CtxFuzz: Context-Aware Directed Fuzzing via Runtime Fuzzer-Agent Cooperation

Jiho Kim
Georgia Institute of Technology
Atlanta, USA
jkim4050@gatech.edu

Dongkwan Kim
Georgia Institute of Technology
Atlanta, USA
dongkwan@gatech.edu

HyungSeok Han
Microsoft
Bellevue, USA
hyungseokhan@microsoft.com

Youngjoon Kim
Georgia Institute of Technology
Atlanta, USA
youngjoon.kim@gatech.edu

Soyeon Park
Independent Researcher
Bellevue, USA
spark720@gatech.edu

Sangwoo Ji
Samsung Electronics
Seoul, Republic of Korea
sangwoo_.ji@samsung.com

Joshua Wang
Georgia Institute of Technology
Atlanta, USA
jwang3453@gatech.edu

Dae R. Jeong
Seoul National University
Seoul, Republic of Korea
dae.r.jeong@snu.ac.kr

Taesoo Kim✉
Georgia Institute of Technology
Atlanta, USA
taesoo@gatech.edu

Abstract

Automated tools produce vulnerability reports at unprecedented scale, but turning a report into a confirmed proof-of-vulnerability remains challenging. Directed grey-box fuzzers are widely used for validation but lack the ability to consume the rich context these reports carry, reducing each target to a bare location. In contrast, LLM-based approaches can consume this context but are often unable to identify the true triggering mechanism or to synthesize a working proof-of-vulnerability.

We present CtxFuzz, a context-aware directed fuzzer that enables an LLM agent and a coverage-guided fuzzer to iteratively exchange and refine bug-triggering context throughout the campaign. The fuzzer contributes execution context, such as what inputs were tried and how the program behaved, grounding the agent's reasoning in concrete evidence. The agent contributes semantic context, including hypotheses about the triggering conditions, explanations of failed attempts, and targeted mutations to overcome them, guiding the fuzzer toward relevant regions of the input space. Failed attempts are not discarded but decomposed into mutation scripts that translate each identified blocker into a fuzzer-executable strategy. This iterative exchange enables CtxFuzz to trigger vulnerabilities that neither directed fuzzers nor LLM agents reach independently.

On 62 vulnerabilities across 16 open-source projects from the AIxCC Final competition (39 C, 23 Java), CtxFuzz exposes 61 of 62 targets, including 29 that no state-of-the-art directed fuzzer reaches, and exposes two zero-day vulnerabilities through a real-world vulnerability report validation pipeline.

CCS Concepts

• **Security and privacy → Software security engineering**.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834365

Keywords

directed fuzzing, context-aware fuzzing, agent-fuzzer cooperation, large language models, vulnerability validation

ACM Reference Format:
Jiho Kim, Dongkwan Kim, HyungSeok Han, Youngjoon Kim, Soyeon Park, Sangwoo Ji, Joshua Wang, Dae R. Jeong, and Taesoo Kim. 2026. CtxFuzz: Context-Aware Directed Fuzzing via Runtime Fuzzer-Agent Cooperation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834365

1 Introduction

AI tools generate vulnerability reports faster than humans can validate them, with a significant fraction being false or incomplete signals. The curl project shut down its bug bounty program after AI-generated submissions proved largely irreproducible [37], and the OpenSSF formed a working group to manage similar surges across open-source projects [33]. Each report carries rich context, including descriptions, patches, and CWE types, but what is missing is the ability to automatically consume this context and produce a concrete input that triggers the vulnerability.

Directed grey-box fuzzers (DGF) [7, 8, 10, 11, 15, 17, 22–24, 26, 27, 30, 32, 36, 48] are natural candidates for building such pipelines, as they are specifically designed to steer execution toward target locations and expose bugs. Despite their promise, however, these fuzzers often fail to reach targets or trigger the vulnerabilities because they rely on syntactic distance metrics (*e.g.*, basic block proximity) without accounting for the semantic barriers inherent in program execution. Thus, they tend to become trapped in local optima during the search process instead of successfully triggering target vulnerabilities.

LLM-Assisted Directed Fuzzing. The contextual knowledge that directed fuzzers discard is precisely what LLM agents can process, such as descriptions, source code, and patches that encode the conditions required to trigger a vulnerability. Recent work exploits this capability [9, 40, 41, 46]. Locus [47] uses an LLM to synthesize

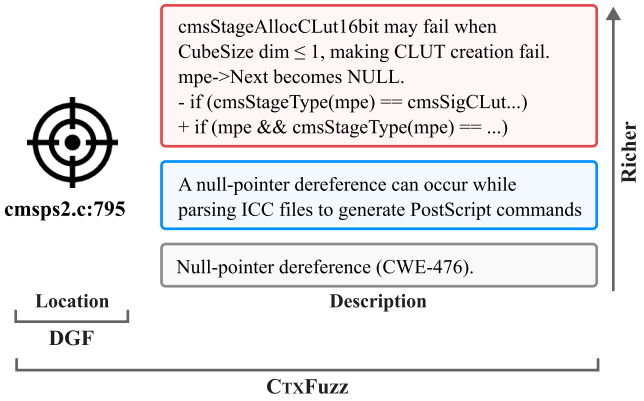

Figure 1: A vulnerability target comprises a location and a description. Analysis tools produce descriptions at varying levels of richness. Directed fuzzers use only the location, discarding descriptions that encode the triggering conditions; CtxFuzz consumes all available context.

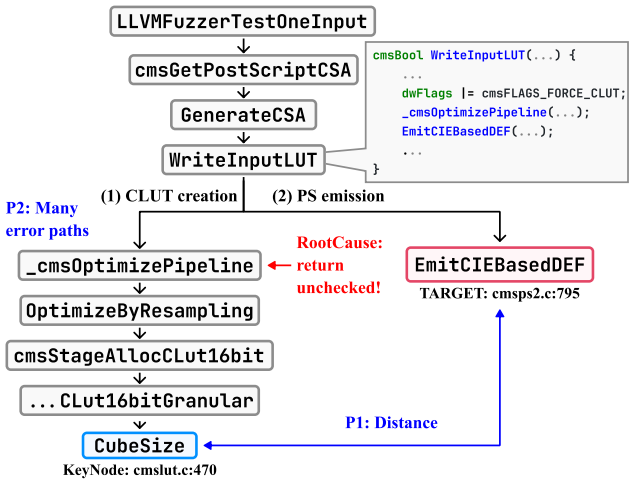

Figure 2: Call graph of a Little CMS [1] null-pointer dereference. `WriteInputLUT` forces CLUT creation (1) before emitting PostScript (2), but the return of (1) is never checked, so when CLUT allocation fails, `EmitCIEBasedDEF` dereferences `NULL`. P1: the key node (`CubeSize`) and the sink (`EmitCIEBasedDEF`) are in separate subtrees and distant, limiting the effectiveness of distance metrics. P2: `_cmsOptimizePipeline` is a complex pipeline; understanding the mechanism to make it fail is non-trivial.

intermediate predicates that are compiled into the target binary; HGFuzzer [40] generates reachable seeds and target-specific mutators before fuzzing begins. Yet in every case the LLM's guidance is fixed before fuzzing begins. When runtime evidence reveals that a hypothesis was wrong or a precondition was missed [44], adapting would require re-running the LLM analysis and, for compile-time approaches, recompiling the target binary, which none of these systems support [47].

Motivating Example. Consider the Little CMS vulnerability in Figure 1 and Figure 2. The description identifies a null-pointer dereference triggered when a specific ICC profile structure causes CLUT allocation to fail silently. Triggering this vulnerability requires reaching the sink, reaching the key condition, satisfying structured input constraints, and triggering the sanitizer crash—requirements that, in our evaluation, neither traditional directed fuzzing nor an LLM's reasoning capability satisfies in full. For traditional directed fuzzers, the key node (`CubeSize`) and the sink (`EmitCIEBasedDEF`) reside in separate call subtrees (Figure 2); distance can steer execution toward the sink, but does not by itself specify which field to mutate to satisfy the key condition, and generic mutation is unlikely to produce the structured binary profile that triggers the failure. For LLM-assisted approaches, the vulnerability is guarded by a complex optimization pipeline (`_cmsOptimizePipeline`). Understanding the true failure mechanism, that CLUT allocation must be skipped through a specific input structure, is itself non-trivial. LLM-based analysis frequently misidentifies the triggering mechanism, yielding false positives and generating inputs that fail to bypass CLUT allocation. An effective tool must iteratively refine its understanding through execution feedback, not rely on a single round of reasoning.

