedings of the 35th IEEE International Conference on Automated Software Engineering (ASE)*, Sep. 2020.

[55] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*, Aug. 2020.

[56] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 18–35.

[57] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE-Companion)*, 2018, pp. 61–64.

[58] S. Saha, L. Sarker, M. Shafiuzzaman, C. Shou, A. Li, G. Sankaran, and T. Bultan, "Rare path guided fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 1295–1306.

[59] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model Guided Protocol Fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[60] Z. Luo, Q. Du, Y. Wang, A. Roychoudhury, and Y. Jiang, "Enhancing protocol fuzzing via diverse seed corpus generation," *IEEE Transactions on Software Engineering*, 2025.

[61] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[62] H. Liang, L. Jiang, L. Ai, and J. Wei, "Sequence directed hybrid fuzzing," in *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 127–137.

[63] L. Situ, C. Zhang, L. Guan, Z. Zuo, L. Wang, X. Li, P. Liu, and J. Shi, "Physical devices-agnostic hybrid fuzzing of iot firmware," *IEEE Internet of Things Journal*, vol. 10, no. 23, pp. 20 718–20 734, 2023.

[64] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "How effective are they? Exploring Large Language Model based fuzz driver generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1223–1235.

[65] H. Xu, W. Ma, T. Zhou, Y. Zhao, K. Chen, Q. Hu, Y. Liu, and H. Wang, "CKGFuzzer: LLM-based fuzz driver generation enhanced by code knowledge graph," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2025, pp. 243–254.

[66] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal fuzzing with Large Language Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[67] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by Large Language Models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.

[68] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li *et al.*, "When fuzzing meets LLMs: Challenges and opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE-Companion)*, 2024, pp. 492–496.

[69] H. Tu, S. Lee, Y. Li, P. Chen, L. Jiang, and M. Böhme, "Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2026.
[70] Y. Li, R. Meng, and G. J. Duck, "Large Language Model Powered Symbolic Execution," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, 2025.
[71] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 2019)*, 2019, pp. 1967–1983.
[72] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 360–372.
[73] V.-T. Pham, M.-D. Nguyen, Q.-T. Ta, T. Murray, and B. I. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1337–1341.

APPENDIX

A. Detailed Evaluation Results

In this section, we supplement Figure 4 and Table IV with Table V, which presents a detailed per-subject breakdown of the final GCov branches covered, relative improvement, and end-to-end costs of AHFUZZ.

In particular, the aggregated coverage improvement is computed as follows. Let $\overline{Cov.}_{\mathsf{tool}}(s) = \frac{1}{R}\sum_{j=1}^{R} Cov._{\mathsf{tool}}^{j}(s)$ denote the final GCov branch coverage of tool on subject s averaged over $R = 10$ repetitions, as reported in the "Branch" columns of Table V. For each $s \in \mathcal{S}$ from the set of all subjects, we compute the coverage improvement of AHFUZZ over tool on s, reported in the "Cov.Δ%" columns, as:

$$Cov.\Delta\%(s) = \frac{\overline{Cov.}_{\text{AHFUZZ}}(s)}{\overline{Cov.}_{\mathsf{tool}}(s)} - 1$$

The overall improvement of AHFUZZ is then computed as the arithmetic mean over all per-subject $Cov.\Delta\%(s)$ values, as reported in the "Average" row:

$$Cov.\Delta\%_{avg} = \frac{1}{|\mathcal{S}'|}\sum_{s\in\mathcal{S}'} Cov.\Delta\%(s)$$

where $\mathcal{S}' \subseteq \mathcal{S}$ excludes the subjects where tool fails to run (marked "–" in Table V).

To complement the ratio-based calculation above, we report aggregated improvements over *raw* covered branches:

$$Cov.\Delta\%_{raw} = \frac{\sum_{s\in\mathcal{S}'}\overline{Cov.}_{\text{AHFUZZ}}(s)}{\sum_{s\in\mathcal{S}'}\overline{Cov.}_{\mathsf{tool}}(s)} - 1$$

Under this aggregation, AHFUZZ achieves +114.6%, +93.5%, +124.1%, +97.8%, +92.1%, and +58.0% higher branch coverage than SymSan++, MEUZZ, CoFuzz, AFL++, Advanced AFL++, and autofz, respectively, outperforming *all* baselines.

B. Detailed Results of Ablation Study

In this section, we present detailed results of the ablation study referenced in §V-D. The per-subject breakdown is presented in Table VI, averaged across repetitions.

