# DF-FUZZ: DEPENDENCY-GUIDED CONTEXT RETRIEVAL FOR LLM-ASSISTED LANGUAGE PROCESSOR FUZZING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Language processor fuzzing assisted by large language models (LLMs) has shown substantial promise in synthesizing test programs that satisfy high-level syntactic and semantic specifications to penetrate deep code validations. However, existing LLM-driven constraint-solving fuzzers face a severe semantic scarcity bottleneck when testing large-scale compiler systems. Conventional physical function slicing retains only the enclosing function containing the target bottleneck basic block, severing critical semantic linkages to external type declarations, preprocessor macros, and class hierarchies. Consequently, LLM-generated test programs suffer from low compilation pass rates. Meanwhile, interactive retrieval designs incur prohibitive round-trip latency and token overhead, frequently triggering format hallucinations and state-machine deadlocks. To overcome these limitations, we present DF-Fuzz, a dependency-guided context retrieval framework for language processor fuzzing. DF-Fuzz introduces the Dual-Flow Hybrid Dependency Graph (DF-HDG) to unify intra-procedural control constraints and inter-procedural, cross-file data dependencies within an offline-constructed representation. Operating via a dual-track architecture, DF-Fuzz performs reverse dominator tree slicing to isolate the minimal condition chains governing branch reachability, while concurrently employing inter-procedural symbol resolution with namespace wrapping to reconstruct self-contained scopes. To mitigate context bloat, DF-Fuzz integrates Jaccard-similarity-based semantic pruning with an API co-occurrence few-shot retrieval mechanism. Furthermore, DF-Fuzz incorporates an adaptive feedback-driven self-healing loop that translates local compiler diagnostic errors into dynamic exemption weights, repairing omitted context in subsequent iterations at zero additional LLM interaction cost. We evaluated DF-Fuzz across eight real-world language processors spanning six programming languages, including Clang++, Flang, Hermes, and Lua. In 24-hour comparative evaluations, DF-Fuzz achieved 100,153 edge coverage on Clang++, outperforming AFL++ by 78.8% and the state-of-the-art LLM baseline HLPFuzz by 5.7%, while reducing API operational costs by 60.3% and increasing execution throughput by 26.9%. In extended testing across all targets, DF-Fuzz discovered 45 previously unknown vulnerabilities, including 10 heap use-after-free and 8 heap buffer overflow vulnerabilities. Our code and artifacts are publicly available at `https://osf.io/dkqz4/overview?view_only=d75600e5f99f41a6a849bbc36515e919`.

## 1 INTRODUCTION

Modern language processors, including compilers, interpreters, and runtime engines, constitute the foundational infrastructure of modern computing stacks (Ma, 2023). Ensuring their correctness, robustness, and security is paramount; bugs in compiler frontends, middle-end optimizers, or code generation backends can lead to compiler crashes, miscompilations, or critical security bypasses (Manès et al., 2019; Zhu et al., 2022). However, modern production-grade compilers, such as Clang/LLVM, comprise millions of lines of intricate C++ code characterized by stringent lexical, syntactic, and semantic static validation rules.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Traditional coverage-guided fuzzers (e.g., AFL++ (Fioraldi et al., 2020), LibAFL (Fioraldi et al., 2022), CollAFL (Gan et al., 2018), MOPT (Lyu et al., 2019)) generate test cases primarily via random bit- and byte-level mutations. Lacking deep semantic awareness, these inputs overwhelmingly fail early frontend validation stages (e.g., AST construction, type checking, symbol resolution) and are immediately discarded (Padhye et al., 2019), struggling to exercise deeper compiler optimization and code generation pipelines (Lemieux & Sen, 2018). Although concolic execution and symbolic solving penetrate narrow branch guards (Stephens et al., 2016; Yun et al., 2018), they suffer from combinatorial path explosion on large codebases. To address this barrier, recent research leverages large language models (LLMs) to guide fuzzing (e.g., HLPFuzz (Yang et al., 2025b), Fuzz4All (Xia et al., 2024), TitanFuzz (Deng et al., 2023), FuzzGPT (Deng et al., 2024), ChatFuzz (Hu et al., 2023)). In particular, constraint-solving hybrid fuzzers formulate bottleneck branches as targeted code transformation tasks for LLMs, generating semantic mutations that satisfy path conditions leading to uncovered code (Wang et al., 2024).