CtxFuzz: Context-Aware Runtime Cooperation. We present CtxFuzz, a context-aware directed fuzzer built on the runtime cooperation of a coverage-guided fuzzer and an LLM agent. Unlike prior LLM-assisted approaches whose guidance is fixed before fuzzing begins, CtxFuzz lets an LLM agent cooperate with the fuzzer via a *context interface* throughout the campaign, enabling bidirectional exchange of diverse contextual information, with no additional target-specific instrumentation or recompilation. The fuzzer provides seeds and coverage to the agent, enabling analysis based on execution evidence, and the agent returns scheduling hints and targeted mutation scripts to the fuzzer. When exploration stalls, CtxFuzz identifies semantic constraints, which we call *blockers*, that prevent progress toward the target and recursively decomposes each into targeted mutation scripts. The fuzzer applies these scripts across its seed pool at scale. The agent then observes the new coverage, identifies any remaining blockers, and repeats, driving execution closer to the target.

We evaluate CtxFuzz on 16 open-source projects from the AIxCC Final competition (62 vulnerabilities across 39 C and 23 Java programs) and additional real-world projects. CtxFuzz exposes 61 of 62 targets, including 29 that no baseline reaches, outperforming state-of-the-art directed fuzzers by nearly three times on C targets. It exposes two zero-days in real-world software by validating reports from automated code review. CtxFuzz is being integrated into the OSS-CRS platform [12] under the OpenSSF and the Linux Foundation.

This paper makes four contributions:

- CtxFuzz, a novel directed fuzzing architecture with a *context interface* that enables an LLM agent and a coverage-guided fuzzer to iteratively exchange seeds, coverage, scheduling hints, and mutation scripts at runtime, requiring no target-specific instrumentation or recompilation (§3).

- An evaluation on 62 vulnerability targets from 16 real-world projects (the AIxCC Final dataset) showing that CtxFuzz outperforms state-of-the-art directed fuzzers (§5.2).
- An ablation study showing that agent-fuzzer cooperation and rich vulnerability context are both necessary for effective directed fuzzing (§5.3).
- During initial integration testing with the OSS-CRS platform [12] under the OpenSSF and the Linux Foundation, CtxFuzz has already exposed two zero-day vulnerabilities in a real-world open-source project, one of which is critical (CVSS 9.8) [2] (§5.4).

2 Background

2.1 Directed Grey-box Fuzzing

Directed grey-box fuzzing (DGF) [7, 8, 10, 11, 15, 17, 22–24, 26, 27, 30, 32, 36, 48] steers fuzzing effort toward a chosen target location. A typical workflow has three stages: *(1) static preprocessing* recovers control-flow and call-graph structure and computes a guidance metric toward the target, most commonly shortest-path distance over the call graph (CG) and control-flow graph (CFG) [7, 8, 10, 15, 26]; *(2) seed scheduling* prioritizes inputs whose executions are deemed closer to the target; *(3) mutation* generates new inputs from scheduled seeds and repeats the loop until the target is reached or the budget (typically a timeout) is exhausted. Most systems use generic AFL-style mutations (*e.g.*, bit-flips, byte insertions) [18], though some add target-aware strategies [10, 11, 15, 23, 38].

Practical constraints. Although some tools adapt scheduling weights at runtime [39], the distance metric and instrumentation computed during preprocessing remain fixed throughout the campaign. When the exploration bottleneck shifts (*e.g.*, the fuzzer reaches a new function but stalls at a branch condition inside it due to semantic constraints), the guidance cannot adapt (*cf.* Figure 2). Also, most existing DGF tools are LLVM-centric and have been evaluated primarily on C/C++ programs, limiting portability to ecosystems such as Java. These constraints are not merely theoretical. Even at the AIxCC competition, where challenges provided vulnerability locations and descriptions in SARIF format, few teams adopted directed fuzzers [44].

2.2 LLM Agents as Fuzzing Components

LLM agents can read source code, trace data flow across functions, and generate executable artifacts such as seeds, mutators, and test harnesses [9, 31, 40, 41, 43, 46, 47]. Equipped with program-analysis tools (*e.g.*, call-graph queries, constant lookup), an agent can identify which function parses a security-relevant field, trace how input bytes reach it, and write a mutation script that targets those bytes, a capability that directed fuzzers lack. However, LLMs degrade on long contexts, particularly for generation rather than retrieval or classification [28]. In particular, an agent can reliably *diagnose* which branch blocks progress even from a large call graph, but *producing* a correct mutation script requires narrowly scoped, per-function reasoning that current models struggle to sustain over repository-scale inputs [34, 45]. This limitation extends to LLM-only approaches. Without execution feedback to validate generated inputs, agents tend to produce syntactically plausible but semantically incorrect test cases that fail to reach the intended target. Thus, directed fuzzers lack semantic understanding of the target, while LLM agents lack grounded validation, leaving neither sufficient alone.

3 Design of CtxFuzz

CtxFuzz bridges this gap with a *context interface* through which the agent and the fuzzer cooperate at runtime: the agent contributes target-aware mutation scripts, and the fuzzer returns execution feedback that grounds the agent's next reasoning step.

3.1 Overview

As shown in Figure 3, CtxFuzz comprises a *preprocessing* stage, a *directed fuzzer*, and an *LLM agent*, all connected through the context interface.

Preprocessing. Given a target project, a target location and a description, CtxFuzz builds the analysis context that all subsequent stages consume: a call graph and instrumented target binaries.

Context interface. The agent exchanges directives and observations with the directed fuzzer through a lightweight *context interface* exposed as agent tools. Three context artifacts flow between fuzzer and agent: *bug contexts* (*BC*s) encode critical lines to cover and key conditions to satisfy for vulnerability triggering, *mutation scripts* are agent-generated targeted mutators for the fuzzer, and a *seed–coverage database* is a shared artifact that captures the current progress.

Directed fuzzer. The directed fuzzer extends a standard coverage-guided fuzzer with *context-aware weighted scheduling* and a *script mutator*. The directed fuzzer polls for new BC*s* and *mutation scripts* at each iteration, incorporating the agent's context-aware directives in real time so that its scheduling and mutation strategies evolve throughout the campaign.

LLM agent. The agent pipeline has four stages. In the first stage, multiple *autonomous exploiters* run in parallel, each equipped with tools such as `query_call_graph`, `read_seed_cov`, `execute_blob`, and `report_attempt`, among others. Each exploiter navigates the program freely and submits *attempt reports* that record validated call paths, coverage depth, and failure reasons. When exploitation stalls, the subsequent stages run sequentially: the *blocker analyzer* identifies blockers, the *key-condition explorer* recursively decomposes each into *key conditions*, and the *mutator generator* produces *mutation scripts* to resolve them.

3.2 Preprocessing

Given a target project with source code, initial seeds, a vulnerability location, a vulnerability description, and an optional bug-inducing diff, CtxFuzz produces two key artifacts: a *minimally instrumented binary* and an *enriched call graph*.

Instrumentation. CtxFuzz compiles the target with sanitizer and coverage instrumentation sufficient for crash detection and line coverage tracking, without the heavyweight static analyses (*e.g.*, distance computation [8] or critical-branch identification [39]) that existing directed fuzzers require. This lightweight design supports diverse projects and preserves fuzzing speed by offloading semantic reasoning to the agent.

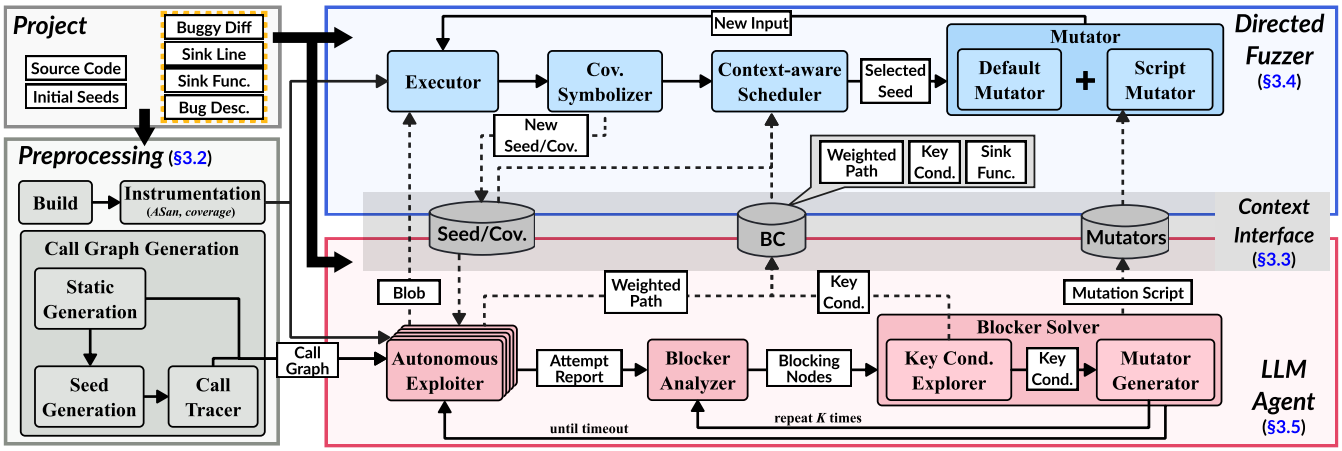

Figure 3: Overview of CtxFuzz. After preprocessing (left) builds a call graph and target binaries, the directed fuzzer (blue, top) and LLM agent (red, bottom) cooperate in a closed loop through the context interface (grey, middle).