Overall, the full-fledged AHFUZZ significantly outperforms the ablative baselines. Among the four AHFUZZ variants, VELOCITY leaves the smallest coverage gap to AHFUZZ at +24.1%. By triggering a symbolic dispatch only when the coverage growth decelerates, VELOCITY improves upon STATIC's hardcoded trigger by a large margin. STATIC and VELOCITY both dispatch CONCOLLMIC at fuzzer-driven triggers and draw seeds from a FIFO queue. As a result, symbolic execution often works on sufficiently fuzzed seeds or non-critical paths, duplicating coverage already attainable by the fuzzer. On the other hand, RANDOM's random dispatch decisions are ineffective, performing the worst across all ablative variants in comparison. RANDTESTER retains the same randomized tester selection as RANDOM while allowing the orchestrator agent to search for a promising seed, yielding a noticeable coverage gain over RANDOM (relative difference of +27.1% vs. +48.5%). By construction, the gap between RANDTESTER and AHFUZZ precisely isolates the contribution of tester scheduling, whereas the gap between RANDTESTER and RANDOM reflects the effect of agentic seed selection.

The bug-finding results reinforce the conclusion, with each ablative variant only reproducing a small subset of AHFUZZ's findings. We note that *none* of the four ablative variants, nor standalone CONCOLLMIC, discovers the studied bug #5 in `libsoup` (§V-C), which requires lock-step coordination between the fuzzer and the symbolic executor. Among these baselines, CONCOLLMIC alone recovers the most, 7 of 13, vulnerabilities. Manual inspection reveals that most of the missed bugs typically surface only under counter-intuitive, extreme inputs that lie beyond what end-to-end agentic reasoning would naturally enumerate—a regime in which fuzzing's high-throughput mutations consistently excel once the agent has steered execution into the relevant region. In addition to efficacy gains, AHFUZZ also incurs considerably less (-55.2%) LLM API cost than CONCOLLMIC, since neighborhood exploration is often delegated to fuzzing.

C. Model Sensitivity and Reproducibility

To assess the sensitivity of our approach to the underlying reasoning engine, we repeat the main experiment using a cheaper, open-weight model, deepseek-v4-pro [52], as the LLM backend for *both* the orchestrator and the concolic engine. For conciseness, we denote the deepseek-powered variant as AHFUZZ–🐬, and the original version with claude-sonnet-4.5 [43] as AHFUZZ–✳.

The rightmost columns of Table VI directly compare the same AHFUZZ framework instantiated with two LLM backends. On average, AHFUZZ–🐬 is closely comparable to AHFUZZ–✳, with a mere 3.3% difference in the final branch coverage. AHFUZZ–🐬 also surfaces 9 distinct bugs across the campaigns in total. Six of these coincide with AHFUZZ–✳'s 13 findings; the remaining three are previously *unknown* and were never triggered by AHFUZZ–✳. These bugs are listed as #14–#16 in Table VII. Overall, there is no observable performance dominance in either coverage or bug-finding

TABLE V
DETAILED EVALUATION DATA AND STATISTICS USED IN FIGURE 4 AND TABLE IV.

Subject	AHFUZZ			SymSan++		MEUZZ		CoFuzz		AFL++		Adv. AFL++		autofz	
	Branch*	Time(h)†	Cost($)‡	Branch	Cov.Δ%§	Branch	Cov.Δ%	Branch	Cov.Δ%	Branch	Cov.Δ%	Branch	Cov.Δ%	Branch	Cov.Δ%
objdump	6,520	20.21	45.47	4,688	+39.1%	–	–	4,682	+39.3%	4,842	+34.7%	5,093	+28.0%	5,334	+22.2%
readelf	7,948	23.97	29.24	6,522	+21.9%	–	–	7,227	+10.0%	6,098	+30.3%	6,196	+28.3%	7,105	+11.9%
libjpeg	2,614	7.56	17.99	1,638	+59.6%	1,920	+36.2%	–	–	1,807	+44.7%	1,864	+40.2%	2,061	+26.8%
woff2	3,163	2.99	10.94	1,623	+94.9%	–	–	–	–	1,689	+87.3%	1,427	+121.6%	1,503	+110.5%
lcms	1,620	4.88	19.62	787	+106.0%	–	–	399	+305.7%	760	+113.1%	616	+163.2%	–	–
oggenc	673	4.80	20.02	173	+288.3%	–	–	170	+296.8%	172	+292.2%	220	+206.5%	221	+204.5%
cisv	336	2.88	11.49	114	+194.7%	74	+354.1%	117	+187.2%	117	+187.2%	117	+187.2%	117	+187.2%
tomlc17	778	4.96	11.85	723	+7.7%	753	+3.4%	662	+17.6%	742	+4.8%	726	+7.2%	742	+4.8%
libxml2	14,418	16.03	67.65	3,448	+318.2%	1,496	+863.6%	3,475	+314.9%	3,594	+301.2%	4,906	+193.9%	4,903	+194.1%
libyaml	2,762	4.50	11.65	2,608	+5.9%	2,737	+0.9%	2,555	+8.1%	2,657	+4.0%	2,714	+1.8%	2,827	-2.3%
confetti	343	1.78	5.34	309	+11.1%	324	+5.9%	311	+10.3%	308	+11.4%	312	+9.9%	324	+5.9%
proj4	14,498	11.17	140.07	3,007	+382.2%	11,042	+31.3%	2,337	+520.3%	4,952	+192.8%	4,373	+231.5%	–	–
libsoup	1,713	5.10	15.86	–	–	–	–	–	–	1,107	+54.7%	1,096	+56.2%	–	–
libmatheval	354	2.59	7.90	287	+23.4%	320	+10.6%	282	+25.4%	298	+18.6%	358	-1.2%	351	+0.9%
bc	1,413	6.41	14.17	1,030	+37.2%	970	+45.7%	1,013	+39.4%	1,002	+41.0%	1,018	+38.8%	1,000	+41.2%
krep	891	6.99	42.78	218	+309.6%	215	+314.6%	220	+305.2%	217	+310.9%	217	+310.8%	232	+284.2%
Average		**7.93**	**29.50**		**+126.7%**		**+166.6%**		**+160.0%**		**+108.1%**		**+101.5%**		**+84.0%**