Despite their potential, existing LLM-assisted constraint-solving fuzzers encounter two fundamental technical barriers when applied to large-scale, real-world compiler codebases:

*Semantic Scarcity in Isolated Slices.* Existing techniques typically extract an isolated physical slice of the enclosing function containing the target bottleneck basic block. Modern compilers, however, rely heavily on object-oriented hierarchies, extensive template metaprogramming, complex macro expansions, and multi-layered class inheritance. Isolated function slicing severs the semantic connection between the bottleneck condition and external type definitions, data structures, and helper interfaces. Deprived of this contextual knowledge, LLMs frequently synthesize invalid symbol references or mismatched types, causing over 90% of generated test cases to fail compiler frontend checks.

*High Latency and Fragility of Interactive Retrieval.* Prior efforts attempt to retrieve missing semantics by incorporating dynamic tool-invocation interfaces (e.g., prompting the LLM to issue {`DEFINITION_LOOKUP`} queries across multiple dialogue turns). This interactive paradigm introduces significant inefficiencies: multi-round RPCs result in exponential token consumption as entire interaction histories are repeatedly transmitted, while network round-trips severely throttle fuzzer throughput. Furthermore, LLMs frequently experience format hallucinations (Ji et al., 2023) or context-window truncation over extended conversations, causing session deadlocks and halting the fuzzing campaign.

To resolve these challenges, we present DF-Fuzz, a dependency-guided context retrieval framework designed for language processor fuzzing. While offline graph modeling and lightweight retrieval have been explored in repository-level code completion and code search (Shrivastava et al., 2023; Chen et al., 2024; Shen et al., 2024; Zhang et al., 2025b), as well as in kernel fuzzing with dependency-based RAG (Zhang et al., 2025a) and fuzz driver synthesis (Xu et al., 2025), language processor fuzzing imposes distinct constraints: the requirement for high test execution throughput and the strict demand that generated code satisfy exact compiler semantic checks. Consequently, multi-round interactive retrieval frameworks cannot be directly applied.

DF-Fuzz unifies intra-procedural path constraints and inter-procedural, cross-file data dependencies within an offline-constructed Dual-Flow Hybrid Dependency Graph (DF-HDG). This design replaces runtime multi-round querying with single-round, serialized pre-injection, preserving high fuzzing throughput. Utilizing a dual-track navigation engine, DF-Fuzz combines Abstract Syntax Tree (AST) parsing with Language Server Protocol (LSP) indexing to resolve external definitions, reconstructing self-contained mock scopes via automated namespace wrapping without requiring a full project build. To maintain prompt compactness, DF-Fuzz applies Jaccard-similarity-based semantic pruning alongside API co-occurrence few-shot retrieval. To mitigate over-pruning, DF-Fuzz implements an adaptive feedback-driven self-healing loop that extracts compiler diagnostic errors locally and assigns infinite exemption weights to missing symbols, automatically restoring essential context in subsequent attempts at zero LLM interaction cost (Dou et al., 2024).

This paper makes the following primary contributions:

(1) Dual-Flow Dependency Formulation: We formulate the DF-HDG, a graph model capturing control-flow reachability constraints and cross-file data and inheritance dependencies.

Table 1: Performance and stability metrics of multi-round interactive RAG in compiler fuzzing.

| Evaluation Metric | Measured Value |
|---|---|
| Average query rounds for cross-file multi-layer dependencies | $\geq 3$ rounds |
| Average token consumption per single interaction | 4.2k $\sim$ 8.1k tokens |
| Average latency per single round of external RPC | 15.4 seconds |
| Session crash or deadlock rate (interactions over 3 rounds) | 22.0% (format deviation or truncation) |

(2) Dependency-Guided Context Retrieval: We design an extraction engine performing backward dominator slicing and global symbol relocation with automated namespace wrapping to synthesize scope-complete mock declarations.

(3) Context Optimization and Adaptive Self-Healing: We implement semantic pruning, API co-occurrence exemplar retrieval, and a local compiler-feedback-driven self-healing loop to dynamically recover omitted dependencies at zero multi-turn LLM cost.