Call graph generation. CtxFuzz builds the initial call graph by performing a breadth-first traversal from the harness entry point using CodeQL queries [19]. If the static call graph does not contain a path from the harness to the sink, CtxFuzz enriches it through seed generation and call tracing. An LLM agent first infers the expected input format and produces seeds designed to exercise diverse paths. CtxFuzz then executes these seeds under a call tracer, discovering caller–callee edges that static analysis alone cannot resolve. The resulting enriched call graph serves as the agent's primary context throughout the entire campaign.

3.3 Context Interface

The agent understands program semantics but lacks high-throughput execution; the fuzzer explores paths efficiently but lacks contextual reasoning. Sharing rich context between them through structured artifacts allows each component to compensate for the other's weakness. To this end, CtxFuzz introduces three context artifacts: *BC*s (which locations to prioritize), mutation scripts (what mutations to apply), and a seed–coverage database (what the fuzzer has observed).

Bug context (*BC*s). The agent knows which code locations matter from its semantic analysis, but directed fuzzers embed this knowledge at compile time through custom instrumentation. *BC*s let CtxFuzz communicate the same information at runtime: the agent identifies the important locations through its semantic reasoning, and the fuzzer's seed scheduler, which consumes *BC*s, gives higher priority to seeds whose coverage overlaps with the annotated locations. The fuzzer's *BC*-weighted guidance becomes progressively more specific as the campaign learns more: ❶ *sink function* registers the sink function given as the fuzzing target; ❷ *weighted path* annotates source lines along a call path from the entry to the sink, identified through the agent's reasoning; ❸ *key condition* annotates the source line of a specific branch condition that the agent determines must be satisfied to trigger the vulnerability, receiving the highest scheduling priority. All three types are source-location annotations: the scheduler consumes them through line-level coverage rather than through branch- or basic-block-distance instrumentation.

Seed–coverage database. The seed–coverage database makes the fuzzer's exploration state visible to the agent. By querying which seeds reach a given function and examining their line-level coverage, the agent can identify where the fuzzer is stuck and decide what action to take next. Seeds and their coverage data drive every agent decision, from identifying blockers to generating mutation scripts. The database is strictly read-only for agents; agent-generated inputs are fed into the fuzzer's execution cycle and reach the database only if they trigger new coverage. By delegating bug triggering entirely to the fuzzer, CtxFuzz preserves the fuzzer's corpus quality while benefiting from the agent's semantic guidance.

Mutation scripts. A mutation script is a Python function of the form `def mutate(seed: bytes) -> bytes` that encodes a target-specific, context-aware mutation targeting a key condition identified by the agent. Unlike the blind mutations of traditional fuzzers, each script is tailored to the specific vulnerability: the agent determines how to mutate the input based on its understanding of the target conditions. The fuzzer executes the script across its entire seed pool, building on valid seeds rather than generating inputs from scratch.

3.4 Directed Fuzzer

The directed fuzzer extends a coverage-guided fuzzer with four components: an *executor* that runs inputs and collects coverage and crashes, a *coverage symbolizer* that translates edge coverage into line coverage, a *context-aware scheduler* that prioritizes seeds using *BC*-annotated locations, and a *script mutator* that applies agent-generated mutation scripts alongside the fuzzer's default mutations. The agent continuously provides *BC*s and mutation scripts, so the fuzzer's scheduling and mutation strategies are dynamically guided by the agent's context-aware reasoning. Algorithm 1 presents the complete fuzzer loop.

Executor. The executor runs each mutated input against the instrumented binary. The executor collects edge coverage, crash signals.

Algorithm 1: Directed fuzzer of CtxFuzz

```
Input: initial seed corpus S_0, sink (sink_func, sink_line)
Output: crashing inputs S_×
Context input: blobs S_gen, mutators M, bug contexts BC
Context output: seed corpus S, line coverage set C_line
function SelectSeed(S, BC):
    S_BC ← {s ∈ S | ∃ (ℓ, τ) ∈ BC: ℓ ∈ cov_line(s)}
    r ← Uniform(0, 1)
    if r < 0.25 then return UniformRandom(S)
    if r < 0.50 then return UniformRandom(S_BC)
    foreach s ∈ S_BC do
        w(s) ← 1 + Σ_{(ℓ,τ)∈BC} w_τ · 𝟙[ℓ ∈ cov_line(s)]
    return s ∈ S_BC with Pr[s] = w(s) / Σ_{s'∈S_BC} w(s')
function RunAndUpdate(t, S, S_×):
    cov_edge, crash ← Execute(t)
    cov_line ← Symbolize(cov_edge)
    C_line ← C_line ∪ {(t, cov_line)}
    if crash then S_× ← S_× ∪ {t}
    else if IsInteresting(cov_line, t) then S ← S ∪ {t}
S ← S_0
C_line ← {(s, cov_line(s)) | s ∈ S_0}
BC ← {(sink_func, sink), (sink_line, path)}
M ← ∅
repeat
    BC ← BC ∪ PollBugContexts(); M ← M ∪ PollMutators()
    foreach b ∈ PollGenBlobs() do
        RunAndUpdate(b, S, S_×)
    s ← SelectSeed(S, BC)
    p ← AssignEnergy(s)
    for i ← 1 to p do
        s' ← DefaultMutate(s)
        RunAndUpdate(s', S, S_×)
    foreach m ∈ M do
        if (s, m) already executed then skip
        s'' ← m.mutate(s)
        RunAndUpdate(s'', S, S_×)
until timeout or crash
```

Both fuzzer-mutated inputs and agent-generated blobs pass through the executor; those that trigger new coverage or crashes are saved to the corpus.

Coverage symbolizer. The agent reasons about source code, not raw edge coverage, yet fuzzers track only edge coverage for efficiency. The coverage symbolizer bridges this gap by converting edge coverage into line-level coverage for every seed and crash input. These records populate the seed–coverage database, allowing the agent to inspect exactly which lines and functions each seed reaches. This translation enables fuzzer-agent cooperation; without source-level coverage, the agent cannot relate the fuzzer's progress to the source code it reasons about.

Context-aware scheduler. The core challenge of combining an LLM agent with a directed fuzzer is how to apply the agent's context-aware reasoning to the fuzzer's exploration. The context-aware scheduler is the key component that addresses this challenge. It prioritizes seeds based on *BC*s, which encode the critical lines and conditions the agent identifies as necessary to trigger the vulnerability. Formally, each *BC* is a pair $bc = (\ell, \tau)$ of a source line location ℓ and a type $\tau \in \{\texttt{sink}, \texttt{path}, \texttt{key_cond}\}$ corresponding to the three emission types above. The context-aware scheduler scores each seed s by its coverage overlap with the active *BC*:

$$w(s) \;=\; 1 \;+\; \sum_{(\ell,\tau)\in BC} w_\tau \;\cdot\; \mathbb{1}\big[\ell \in \mathrm{cov}_{\mathrm{line}}(s)\big] \tag{1}$$

where $w_{\texttt{sink}} = 8$, $w_{\texttt{key_cond}} = 4$, $w_{\texttt{path}} = 1$. Seeds whose coverage overlaps any *BC*-annotated location are placed in an interesting pool S_{BC}. However, purely score-based selection risks three failure modes: the agent may emit incorrect *BC*s, the fuzzer may need to mutate seeds that have not yet touched any key line, and the score can overfit to *BC*s with many annotated locations. To mitigate these risks, the scheduler uses a mixed selection strategy. With 25% probability, it selects uniformly from the full corpus to explore untouched paths. With another 25%, it selects uniformly from S_{BC} to give all interesting seeds a chance. The remaining 50% selects from S_{BC} weighted by $w(s)$, concentrating effort on the most promising seeds. This context-aware scheduler enables a dynamically shifting exploration focus, in contrast to traditional directed fuzzers whose focus is fixed at compile time and cannot adapt.

Mutator. Even with context-aware scheduling, the fuzzer cannot trigger complex vulnerabilities through generic mutations alone. The script mutator addresses this with semantically targeted mutations: it applies agent-generated Python scripts that run `mutate(seed)` on the selected seed based on the agent's understanding of the target conditions. The default mutator continues to apply the base fuzzer's built-in mutations (e.g., libFuzzer for C/C++, Jazzer for Java) for broad exploration. Both strategies share the same seed pool and contribute to the same corpus, so agent-guided mutations and random mutations reinforce each other.

3.5 LLM Agent

CtxFuzz's LLM agent performs context-aware reasoning about the target vulnerability. It directly crafts inputs to trigger the vulnerability and guides the fuzzer past semantic barriers, reaching bugs that directed fuzzing alone cannot. The agent proceeds through four stages: *autonomous exploitation*, where multiple parallel LLM agents freely use the provided tools to craft crash-triggering inputs; *blocker analysis*, which diagnoses semantic obstacles that prevent reaching or triggering the target; *key condition exploration*, which recursively decomposes each blocker along function boundaries into concrete key conditions; and *mutator generation*, which produces context-aware mutation scripts that resolve the identified key conditions. Figure 4 and Algorithm 2 illustrate and summarize the pipeline.