"–" indicates tool failures on specific targets. Averages are computed over the subjects where the corresponding tool ran successfully.
* **Branch**: final number of GCov branches covered, aggregated across repetitions.
† **Time(h)**: average end-to-end campaign duration of AHFUZZ, same as the "Time" column from Table IV.
‡ **Cost($)**: average end-to-end LLM API cost, equal to Time × (Orch. + S.E.) from Table IV for each subject.
§ **Cov.Δ%**: AHFUZZ's relative gain over the corresponding baseline in final GCov branch coverage ("Branch" column).

TABLE VI
COMPARISON OF AHFUZZ AGAINST FOUR ABLATIVE VARIANTS (STATIC, VELOCITY, RANDOM, RANDTESTER) AND THE STANDALONE CONCOLLMIC, ALL POWERED BY CLAUDE-SONNET-4.5. ADDITIONAL COMPARISON WITH FULL AHFUZZ USING AN OPEN-WEIGHT MODEL DEEPSEEK-V4-PRO.

Subject	vs. STATIC		vs. VELOCITY		vs. RANDOM		vs. RANDTESTER		vs. CONCOLLMIC			vs. AHFUZZ–		
	Cov.Δ%*	Bugs†	Cov.Δ%	Bugs	Cov.Δ%	Bugs	Cov.Δ%	Bugs	Cov.Δ%	Bugs	Cost.Δ%‡	Cov.Δ%	Bugs	Cost.Δ%
objdump	+36.6%	0/0	+10.8%	0/0	+49.8%	0/0	+1.5%	0/0	+1.9%	0/0	-63.3%	-28.7%	0/0	+1034.1%
readelf	+31.8%	0/0	+17.6%	0/0	+69.9%	0/0	+21.5%	0/0	+93.6%	0/0	-72.7%	-6.7%	0/0	+647.1%
libjpeg	+13.9%	0/0	+9.3%	0/0	+26.7%	0/0	+15.5%	0/0	+4.5%	0/0	-80.3%	-10.9%	0/0	+819.7%
woff2	+115.3%	0/0	+115.2%	0/0	+142.4%	0/0	+136.4%	0/0	+161.8%	0/0	-66.2%	+8.7%	0/0	+901.6%
lcms	+20.5%	0/1	+12.5%	0/1	+49.9%	0/1	+12.8%	0/1	+10.9%	0/1	-55.2%	+3.2%	0/1	+1710.6%
oggenc	+49.6%	1/1	+30.4%	1/1	+100.9%	1/1	+38.2%	1/1	+10.0%	1/1	-44.4%	-5.9%	1/1	+1215.7%
cisv	+28.2%	1/2	+32.8%	1/2	+56.3%	1/2	+18.7%	0/2	+13.5%	2/2	-44.6%	-3.1%	1/2	+1016.3%
tomlc17	-1.1%	0/0	-1.0%	0/0	+6.3%	0/0	+0.9%	0/0	+12.3%	0/0	-72.7%	-0.4%	0/0	+761.4%
libxml2	+124.3%	0/0	+85.0%	0/0	+145.5%	0/0	+75.1%	0/0	+34.4%	0/0	-54.5%	-20.9%	0/0	+990.0%
libyaml	+5.2%	0/0	+2.1%	0/0	+4.7%	0/0	+15.6%	0/0	+51.3%	0/0	-70.8%	+6.2%	0/0	+891.3%
confetti	+5.2%	0/0	+4.6%	0/0	+10.3%	0/0	+10.3%	0/0	+8.9%	0/0	-63.9%	+5.2%	0/0	+834.8%
proj4	+28.2%	0/0	+14.2%	0/0	+11.7%	0/0	+11.0%	0/0	+17.8%	0/0	+33.0%	-2.9%	0/0	+2761.3%
libsoup	+8.2%	0/2	+9.7%	0/2	+23.4%	0/2	+10.0%	0/2	+14.1%	0/2	-71.8%	-8.6%	0/2	+941.3%
libmatheval	+8.9%	1/1	+3.5%	0/1	+22.9%	1/1	+5.0%	1/1	+3.2%	1/1	-67.1%	-3.3%	0/1	+1048.2%
bc	+24.6%	1/3	+5.6%	2/3	+22.4%	1/3	+16.9%	1/3	+6.2%	1/3	-68.9%	+7.0%	3/3	+1424.7%
krep	+58.3%	0/3	+33.6%	0/3	+33.2%	0/3	+43.5%	1/3	+10.4%	2/3	-20.3%	+8.9%	1/3	+2297.9%
Average (Total)	**+34.9%**	**(4/13)**	**+24.1%**	**(4/13)**	**+48.5%**	**(4/13)**	**+27.1%**	**(4/13)**	**+28.4%**	**(7/13)**	**-55.2%**	**-3.3%**	**(6/13)**	**+1206.0%**