(4) Empirical Evaluation: We evaluate DF-Fuzz across eight real-world language processors. DF-Fuzz achieves substantial coverage gains (+78.8% on Clang++ vs AFL++, +5.7% vs HLPFuzz) while reducing API costs by 60.3%, uncovering 45 previously unknown vulnerabilities (18 memory corruptions).

## 2 Motivation & Pilot Study

### 2.1 Limitation Analysis of Isolated Enclosing Function Slicing

To investigate how isolated physical function slicing hinders compiler fuzzing, consider a representative pattern from Clang's AST serializer, `WriteBlockInfoBlock()`:

```
void ASTWriter::WriteBlockInfoBlock() {
  RecordData Record; Stream.EnterSubblock(bitc::BLOCKINFO_BLOCK_ID, 3);
  if (unsigned NumAbbrevs = BLOCKINFO_ABBREV_LIMIT) {
    for (unsigned i = 0; i != NumAbbrevs; ++i)
      if (Stream.EmitBlockInfoAbbrev(bitc::RECORD_ABBREV_ID, ...)) { /* Bottleneck branch */
          }
  }
}
```

In this function, `RecordData` is a template defined in an external header; `Stream` is a private member variable of `ASTWriter`; and `BLOCKINFO_ABBREV_LIMIT` is an external macro. Under conventional function slicing, only the isolated body of `WriteBlockInfoBlock()` is provided to the LLM. Deprived of the method signature of `Stream`, macro values, and type declarations, the LLM hallucinates member accesses. In our pilot study on Clang, 90% of test programs synthesized under this semantic-scarce setting failed compilation due to undeclared identifiers.

### 2.2 Empirical Measurement of Interactive Query Overhead

To quantify the overhead of interactive multi-round RAG (where the LLM dynamically requests missing definitions via {`DEFINITION_LOOKUP`} commands), we evaluated 100 randomly sampled bottleneck basic blocks in Clang (v22.1.1) using GPT-5. As shown in Table 1, interactive RAG suffers from severe token inflation (4.2k–8.1k tokens/interaction), high latency (15.4s/round), and frequent session deadlocks (22.0% crash rate), motivating offline graph construction with single-round pre-injection.

These empirical measurements highlight three severe bottlenecks of interactive RAG in fuzzing: *Token Cost Inflation* (resolving a single branch frequently consumes tens of thousands of tokens); *Throughput Degradation* (RPC round-trips averaging 15.4s severely throttle execution speed); and *Session Fragility* (22.0% of multi-turn sessions experience deadlocks due to format hallucinations (Ji et al., 2023)). These findings motivate an offline dependency graph construction with single-round pre-injection.

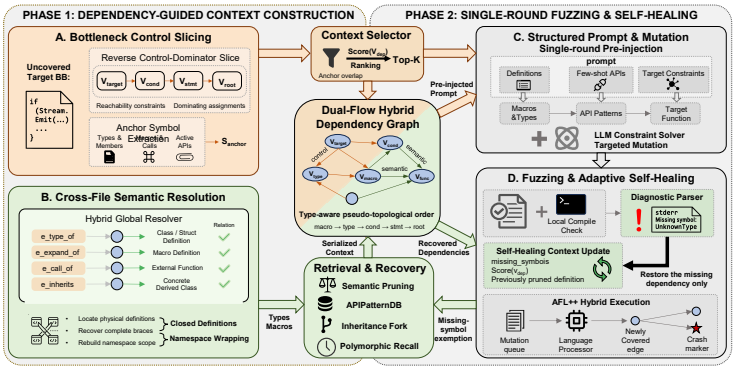


Figure 1: Overview of the DF-Fuzz architecture. Phase 1 constructs dependency-guided context via bottleneck control slicing, inter-procedural resolution, namespace wrapping, semantic pruning, and topological serialization. Phase 2 executes single-round mutation and local compiler-feedback-driven self-healing.

## 3 METHODOLOGY

DF-Fuzz employs a dual-track architecture that integrates local dominator slicing with global semantic resolution, augmented by a compiler-guided dynamic self-healing loop. Figure 1 illustrates the overall system workflow, structured into two core phases: Dependency-Guided Context Construction (Phase 1) and Single-Round Fuzzing & Adaptive Self-Healing (Phase 2).