Autonomous exploitation. The agent first attempts to directly generate inputs that trigger the bug at the target location. Multiple autonomous LLM-agents run in parallel, equipped with tools for two purposes: analyzing the vulnerability context through call-graph navigation, and interacting with the fuzzer through seed–coverage queries and payload execution. Each exploiter reads the vulnerability description and location, navigates the call graph to understand paths toward the sink, and examines existing seeds in the seed–coverage database to generate new inputs based on fuzzer's progress. Running multiple exploiters in parallel allows diverse strategies to be tried simultaneously. Each exploiter produces test inputs and an *attempt report* that records the call path it tried and the coverage depth it achieved. The generated inputs

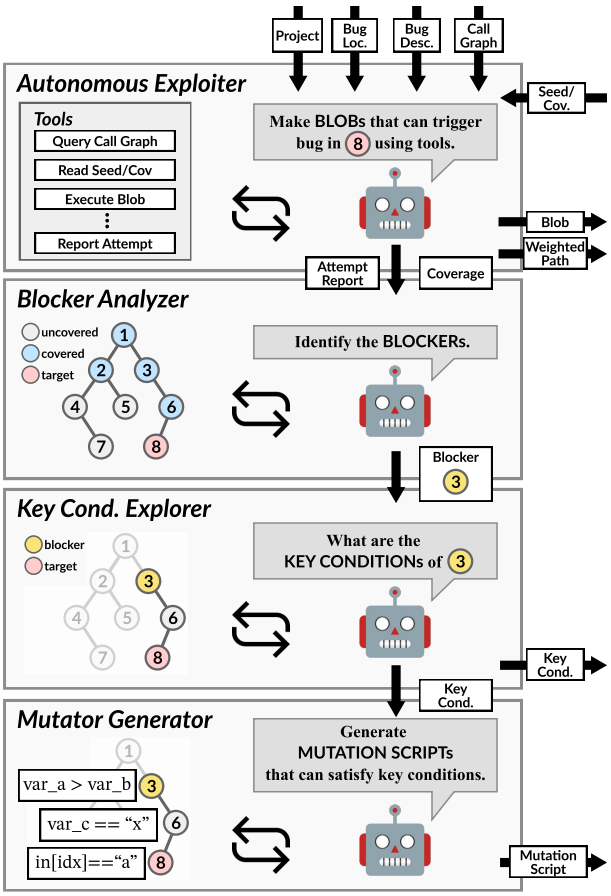

Figure 4: LLM agent architecture of CtxFuzz, consisting of four modules: autonomous exploiter, blocker analyzer, key-condition explorer, and mutator generator. The top arrows denote inputs provided by the project and preprocessing, while the right arrows show context artifacts exchanged with the directed fuzzer.

are submitted to the fuzzer for execution, and the attempt reports inform the subsequent blocker analysis.

Blocker analysis. When autonomous exploitation fails to trigger the target, the agent must understand *why* progress has stalled. Blocker analysis answers this question by identifying *semantic blockers*—conditions that prevent CtxFuzz from triggering the vulnerability. The analysis begins by selecting the most promising call path, ranked by coverage depth from the attempt reports, and emitting it as a *BC* to guide the fuzzer toward functions in the selected call path. The agent then examines the full source code along the selected path together with all prior attempt reports. In practice, blockers take many forms: a dispatch decision that routes execution away from the vulnerable code path, a value constraint that rejects dangerous inputs, a data transformation that sanitizes a field, or a protocol-state requirement that the fuzzer has not yet satisfied. The agent identifies concrete blockers and outputs each

Algorithm 2: LLM agent workflow of CtxFuzz. ■ LLM-driven.

```
Input: sink (sink_func, sink_line), vulnerability description desc, call graph G
Context input: seed corpus S, line coverage set C_line
Context output: blobs S_gen, mutators M, bug contexts BC
// constants: N=5, K=2, D=5
S_gen, M, BC ← ∅
repeat
    R ← ∅
    for i ← 1 to N do in parallel
        (R_i, T_i) ← AutonomousExploit(sink, desc, G, S, C_line)
        R ← R ∪ R_i
        S_gen ← S_gen ∪ T_i
    for j ← 1 to K do
        π_j ← SelectPath(R, G)
        BC ← BC ∪ {(f, path) | f ∈ π_j}
        B ← AnalyzeBlockers(π_j, R, G)
        KC ← ∅
        foreach (func, c_nec) ∈ B do in parallel
            KC ← KC ∪ ExploreKeyCond(func, c_nec, ⟨⟩, 0)
        foreach (c_suf, ctx) ∈ KC do in parallel
            (m, ℓ) ← GenerateMutator(c_suf, ctx)
            if m ≠ ⊥ then
                M ← M ∪ {m}
                BC ← BC ∪ {(ℓ, key_cond)}
until timeout
function ExploreKeyCond(f, c_nec, ctx, d):
    KC ← ∅
    F_callee ← {g ∈ G | (f, g) ∈ edges(G)}
    P ← KeyCondClassify(src(f), c_nec, ctx, F_callee)
    foreach (τ, f_callee, c_suf, c'_nec) ∈ P do in parallel
        if τ = skip then continue
        if τ = leaf or d ≥ D then
            KC ← KC ∪ {(c_suf, ctx)}
        else
            ctx' ← ctx · ⟨f, c_suf⟩
            KC ← KC ∪ ExploreKeyCond(f_callee, c'_nec, ctx', d+1)
    return KC
```

as a pair (*func*, c_{nec}) of the blocking function and the necessary condition it must satisfy.

Key condition exploration. Identifying a blocker is not enough; the agent must determine how to satisfy its necessary condition c_{nec} from the fuzz input. CtxFuzz performs blocker-driven decomposition, resolving each blocker along function boundaries. At each function, the agent analyzes only that function's source code and classifies each path as either a `leaf`—where a sufficient condition c_{suf} for c_{nec} is determined entirely within the current function, a `callee`—where the condition depends on a called function and the agent recurses with an updated necessary condition c'_{nec}, or `skip`—where the path is irrelevant to the necessary condition and can be safely ignored. The accumulated context records the chain of functions and conditions traversed, which later guides mutator generation by providing the data flow path from the fuzz input to the leaf condition. This per-function recursive design constrains the LLM's scope to one function at a time, sharpening its focus and preventing context overflow on deep call paths. Recursion is bounded by depth D; at the limit, all remaining paths are forced to `leaf`. All blockers from the analysis stage are explored in parallel.

Mutator generation. The final stage produces from each key condition a context-aware mutation capable of solving it, which

the fuzzer can execute at scale. Each key condition yields a mutation script (`def mutate(seed: bytes) -> bytes`) that transforms an existing seed to satisfy the identified condition. The generation leverages the full context accumulated during the previous stages: the blocker description, the call-path context from the blocker to the key condition, the harness source code, and the crash condition. Before writing the script, the LLM traces the data flow from the failure site back to the harness input bytes; if it cannot complete the trace, no script is generated rather than a wrong one. Valid scripts are provided to the fuzzer, which applies them across its seed pool. For each script, CtxFuzz emits a *BC* `key_condition` targeting the critical source line that the script is designed to exercise, giving that location a highest scheduling weight in the fuzzer.

Iteration and termination. The agent operates in two nested loops: an *outer loop* that repeats until timeout, and an *inner loop* of K iterations within each round. The inner loop retries the blocker analysis pipeline, each time selecting a different call path, so that a misidentified blocker in one iteration can be corrected by the next. The outer loop restarts the full pipeline from autonomous exploitation, re-observing the fuzzer's current state through the seed–coverage database, so that each round's analysis is grounded in strictly more execution evidence than the last. Note that all generated scripts and *BC*s are immediately available to the fuzzer, which continues fuzzing throughout.

4 Implementation

CtxFuzz is implemented in ~15K lines of Python with a small native tracing helper. The system uses CodeQL [19] for project-independent static analysis and the Claude Agent SDK [4] for LLM orchestration via the Model Context Protocol (MCP) [5]. To support the context interface (§3.3), we built CtxFuzz-fuzzer on top of libFuzzer [29] (C/C++) and Jazzer [13] (Java). CtxFuzz-fuzzer operates fundamentally the same as these fuzzers but additionally exposes line coverage to the agent, polls for *BC*s and mutation scripts, and applies them at runtime as described in §3.4. A single orchestration stack supports both C/C++ and Java; language-specific behavior is confined to CodeQL query selection and the dynamic tracer (e9patch [16] for native targets, with GDB as a fallback; JVM function tracing for Java). Call-graph queries are batched; for Java, queries additionally resolve virtual dispatch and reflection (*e.g.*, `Method.invoke`, `Constructor.newInstance`). The workflow is model-agnostic: substituting a different LLM backend requires changing the model endpoint.