All columns are reported *relative to* AHFUZZ: a *positive* value means AHFUZZ covers more branches or costs more; *negative* means the opposite.
* **Cov.Δ%**: relative difference in the final GCov branch coverage, aggregated across repetitions.
† **Bugs**(X/Y): X of AHFUZZ's Y bugs on that subject (Table III) were also found by the comparison tool; only AHFUZZ– found additional bugs (Table VII).
‡ **Cost.Δ%**: relative difference in the end-to-end LLM API cost. A negative value indicates a cost *saving* of AHFUZZ over the comparison tool.

capabilities, and AHFUZZ remains effective when instantiated with an open-weight backend.

The most noteworthy distinction lies in their respective costs incurred by LLM usage. Across all runs, AHFUZZ– is 13× as *expensive* as its deepseek counterpart on average. This makes AHFUZZ– a practical default that offers comparable performance at a fraction of the cost, further improving the applicability of AHFUZZ.

D. Future Work

1) Exploration of Algorithmic Equivalence: AHFUZZ relies on an LLM agent for smart dispatch, which requires (i) understanding a task's nature in order to route it to the better-suited tester, and (ii) selecting suitable starting seeds. Both can be difficult to achieve with static algorithms alone. Task nature has many facets—the constraint inter-dependencies of Figure 2, the complex path constraints of bug #5, the environment constraints of bug #11—*each* demanding its own heavyweight program analysis (e.g., taint tracking [24]) or learning-based heuristic [11], [16]. Seed selection is no easier: a shorter distance on the control- or data-flow graph may not imply easier reachability [42], since these *unweighted* edges collapse semantic difficulty into hop counts. Replicating either without an LLM would mean stitching together multiple specialized analyses, at a cost to usability compared with the testers AHFUZZ uses out of the box. The difficulty compounds for direction shepherding, which requires distilling decision signals from voluminous yet undifferentiated artifacts. As such, we leave a systematic study of these non-LLM equivalents to future work.

2) Integration with Other Testers and Languages: AHFUZZ could in principle orchestrate additional testing tools beyond AFL++ and CONCOLLMIC. Integrating a new tool requires

TABLE VII
ADDITIONAL PREVIOUSLY UNKNOWN VULNERABILITIES EXPOSED *only* BY AHFUZZ WITH THE BACKEND LLM DEEPSEEK-V4-PRO.

#	Subject	Bug Description	Status
14	`readelf`	Out-of-bounds read in `sframe_decode_free()` due to unvalidated entry size when decoding a truncated `SFrame`	Reported
15	`oggenc`	Undefined behavior in `aiff_open_impl()` due to the out-of-range float-to-int cast of an `AIFF` sample rate	Reported
16	`krep`	Infinite loop in `print_matching_items()` due to zero-length matches at end-of-line when searching with an empty regex	Fixed

(1) an interface to invoke it and distill feedback from its output, and (2) a conversion routine to synchronize its generated test cases and coverage contributions with the global repository. In this work, we instantiate the framework with two techniques whose synergy has been well established [2] to demonstrate the potential of agentic orchestration. Furthermore, the uniform tester interfaces of our framework (see §IV) make it extensible to other programming languages. This requires pairing the language-agnostic symbolic engine CONCOLLMIC with an appropriate fuzzer and coverage toolchain for the target language, e.g., `<Jazzer, JaCoCo>` for Java. We leave this as a promising direction for future work.