### 3.1 FORMAL MODELING OF DUAL-FLOW HYBRID DEPENDENCY GRAPH (DF-HDG)

To capture both control-flow path constraints and cross-file symbol relationships, DF-Fuzz models program dependencies as a multi-relational directed graph $G = (V, E_C \cup E_D)$.

**Vertex Space** $V$. The node set $V$ defines entities involved in control and data tracking: $v_{target} \in V$ represents the uncovered target basic block or bottleneck control branch; $v_{cond} \in V$ denotes control predicate nodes dominating $v_{target}$ (e.g., `if (x > 0)`); $v_{root} \in V$ is the entry node of the enclosing function; $v_{type} \in V$ encompasses type declaration entities (structs, classes, unions, and enums); $v_{macro} \in V$ represents preprocessor macro definition entities (e.g., `#define BLOCKINFO_ABBREV_LIMIT 10`); $v_{func} \in V$ includes external auxiliary function declarations; and $v_{stmt} \in V$ denotes statement-level variable declarations, initializations, and assignments.

**Control Dependency Edge Space** $E_C$. Directed edges $(v_i, v_j) \in E_C$ indicate that execution of statement $v_i$ conditionally depends on the boolean outcome of predicate $v_j$. Control-flow analysis establishes the intra-procedural reachability chain $v_{target} \rightarrow v_{cond} \rightarrow v_{root}$.

**Data Dependency Edge Space** $E_D$. Data dependency edges capture cross-file and inter-procedural def-use mappings: (1) *Local Assignment Tracking* ($e_{comes_from}$) directed from a variable use to its reaching assignment $v_{stmt}$; (2) *Type Association* ($e_{type_of}$) directed from a variable to its type entity $v_{type}$; (3) *Function Invocation* ($e_{call_of}$) directed from a call site to callee $v_{func}$; (4) *Macro Expansion* ($e_{expand_of}$) directed from a macro reference to originating node $v_{macro}$; and (5) *Polymorphic Inheritance* ($e_{inherits}$) directed from a base class entity to its derived concrete subclass entities $v_{type_derived}$.

### 3.2 DUAL-TRACK LOCAL CONTROL SLICING AND GLOBAL DATA-FLOW TRACKING

**Intra-Procedural Backward Control Slicing:** Starting from an uncovered target branch $v_{target}$, DF-Fuzz computes the reverse dominator tree within the enclosing function $v_{root}$. By traversing control edges $(v_{target}, v_{cond}) \in E_C$, it isolates the minimal sequence of branch predicates governing target reachability. For all variables referenced in these predicates, $e_{comes_from}$ edges are traversed backward to extract their reaching initialization statements, producing a compact intra-procedural constraint slice.

**Hybrid Inter-Procedural Semantic Resolution:** To supply missing external definitions for symbols referenced within the local slice, DF-Fuzz executes a two-stage resolution: (1) *Global Symbol Relocation*: It extracts referenced type identifiers, template arguments, and macro names, querying a global index to obtain source coordinates $(File, Line, Col)$. (2) *Local Boundary Extraction & Namespace Wrapping*: Using local AST parsing, DF-Fuzz determines syntactic definition boundaries. To prevent scope errors in C++, the engine traverses the AST parent chain to identify enclosing namespace declarations (e.g., `namespace clang { namespace serialization { ... } }`) and wraps the extracted definition inside identical namespace blocks, generating syntactically self-contained mock declarations without full project headers.

### 3.3 Data-Flow Topological Sorting and Graph Serialization

To present prompt context in an order aligned with compiler parsing semantics and LLM autoregressive attention, DF-Fuzz serializes the DF-HDG into structured text using type-aware topological sorting. Graph edges are first inverted (from definition to use) to apply Kahn's algorithm. For nodes with equal in-degrees, DF-Fuzz applies the strict priority: Priority$(v_{macro}) >$ Priority$(v_{type}) >$ Priority$(v_{cond}) >$ Priority$(v_{stmt}) >$ Priority$(v_{root})$. This ordering places global macros and types at the top as declarations, intermediate predicates in the middle, and the target function $v_{root}$ requiring mutation at the end.