5 Evaluation

We evaluate CtxFuzz along three research questions:

- **RQ1** (§5.2): How does CtxFuzz compare to state-of-the-art directed fuzzers on real-world projects?
- **RQ2** (§5.3): Where does the improvement come from? Do agent-fuzzer cooperation and rich vulnerability context each contribute to effectiveness?
- **RQ3** (§5.4): Can CtxFuzz validate real-world vulnerability reports and expose previously unknown vulnerabilities?

5.1 Methodology

Dataset. We evaluate CtxFuzz on the AIxCC Final competition dataset [14]: 62 vulnerabilities (39 C, 23 Java) across 16 real-world open-source projects, spanning memory corruption, path traversal, and command injection in codebases from 16 KLOC to 4.9 MLOC. We chose AIxCC over Magma [21] to avoid dataset contamination: Magma has been public since 2020 and is likely in LLM training corpora. The AIxCC dataset has not been publicly released. We use `claude-sonnet-4-5-20250929` (training-data cutoff: July 2025), which predates the AIxCC Final at DEF CON in August 2025.

Baselines. We compare against four baselines: *CtxFuzz-fuzzer* (fundamentally libFuzzer [29] and Jazzer [13] running without the agent, §4) and three directed fuzzers (AFLGo [8], WAFLGo [39], Lyso [7]). Directed baselines require LLVM instrumentation and are evaluated on C targets only. Porting directed fuzzers to AIxCC required ~1 person-week each; AFLGo's distance calculation timed out on several large targets, and WAFLGo failed to build 12 of 39 C targets. Locus [47] and HGFuzzer [40] are excluded because their artifacts are not publicly available at the time of submission.

Experimental setup. All experiments run on Google Cloud `n2d-standard-224` instances (224 vCPUs, 896 GB RAM, Ubuntu 24.04), allocating 16 cores per target (one for the LLM agent, 15 for the fuzzer). The per-target timeout is 8 hours. All baseline campaigns run 3 times; CtxFuzz campaigns run once per configuration (~$350 per configuration across all targets). To isolate fuzzing effectiveness from build-system variability, all tools receive pre-built artifacts (instrumented binaries for baselines, sanitizer builds and CodeQL databases for CtxFuzz) and share the same initial seed corpus. The total experimental budget is ~92,500 CPU-hours in cloud compute and ~$1,050 in LLM inference.

Validation. Each candidate PoV is validated through two checks following Kim *et al.* [25]: (1) *sanitizer-based*: the input must trigger a crash under the target sanitizer (ASan [35] for C, Jazzer for Java); (2) *patch-based*: the input must stop crashing after the official fixing patch is applied. An input passing both checks is a *validated trigger*; a target counts as *exposed* only once one is found. Because a single harness may cover multiple injected vulnerabilities, we isolate each target by patching all other vulnerabilities in the same harness, leaving only the target triggerable. We report time-to-reach (TTR), time-to-exposure (TTE), and LLM cost per campaign.

CtxFuzz configurations. We evaluate four CtxFuzz configurations: *CtxFuzz-full* (complete system with full context), *CtxFuzz-loc* (full cooperation but target location only, without vulnerability description or diff), *CtxFuzz-agent* (agent only, no fuzzer cooperation), and *CtxFuzz-fuzzer* (fuzzer only, no agent).

5.2 RQ1: Comparison with State of the Art

CtxFuzz-full exposes 61 of 62 targets We compare CtxFuzz-full against all baselines (Figure 5 and Table 1). Across all 62 targets, CtxFuzz-full reaches all 62 and exposes 61 (98%); §6 analyzes the single exception. Prior directed fuzzers, which apply only to the 39 C targets, expose far fewer: AFLGo 13, WAFLGo 14, and

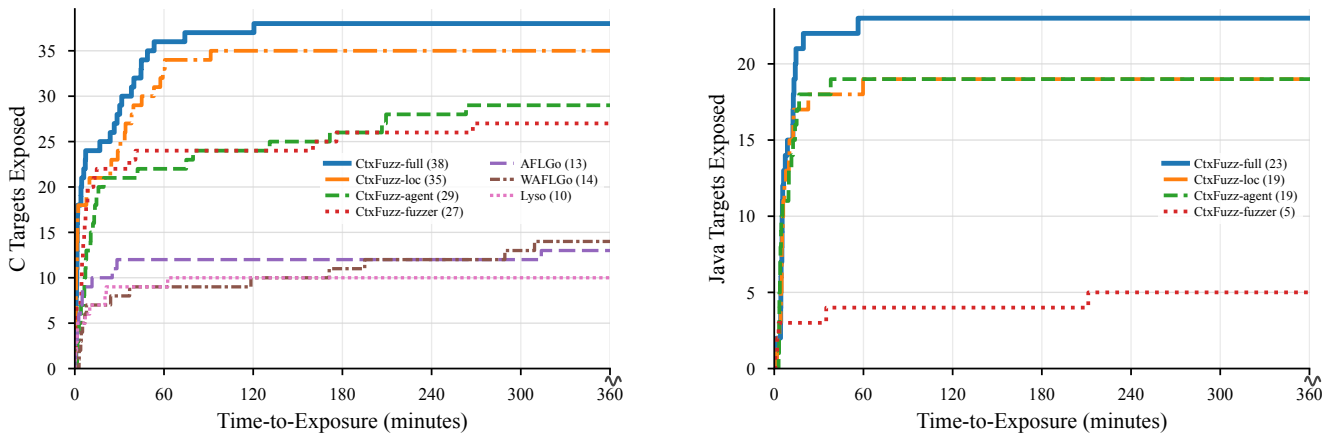

Figure 5: TTE cactus plots split by language (left: 39 C targets; right: 23 Java targets). On the left, all tools compete but CtxFuzz-full dominates; on the right, the directed fuzzers do not apply, and CtxFuzz-fuzzer exposes only five of 23 targets while every agent-containing variant exposes at least 19. The x-axis is truncated at 360 minutes, as no configuration shows additional growth after that point.

Lyso 10, compared to CtxFuzz-full's 38 (Figure 5). Of the 61 targets CtxFuzz-full exposes, 29 are exposed by no baseline—neither CtxFuzz-fuzzer nor any directed fuzzer exposes them.

Prior directed fuzzers are limited in applicability. On the 39 C targets, the only ones the directed fuzzers support, even the undirected CtxFuzz-fuzzer (27/39) outperforms every directed fuzzer by nearly a factor of two, exposing 7 targets that no directed fuzzer does. These tools also face applicability barriers (§5.1): they require LLVM instrumentation and thus cannot target Java at all, and even on C, WAFLGo failed to build 12 of the 39 targets. By contrast, CtxFuzz ran on all 62 targets in both C and Java without per-target modification. The total LLM inference cost for CtxFuzz-full is approximately $300 across all 62 targets. RQ2 decomposes these gains by system component and by language.

> **RQ1 finding:** CtxFuzz-full exposes 61 of 62 targets, uniquely exposing 29 that no baseline does. Unlike prior directed fuzzers, it covers both C and Java targets without per-target modification.

5.3 RQ2: Component Analysis

We ablate CtxFuzz's two key design choices using the four configurations (§5.1): comparing *full* with *agent* analyzes the fuzzer's contribution; comparing *full* with *loc* investigates the value of rich vulnerability context.

Fuzzer–agent cooperation is the larger source of improvement. Removing cooperation drops success from 61 to 48 targets. On the 48 targets both configurations expose, CtxFuzz-full is on average 22 minutes faster (Table 1). The full system also costs less ($300 vs. $317 for agent-only). No target is exposed by CtxFuzz-agent but missed by CtxFuzz-full, so cooperation never hurts.

Rich vulnerability context adds distinct value. CtxFuzz-full exposes 61 targets, compared with 54 for CtxFuzz-loc—a gap of 7 targets. The median TTEs of the two configurations are similar on their respective exposed sets, suggesting that rich context primarily determines whether a target can be exposed at all, rather than how quickly. CtxFuzz-loc also incurs higher total LLM cost than CtxFuzz-full ($412 vs. $300), consistent with richer context reducing unproductive exploration.

Cooperation plays different roles by language. On C (Figure 5, left), CtxFuzz-fuzzer already exposes 27 of 39 targets and cooperation adds 11 more. On Java (Figure 5, right), CtxFuzz-fuzzer exposes only 5 of 23. Java targets typically involve complex structured inputs (*e.g.*, XML documents, binary record streams) where semantic understanding of the format is more effective than random mutation; agent-driven reasoning is accordingly the primary source of progress: CtxFuzz-agent alone exposes 19 of 23, and cooperation closes the remaining four (targets 44, 45, 50, and 60) to reach all 23.

Case study: lcms. The target vulnerability requires reaching `CubeSize` with a dimension $\leq$ 1. The agent identifies the blocking mechanism: `WriteInputLUT` unconditionally sets `cmsFLAGS_FORCE_CLUT` (line 903) before calling `_cmsOptimizePipeline` (line 904), forcing `OptimizeByResampling` to always add a CLUT stage after any CurveSet. The agent-only configuration identifies this but cannot synthesize a valid input blob end-to-end, because the optimization pipeline's internal state is too complex to reproduce from scratch. The fuzzer-only baseline lacks the semantic knowledge to target the specific four-byte field that controls CLUT dimensions. Reaching the target and triggering it separate here (Table 1, target 13): CtxFuzz-agent reaches the target line at 320 s but never validates it, whereas CtxFuzz-full reaches it at 291 s and validates at 2384 s. No baseline achieves either within the 8-hour budget. Through blocker-driven decomposition, CtxFuzz-full generates the succeeding script (Figure 6): a single four-byte write at a fixed offset forces `nGridPoints` = 1. The agent derives *what* to change; the fuzzer validates *whether* it works.

Table 1: Per-target results on the AIxCC Final benchmark (Targets 1–39 are C; Targets 40–62 are Java).

				CtxFuzz-full			CtxFuzz-loc			CtxFuzz-agent			CtxFuzz-fuzzer		AFLGo		WAFLGo		Lyso	
#	Project	SLOC	CWE	TTR	TTE	LLM	TTR	TTE	LLM	TTR	TTE	LLM	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE
1	curl▲	238K	476	90	98	2.8	125	134	2.7	137	169	1.3	·	·	·	·	·	·	·	·
2	curl▲	238K	476	276	277	14.0	·	·	21.4	952	953	15.2	·	·	·	·	·	·	·	·
3	curl▲	238K	121	25	108	2.7	25	109	1.8	112	155	1.4	329	·	·	·	49	·	34	·
4	curl▲	238K	476	436	438	6.6	2699	2700	36.8	2530	2532	23.1	·	·	·	·	·	·	·	·
5	curl▲	237K	134	26	70	2	25	47	0.3	635	1168	5.3	154	292	·	·	35	7108	2	·
6	curl▲	237K	787	135	248	3.7	186	·	10.1	342	·	16.3	·	·	·	·	·	·	·	·
7	freerdp▲	453K	122	87	92	2	99	106	0.9	98	103	0.9	·	·	·	·	·	·	·	·
8	freerdp▲	453K	123	358	361	4.0	314	468	5.2	211	238	3.3	3963	·	·	·	104	·	7	·
9	libavif▲	44K	126	417	423	3.5	588	591	5.6	437	440	4.1	444	884	·	·	10925	17356	1005	1218
10	libexif▲	16K	122	24	2931	26.4	23	·	28.1	248	·	14.6	32	·	2087	·	267	·	746	·
11	libexif▲	15K	121	23	28	0.5	23	27	0.1	285	812	4.1	340	340	21	21	170	170	69	69
12	libxml2▲	201K	122	38	76	0.8	36	81	1.1	49	782	4.0	197	281	839	·	11708	18554	6102	·
13	little-cms	87K	476	291	2384	59.6	330	2293	31.6	320	·	17.9	·	·	·	·	·	·	·	·
14	little-cms	87K	126	24	·	23.5	22	·	38.7	189	·	13.9	·	·	·	·	·	·	·	·
15	mongoose▲	352K	121	117	117	1.6	529	529	7.4	109	109	1.9	·	·	·	·	·	·	·	·
16	mongoose▲	352K	125	249	249	2.3	112	112	1.2	463	463	3.5	6029	16064	12	1511	149	10268	117	·
17	mongoose	352K	121	91	91	0.9	83	83	0.9	93	93	0.9	24	9612	0	695	122	11701	1	3749
18	shadowsocks	19K	126	41	44	0.1	27	30	0.4	262	386	1.3	16	20	·	·	282	470	90	·
19	shadowsocks	19K	126	50	78	0.2	28	37	0.6	395	396	1.0	17	26	·	·	717	1445	460	·
20	shadowsocks	19K	126	31	45	0.1	28	44	0.3	203	426	2.0	16	33	·	·	102	261	1	·
21	shadowsocks	19K	126	34	50	0.3	30	43	0.5	216	361	1.0	16	30	18828	18828	81	313	1	·
22	shadowsocks	19K	126	25	41	1.2	24	30	0.5	118	250	1.5	16	270	·	·	34	403	1	·
23	systemd	740K	787	42	45	0.1	38	40	0.1	527	643	1.1	28	28	7	96	57	239	58	95
24	systemd	740K	122	36	97	0.6	43	51	0.3	457	731	2.0	75	75	111	111	·	·	602	602
25	systemd	740K	415	56	56	0.3	54	54	0.3	632	632	3.0	30	30	159	159	·	·	0	0
26	systemd	740K	415	37	48	0.3	33	42	0.4	43	·	1.9	261	261	30	30	2212	2212	1268	1268
27	wireshark	4.9M	121	1710	1894	1.3	1590	1732	1.2	12372	12561	5.5	325	325	297	297	·	·	·	·
28	wireshark	4.9M	416	1497	7230	8.9	1349	5484	13.8	·	·	8.5	10560	10560	·	·	·	·	·	·
29	wireshark	4.9M	134	1261	1705	2.0	1386	3633	5.7	·	·	13.2	·	·	·	·	·	·	·	·
30	wireshark	4.9M	122	57	1591	2.0	1097	1472	2.5	·	·	11.1	377	377	·	·	·	·	·	·
31	wireshark	4.9M	121	2011	2671	3.9	2823	3451	4.6	4519	4773	4.1	456	456	·	·	·	·	·	·
32	wireshark	4.9M	457	77	2685	6.1	84	3194	5.3	15610	15791	5.7	432	432	10	10	·	·	16	16
33	wireshark▲	4.9M	787	52	1428	1.5	53	2014	1.9	3227	·	5.7	751	751	1702	1702	·	·	·	·
34	wireshark▲	4.9M	121	2043	2284	3.6	2102	2364	3.0	4652	12396	6.5	511	511	·	·	·	·	·	·
35	wireshark▲	4.9M	122	1504	1812	2.2	1670	1840	1.3	3259	4496	1.9	512	512	·	·	·	·	·	·
36	wireshark▲	4.9M	126	2826	3201	2.7	3016	3551	3.0	·	·	7.5	2199	2199	·	·	·	·	·	·
37	wireshark▲	4.9M	129	55	1010	1.1	66	1432	1.2	10015	10293	5.7	399	399	310	310	·	·	419	419
38	wireshark▲	4.9M	120	4442	4442	7.8	2047	2047	3.4	6838	7865	6.6	2450	2450	·	·	·	·	·	·
39	xz	41K	416	38	91	0.4	37	83	0.7	216	875	3.9	23	225	3	227	77	179	0	114
40	commons-compress▲	75K	400	340	340	2.9	299	299	3.0	227	227	1.8	49	49	N/A	N/A	N/A	N/A	N/A	N/A
41	commons-compress▲	76K	35	74	268	2.6	69	344	2.6	134	207	2	15	·	N/A	N/A	N/A	N/A	N/A	N/A
42	commons-compress▲	76K	77	68	253	1.5	55	209	1.0	169	176	1.3	15	·	N/A	N/A	N/A	N/A	N/A	N/A
43	commons-compress▲	76K	35	47	260	2.5	61	305	3.0	160	196	1.7	20	·	N/A	N/A	N/A	N/A	N/A	N/A
44	commons-compress▲	76K	35	54	878	4.9	53	·	4.5	189	·	4.3	18	·	N/A	N/A	N/A	N/A	N/A	N/A
45	logging-log4j2▲	54K	917	75	98	0.8	84	111	0.4	347	·	0.8	50	·	N/A	N/A	N/A	N/A	N/A	N/A
46	pdfbox▲	167K	77	259	278	2.5	437	472	1.9	231	240	2.3	92	116	N/A	N/A	N/A	N/A	N/A	N/A
47	pdfbox	167K	611	343	328	3.9	222	260	4.1	157	197	2.6	·	·	N/A	N/A	N/A	N/A	N/A	N/A
48	pdfbox	167K	611	267	314	2.8	216	303	5.5	192	206	2.3	·	·	N/A	N/A	N/A	N/A	N/A	N/A
49	pdfbox	167K	835	833	849	4.4	752	764	8.9	171	840	6.8	13891	·	N/A	N/A	N/A	N/A	N/A	N/A
50	pdfbox	167K	835	754	758	4.0	268	613	6.3	·	·	8.8	·	·	N/A	N/A	N/A	N/A	N/A	N/A
51	pdfbox	167K	789	255	266	3.6	267	270	4.3	251	251	2.6	·	·	N/A	N/A	N/A	N/A	N/A	N/A
52	pdfbox	167K	789	249	307	3.5	226	·	9.1	241	576	4.5	·	·	N/A	N/A	N/A	N/A	N/A	N/A
53	pdfbox	167K	834	749	754	1.3	766	776	1.1	753	759	1.0	·	·	N/A	N/A	N/A	N/A	N/A	N/A
54	pdfbox	167K	834	172	1174	7.7	180	·	10.2	138	694	3.0	5923	12674	N/A	N/A	N/A	N/A	N/A	N/A
55	poi▲	433K	918	88	378	3	84	410	2.5	273	273	2.5	22	·	N/A	N/A	N/A	N/A	N/A	N/A
56	poi▲	433K	121	98	445	7.9	83	590	10.4	254	334	3.5	40	·	N/A	N/A	N/A	N/A	N/A	N/A
57	poi	433K	834	66	770	7.7	72	1375	11.4	1166	2279	7.6	41	·	N/A	N/A	N/A	N/A	N/A	N/A
58	poi	433K	789	61	77	0.9	107	127	1.1	981	1002	2.1	47	2097	N/A	N/A	N/A	N/A	N/A	N/A
59	poi	433K	35	201	319	6.2	265	362	8.0	486	590	4.9	99	175	N/A	N/A	N/A	N/A	N/A	N/A
60	poi	433K	695	64	3384	19.0	67	3583	52.7	892	·	14.8	·	·	N/A	N/A	N/A	N/A	N/A	N/A
61	poi	433K	382	78	532	1.7	72	320	1.3	247	285	1.5	40	·	N/A	N/A	N/A	N/A	N/A	N/A
62	tika▲	188K	834	783	795	1.0	·	·	13.8	891	905	1.8	·	·	N/A	N/A	N/A	N/A	N/A	N/A
			Med.	89	319	2.6	92	332	2.8	262	520	3.4	92	332	111	227	122	957	58	266
			Avg.	429	891	4.8	459	953	6.7	1392	1899	5.1	1141	1955	1628	1846	1594	5048	524	755
			# Reached	**62/62 (100%)**			**60/62 (97%)**			**57/62 (92%)**			**45/62 (73%)**		**15/39 (38%)**		**17/39 (44%)**		**21/39 (54%)**	
			# Exposed	**61/62 (98%)**			**54/62 (87%)**			**48/62 (77%)**			**32/62 (52%)**		**13/39 (33%)**		**14/39 (36%)**		**10/39 (26%)**	