### 3.4 Semantic Pruning and Minimal Necessary Knowledge Filtering

Unrestricted traversal of complex C++ dependency graphs can retrieve hundreds of transitive dependencies, causing severe context bloat. We define the anchor symbol set $S_{anchor}$ as the union of all symbols appearing in enclosing function $v_{root}$ and control predicates $v_{cond}$. For each candidate dependency node $v_{dep}$ identified via BFS, DF-Fuzz computes its Jaccard intersection score with the anchor set: Score$(v_{dep}) = |S_{dep} \cap S_{anchor}|$. Candidate dependency nodes are ranked in descending order of Score$(v_{dep})$, retaining only the top-$K$ highest-scoring nodes (default $K = 6$). Lower-scoring transitive nodes are pruned, bounding prompt token consumption within an efficient operating range.

### 3.5 API Co-occurrence Exemplar Retrieval and Polymorphic Recall

**API Exemplar Retrieval:** Providing class definitions alone is often insufficient for LLMs to infer correct API usage patterns. During an offline pre-processing phase, DF-Fuzz analyzes existing test suites and source files to extract recurring clusters of co-occurring API symbols (e.g., `ASTContext`, `SourceManager`, and `DiagnosticBuilder` within a single function scope). These concise usage snippets ($\leq 50$ lines) are indexed in APIPatternDB using composite hash keys (e.g., `ASTContext+SourceManager`). During branch solving, the system extracts active external classes from path conditions to query APIPatternDB, injecting the top 2–3 matching usage examples as few-shot exemplars (Wei et al., 2022; Nashid et al., 2023).

**Polymorphic Recall:** In object-oriented codebases, critical interfaces are frequently defined as abstract base classes containing pure virtual methods. When a retrieved candidate node $v_{dep}$ represents an abstract class or possesses only protected constructors, DF-Fuzz traverses $e_{inherits}$ edges in the DF-HDG. The global relocation engine identifies concrete derived subclasses implementing the interface and unconditionally injects the top 1–2 most frequently used concrete subclass definitions into the prompt context, allowing the LLM to construct valid concrete objects.

### 3.6 Adaptive Feedback-Driven Self-Healing Loop

Static similarity-based pruning may occasionally discard implicit dependencies (e.g., implicit conversion operators or indirect macro dependencies), leading to compilation errors. DF-Fuzz introduces a local compiler-guided self-healing loop to dynamically recover missing context without multi-turn LLM querying (Dou et al., 2024): *(1) Diagnostic Error Extraction:* Upon receiving a generated test candidate from the LLM, DF-Fuzz executes a local fast syntax verification check (e.g., Clang/GCC `-fsyntax-only`). If compilation fails, the self-healing module parses `stderr` using regular expressions to identify specific missing symbols (e.g., matching `error: use of undeclared identifier 'IdentifierName'`). This diagnostic parsing runs locally in

milliseconds at zero token cost. *(2) Exemption Weight Injection:* The identified missing symbols are added to the active exemption set (`missing_symbols`). In the subsequent RAG retrieval iteration for this target branch, the dependency scoring function assigns infinite weight to any candidate node $v_{dep}$ defining an exempt symbol: Score$(v_{dep}) = \infty$. This assignment guarantees that essential definitions erroneously omitted during initial pruning are unconditionally restored and injected into the prompt in the next mutation attempt.

**End-to-End Walkthrough:** (1) The fuzzer targets the branch condition in `ASTWriter::WriteBlockInfoBlock();` (2) Setting $K = 6$, the macro `BLOCKINFO_ABBREV_LIMIT` is pruned due to low Jaccard similarity; (3) The LLM generates test code referencing the macro, triggering the local compiler error: `error: use of undeclared identifier 'BLOCKINFO_ABBREV_LIMIT'`; (4) The error parser extracts `BLOCKINFO_ABBREV_LIMIT` into `missing_symbols`, setting its node score to $\infty$; (5) In Round 2, the definition `#define BLOCKINFO_ABBREV_LIMIT 10` is unconditionally injected. The regenerated test case compiles cleanly and successfully exercises the target bottleneck branch.