TTR/TTE are in seconds; LLM is in USD; · denotes timeout (8 h); BF denotes build failure; N/A denotes not applicable to Java; and ▲: diff available.
TTE cells use a light blue heatmap. Lighter cells indicate faster exposures; darker cells indicate slower exposures. TTR and LLM columns are unshaded.

```
def mutate(seed: bytes) -> bytes:
  if len(seed) < 12:
    seed = seed + b'\x00' * (12 - len(seed))
  mutated = bytearray(seed)
  # Set bits 16-23 to force nGridPoints = 1,
  # creating a CLUT with all Dimensions[i] = 1
  # that triggers dim <= 1 in CubeSize
  mutated[8:12] = struct.pack('<I', 0x00010000)
  return bytes(mutated)
```

Figure 6: Mutation script for lcms: a four-byte write at a fixed offset forces `nGridPoints` = 1 in `CubeSize`.

```
def mutate(seed: bytes) -> bytes:
  seed = bytearray(seed)
  target_name = b'aixcc-rocks'
  bound_sheet_sid = bytes([0x85, 0x00])
  pos = 0
  while pos < len(seed) - 10:
    if seed[pos:pos+2] == bound_sheet_sid:
      rec_len = struct.unpack('<H',
        bytes(seed[pos+2:pos+4]))[0]
      if rec_len >= 8:
        new_len = 4+2+1+1+len(target_name)
        seed[pos+2:pos+4] = \
          struct.pack('<H', new_len)
        off = pos + 10 # after SID+Len+BOF+Opts
        seed[off] = len(target_name)
        seed[off+1] = 0x00 # 8-bit chars
        seed[off+2:off+2+len(target_name)] \
          = target_name
        break
    pos += 1
  return bytes(seed)
```

Figure 7: Mutation script for Apache POI: runtime scanning for BoundSheetRecord SID to locate and overwrite the sheet-name field.

Case study: Apache POI. Apache POI (target 60), one of the four Java targets CtxFuzz-agent misses, illustrates a different kind of blocker: the mutation is semantically simple, but its location is input-dependent. Triggering it requires overwriting a sheet name with `aixcc-rocks`, yet the BoundSheetRecord (`SID 0x0085`) can appear at any position within a BIFF binary stream. The agent-only configuration understands the BIFF record structure but produces inputs that fail validation because the surrounding workbook context is missing. The fuzzer-only baseline cannot discover the correct offset through generic mutation. CtxFuzz-full generates a script (Figure 7) that scans for the BoundSheetRecord SID at runtime, reads the record length, and overwrites the sheet-name field in place, adjusting lengths accordingly.

> **RQ2 finding:** Both cooperation and rich context contribute measurably. Cooperation is the larger effect (61 vs. 48 targets); rich context provides an additional gain (61 vs. 54). The two components play different roles by language: the fuzzer drives coverage on C, while the agent's semantic reasoning drives progress on Java.

5.4 RQ3: Real-World Validation

As an initial step toward a full discovery-to-validation pipeline, we use Code Review for Claude Code [6], an automated code audit tool, to identify vulnerability candidates and validate them using CtxFuzz. We target OSS-Fuzz [20] projects; OSS-Fuzz is Google's continuous fuzzing platform hosting over 1,000 open-source projects. Among these projects, we select libyang as a case study, a YANG data-modeling library that DARPA nominated as a critical target. Code Review reported 17 candidate vulnerabilities. We then asked it to select the five most likely to be reachable through the project's existing fuzzing harnesses, and ran CtxFuzz on those five. Each run receives the target location and the audit's description as context and attempts to trigger the vulnerability.

Results. Of the five selected candidates, CtxFuzz produced validated triggers for two, both previously unknown vulnerabilities. Each candidate confirmed by CtxFuzz (*i.e.*, the fuzzer triggers a sanitizer crash) is then manually verified as a true positive. One is a critical vulnerability (CVSS 9.8) [2]; the other is a medium-severity vulnerability (CVSS 5.3) [3]. Both have since been publicly disclosed. As this was an initial step, we are actively building a full pipeline, described in §6.

> **RQ3 finding:** CtxFuzz validates real-world vulnerability candidates and exposes two zero-day vulnerabilities.

6 Discussion

Toward a full validation pipeline. Open-source maintainers are increasingly overwhelmed by ungrounded AI-generated vulnerability reports [33, 37]. CtxFuzz's execution-grounded validation directly addresses this burden: by driving execution through the target code path, the fuzzer naturally validates candidates via sanitizer-detected crashes (*e.g.*, ASan, Jazzer). Our initial experiment pairing Code Review for Claude Code [6] with CtxFuzz for validation produced two zero-day vulnerabilities (§5.4), demonstrating the approach is feasible. Building the full closed-loop pipeline—automated discovery, CtxFuzz-based validation, and actionable reporting integrated into CI/CD workflows— is our primary next step. CtxFuzz is being integrated into the OSS-CRS platform [12] under the OpenSSF and the Linux Foundation, which will provide the infrastructure to deploy and evaluate this pipeline at scale.

Limitations of our approach. The one vulnerability CtxFuzz does not expose (Table 1, target 14) marks a boundary of the approach, for two reasons. First, the provided location marks where the bug was *introduced*—the faulty check `IsProperColorSpace`, which every transform creation calls—so CtxFuzz reaches it in 24 s and the reach signal saturates at once. Locating where the bug can be *triggered* is left entirely to the agent: the over-read fires later, during transform execution, in a pixel unroller the description never names. Second, that site defeats our line-level granularity (§3.3). The faulting statement lies inside the loop that unrolls one channel per iteration, so line coverage marks it covered on every benign transform; what separates a crash is the iteration on which the read walks past the four-byte stack slot supplied as the input buffer—a distinction line coverage cannot express. The agent can set the profile fields that govern the channel count, but nothing distinguishes a profile one field short of the crash from one far from it, so decomposition has no gradient to follow.

Threats to validity. CtxFuzz relies on LLM queries, which incur token costs. Because LLMs are probabilistic, results may vary across runs, model versions, and providers. For baseline fuzzers, we

average results over multiple independent runs; for CtxFuzz, we report a single run per configuration due to inference cost.

Ethical considerations. CtxFuzz takes vulnerability descriptions as input and generates triggering inputs, which creates dual-use risk. By design, CtxFuzz is a *validation* tool: it confirms or refutes suspected vulnerabilities rather than autonomously discovering novel attack surfaces. All vulnerabilities discovered in this work were reported through responsible disclosure.

7 Related Work

Directed Grey-box Fuzzing. Since AFLGo [8], DGF has been refined along several axes. Follow-up work improves distance estimation [7, 8, 10, 15, 26], constraint- and path-based guidance [7, 22, 23, 26, 30], and data-flow-aware metrics [15, 24, 30, 32, 49]. Others explore alternative formulations such as Monte Carlo counting and dynamic target selection [17, 36], or domain-specific extensions for the Linux kernel [38, 42], static-analysis alarm verification [7, 49], and target-aware mutation [10, 11, 15, 23, 38]. These advances share a common model: guidance derives from compiler-level instrumentation, and targets are bare locations. CtxFuzz additionally consumes the vulnerability description through a context interface between the agent and the fuzzer.

LLM-Guided Fuzzing. Recent work uses LLMs for structure extraction, generator synthesis, and constraint solving [9, 40, 41, 46]. ChatAFL [31] extracts protocol grammars from RFCs; subsequent works synthesize generation-based fuzzers [9], non-textual input generators [46], and branch-constraint solvers [41]. HGFuzzer [40], the closest to directed fuzzing, produces reachable seeds and target-specific mutators in a staged pipeline. In all cases the LLM guidance is generated before or outside the fuzzing loop. CtxFuzz keeps the agent in the loop, updating its analysis from execution feedback.

Agentic Vulnerability Discovery. Locus [47] synthesizes intermediate predicates offline with an LLM agent and symbolic execution, then compiles them into the binary for a downstream fuzzer that runs without the agent. PBFuzz [43] takes an agentic approach to PoV generation but replaces rather than cooperates with a directed fuzzer. CtxFuzz keeps both agent and fuzzer in the loop, connecting them through a context interface so that execution feedback informs blocker analysis and the fuzzer applies the resulting mutations.

8 Conclusion

To validate vulnerability reports with execution-grounded reasoning, we presented CtxFuzz, a context-aware directed fuzzer in which an LLM agent and a coverage-guided fuzzer cooperate at runtime by exchanging bug contexts, mutation scripts, and coverage information through a context interface. On the AIxCC Final dataset (62 targets across 16 real-world projects in C and Java), CtxFuzz outperformed state-of-the-art directed fuzzers, while ablations confirmed that both runtime cooperation and rich vulnerability context are necessary for the gains. By pairing CtxFuzz with automated code review, we further exposed two zero-day vulnerabilities in a DARPA-nominated critical target, demonstrating that an end-to-end agentic validation pipeline is feasible in practice.

Data Availability

The CtxFuzz source code and all baseline fuzzer configurations are available online.[1] The AIxCC Final challenges that constitute our benchmark have since been released publicly in DARPA's competition archive [14], so the evaluation targets can be reconstructed in full. As CtxFuzz is integrated into the OSS-CRS platform [12], its forthcoming CRSBench harness—which evaluates cyber reasoning systems under a shared sanitizer configuration and resource budget—will support directly comparable community benchmarking. Zero-day vulnerabilities discovered during evaluation were reported to maintainers following responsible-disclosure practices.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00441762 and No. RS-2024-00440780).

AI disclosure. In accordance with ACM policy, we disclose that generative AI tools were used during the preparation of this paper. Specifically, Claude Code (Anthropic) and Codex (OpenAI) were used to assist with drafting, editing, and reviewing text. All AI-generated content was reviewed and verified by the authors.

References

[1] [n. d.]. Little CMS. https://www.littlecms.com/. Accessed: 2026-03-26.
[2] 2026. GHSA-9j24-j628-8cm4: Heap use-after-free in schema compiler. https://github.com/CESNET/libyang/security/advisories/GHSA-9j24-j628-8cm4. libyang security advisory. CVSS 9.8. Accessed: 2026-03-27.
[3] 2026. GHSA-rf85-7hqw-w52q: Assertion failed in lysp_schema_nodeid_match. https://github.com/CESNET/libyang/security/advisories/GHSA-rf85-7hqw-w52q. libyang security advisory. CVSS 5.3. Accessed: 2026-03-27.
[4] Anthropic. 2025. Claude Agent Python SDK. https://github.com/anthropics/claude-agent-sdk-python.
[5] Anthropic. 2025. Model Context Protocol. https://modelcontextprotocol.io/.
[6] Anthropic. 2026. Code Review for Claude Code. https://claude.com/blog/code-review. Accessed: 2026-03-25.
[7] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. In *USENIX Security Symposium (Security)*.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3133956.3134020
[9] Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. 2025. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *USENIX Security Symposium (Security)*.
[10] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3243734.3243849
[11] Yiyang Chen, Chao Zhang, Long Wang, Wenyu Zhu, Changhua Luo, Nuoqi Gui, Zheyu Ma, Xingjian Zhang, and Bingkai Su. 2025. IDFuzz: Intelligent Directed Grey-box Fuzzing. In *USENIX Security Symposium (Security)*.
[12] Andrew Chin, Dongkwan Kim, Yu-Fu Fu, Fabian Fleischer, Youngjoon Kim, HyungSeok Han, Cen Zhang, Brian Junekyu Lee, Hanqing Zhao, and Taesoo Kim. 2026. OSS-CRS: Liberating AIxCC Cyber Reasoning Systems for Real-World Open-Source Security. https://doi.org/10.48550/arXiv.2603.08566 arXiv:2603.08566 [cs.CR]
[13] Code Intelligence. 2024. Jazzer – Coverage-Guided Fuzzing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer.
[14] DARPA. 2026. AIxCC Competition Archive: Challenges. https://archive.aicybercchallenge.com/challenges/. Accessed: 2026-08-03.
[15] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510197

[1] https://zenodo.org/records/19248087

[16] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery. https://doi.org/10.1145/3385412.3385972

[17] Haoran Fang, Kaikai Zhang, Donghui Yu, and Yuanyuan Zhang. 2024. DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3650212.3680324

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

[19] GitHub. 2026. CodeQL. https://codeql.github.com/.

[20] Google. 2024. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29. https://doi.org/10.1145/3428334

[22] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *IEEE Symposium on Security and Privacy (Oakland)*. https://doi.org/10.1109/SP46214.2022.9833751

[23] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan: Efficient Multi-target Directed Greybox Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*. https://doi.org/10.1109/SP54263.2024.00059

[24] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *USENIX Security Symposium (Security)*.

[25] Tae Eun Kim, Jaeseung Choi, Seongjae Im, Kihong Heo, and Sang Kil Cha. 2024. Evaluating Directed Fuzzers: Are We Heading in the Right Direction?. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/3643741

[26] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *USENIX Security Symposium (Security)*.

[27] Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Fengyu Liu, Tian Chen, and Min Yang. 2025. Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection. In *USENIX Security Symposium (Security)*.

[28] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638

[29] LLVM Project. 2024. libFuzzer – A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[30] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *IEEE Symposium on Security and Privacy (Oakland)*. https://doi.org/10.1109/SP46215.2023.10179296

[31] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2024.24556

[32] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[33] OpenSSF Vulnerability Disclosures WG. 2025. AI-SLOP: Develop best current practices for Open Source maintainers. GitHub Issue #178. https://github.com/ossf/wg-vulnerability-disclosures/issues/178.

[34] Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. LongCodeBench: Evaluating Coding LLMs at 1M Context Windows. *arXiv preprint arXiv:2505.07897* (2025). https://doi.org/10.48550/arXiv.2505.07897

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[36] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3548606.3560648

[37] Daniel Stenberg. 2026. The end of the curl bug-bounty. Daniel Stenberg Blog. https://daniel.haxx.se/blog/2026/01/26/the-end-of-the-curl-bug-bounty/.

[38] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3576915.3623146

[39] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical Code Guided Directed Greybox Fuzzing for Commits. In *USENIX Security Symposium (Security)*.

[40] Hanxiang Xu, Yanjie Zhao, and Haoyu Wang. 2025. Directed Greybox Fuzzing via Large Language Model. https://doi.org/10.48550/arXiv.2505.03425 arXiv:2505.03425 [cs.CR]

[41] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. 2025. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *USENIX Security Symposium (Security)*.

[42] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *USENIX Security Symposium (Security)*.

[43] Haochen Zeng, Andrew Bao, Jiajun Cheng, and Chengyu Song. 2025. PBFuzz: Agentic Directed Fuzzing for PoV Generation. https://doi.org/10.48550/arXiv.2512.04611 arXiv:2512.04611 [cs.CR]

[44] Cen Zhang, Younggi Park, Fabian Fleischer, Yu-Fu Fu, Jiho Kim, Dongkwan Kim, Youngjoon Kim, Qingxiao Xu, Andrew Chin, Ze Sheng, Hanqing Zhao, Michael Pelican, David J. Musliner, Jeff Huang, Jon Silliman, Mikel Mcdaniel, Jefferson Casavant, Isaac Goldthwaite, Nicholas Vidovich, Matthew Lehman, and Taesoo Kim. 2026. SoK: DARPA's AI Cyber Challenge (AIxCC): Competition Design, Architectures, and Lessons Learned. https://doi.org/10.48550/arXiv.2602.07666 arXiv:2602.07666 [cs.CR]

[45] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. 1223–1235. https://doi.org/10.1145/3650212.3680355

[46] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *USENIX Security Symposium (Security)*.

[47] Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2026. Locus: Agentic Predicate Synthesis for Directed Fuzzing. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.48550/arXiv.2508.21302

[48] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *USENIX Security Symposium (Security)*.

[49] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium (Security)*.

Received 2026-03-26; accepted 2026-06-18