# 4 System Implementation

DF-Fuzz is built as a modular pipeline combining Tree-sitter (Brunsfeld & Others, 2023) for AST parsing/namespace reconstruction, Bear (Nagy, 2024) for compilation databases, and a persistent JSON-RPC bridge with Clangd (Project, 2024) LSP for inter-procedural symbol indexing. API patterns are stored in APIPatternDB. The runtime incorporates key optimizations: (1) Single-round pre-injection replaces dynamic queries; (2) Hybrid scheduling invokes asynchronous LLM solving only when AFL++ execution stagnates (Yang et al., 2025b; Lemieux et al., 2023); (3) Automatic parallel regex fallback is supported if LSP build databases are unavailable; and (4) AFL++ runs with `-m none -t 3000` to prevent memory timeouts.

# 5 Evaluation

Our evaluation addresses six research questions:

**RQ1 (Coverage & Bugs)**: Coverage and defect discovery gains (Section 5.2);

**RQ2 (Vulnerabilities)**: Real-world vulnerability characteristics (Section 5.3);

**RQ3 (Ablation)**: Contributions of individual algorithmic components (Section 5.4.1);

**RQ4 (Cost & Speed)**: Token cost reduction and fuzzing throughput (Section 5.4.2);

**RQ5 (Sensitivity to $K$)**: Impact of context retention threshold $K$ (Section 5.4.2);

**RQ6 (Generalization)**: Generalizability across diverse LLM backends (Section 5.4.2).

## 5.1 Experimental Setup

**Environment and Protocol.** All experiments were executed on a dedicated Ubuntu 22.04.5 LTS server equipped with dual Intel Xeon Platinum 8462Y+ processors (64 physical cores / 128 threads) and 1,024 GB RAM. Each fuzzing instance ran inside an isolated Docker container pinned to dedicated CPU cores. The default LLM backend was OpenAI GPT-5 (OpenAI, 2023). All baseline fuzzers were initialized with identical seed corpora. Following standard benchmarking practices (Metzman et al., 2021; Hazimeh et al., 2021), each trial for RQ1, RQ3, RQ4, RQ5, and RQ6 was conducted for 24 hours across 5 independent runs ($N = 5$). We report the mean, standard deviation (SD), and 95% confidence intervals (CI). For RQ2, we conducted continuous testing across eight targets over a one-month testing period.

**Evaluation Subjects.** We selected eight widely used language processors spanning six programming languages: Clang (C, v22.1.1), Clang++ (C++, v22.1.1), Hermes (JavaScript, v0.13.0), ChakraCore (JavaScript, v97eu8), njs (JavaScript, v0.9.6), Flang (Fortran, v22.1.1), Lua (Lua, v5.5.0), and solc (Solidity, v0.8.34). For 24-hour comparative evaluations and ablation studies, four primary targets (Clang++, Flang, Hermes, and Lua) were selected to represent diverse language paradigms.

Table 2: Comparison of edge coverage and unique bugs detected over 24-hour runs (mean values, $N = 5$).

| Tool/System | Clang++ (Cov) | Clang++ (Bug) | Flang (Cov) | Flang (Bug) | Hermes (Cov) | Hermes (Bug) | Lua (Cov) | Lua (Bug) |
|---|---|---|---|---|---|---|---|---|
| SymQEMU | 23,148 | 0 | 44,872 | 0 | 5,761 | 0 | 1,118 | 0 |
| REDQUEEN | 24,936 | 0 | 51,684 | 0 | 8,437 | 0 | 1,463 | 0 |
| POLYGLOT | 44,587 | 1 | 74,916 | 1 | 16,428 | 0 | 1,592 | 0 |
| Grammarinator | 29,274 | 0 | 35,183 | 0 | 9,764 | 0 | 978 | 0 |
| YARPGen | 35,412 | 0 | – | – | – | – | – | – |
| Fuzzilli | – | – | – | – | 17,186 | 0 | – | – |
| AFL++ | 56,029 | 1 | 69,843 | 1 | 13,947 | 0 | 3,395 | 0 |
| HLPFuzz | 94,782 | 4 | 116,843 | 3 | 24,917 | 1 | 4,258 | 0 |
| **DF-Fuzz (Ours)** | **100,153** | **6** | **121,239** | **4** | **26,571** | **1** | **4,897** | **0** |

**Baseline Tools.** We compared DF-Fuzz against eight representative fuzzing systems across different paradigms: AFL++ (Fioraldi et al., 2020), SymQEMU (Poeplau & Francillon, 2021), REDQUEEN (Aschermann et al., 2019b), POLYGLOT (Chen et al., 2021), Grammarinator (Hodován et al., 2018), YARPGen (Livinskii et al., 2020), Fuzzilli (Groß et al., 2023), and HLPFuzz (Yang et al., 2025b).

### 5.2 Comparison with Existing Tools (RQ1)

Table 2 presents the edge coverage and unique bug counts achieved by each tool across 24-hour fuzzing campaigns ($N = 5$).

#### 5.2.1 Coverage

As detailed in Table 2, DF-Fuzz consistently outperforms all baseline fuzzers across every evaluation target:

*Static Compilers (Clang++, Flang):* On Clang++, characterized by intricate template metaprogramming, complex overload resolution, and strict type constraints, DF-Fuzz achieves a final mean edge coverage of $100.2\text{k} \pm 3.1\text{k}$ (95% CI: $[97.5\text{k}, 102.9\text{k}]$). This represents a 78.8% improvement over AFL++ ($56.0\text{k} \pm 2.1\text{k}$) and a 5.7% improvement over the state-of-the-art LLM baseline HLPFuzz ($94.8\text{k} \pm 3.0\text{k}$). On Flang, DF-Fuzz achieves $121.2\text{k} \pm 3.8\text{k}$ coverage (95% CI: $[117.9\text{k}, 124.5\text{k}]$), outperforming HLPFuzz ($116.8\text{k} \pm 3.9\text{k}$) by 3.8% and nearly doubling the coverage of AFL++ ($69.8\text{k} \pm 2.5\text{k}$, +73.6%). *Dynamic Engines (Hermes, Lua):* On Hermes, DF-Fuzz reaches a mean coverage of $26.6\text{k} \pm 0.8\text{k}$ (95% CI: $[25.9\text{k}, 27.3\text{k}]$), exceeding Fuzzilli (17.2k, +54.6%) and AFL++ (13.9k, +90.5%), while outperforming HLPFuzz (24.9k, +6.6%). On Lua, DF-Fuzz attains $4.9\text{k} \pm 0.15\text{k}$ coverage (95% CI: $[4.75\text{k}, 5.05\text{k}]$), surpassing AFL++ (3.4k, +44.2%) and HLPFuzz (4.3k, +15.0%).

These empirical results demonstrate that traditional grammar-based and greybox fuzzers frequently stall at early frontend syntactic checks due to random mutation blindness. While HLPFuzz utilizes LLM reasoning to generate constraint-solving inputs, the absence of global dependency retrieval causes many generated test cases to contain unresolved symbols or undeclared types, leading to compiler rejection. By reconstructing self-contained semantic contexts via the DF-HDG and repairing omissions through local self-healing, DF-Fuzz drastically increases the compilation pass rate, enabling continuous exploration into deeper compiler stages.

#### 5.2.2 Bug Discovery

The edge coverage breakthroughs of DF-Fuzz directly translate into superior defect detection capability in 24-hour evaluation trials: on Clang++, DF-Fuzz discovers 6 unique bugs (vs. 4 by HLPFuzz and 1 by AFL++); on Flang, DF-Fuzz detects 4 unique bugs (vs. 3 by HLPFuzz and 1 by POLYGLOT/AFL++); on Hermes, DF-Fuzz and HLPFuzz each uncover 1 unique crash. In total, DF-Fuzz uncovers 11 unique bugs across the four primary targets, outperforming HLPFuzz (8 bugs) and finding over five times as many bugs as AFL++ (2 bugs), confirming that DF-Fuzz penetrates deep compiler logic where critical bugs reside.

Table 3: Distribution of unique vulnerabilities discovered by DF-Fuzz across eight targets.

| Target Subject | AF | SV | SOF | ND | HBOF | HUAF | FPE | SUAS | Total Vulnerabilities |
|---|---|---|---|---|---|---|---|---|---|
| Clang++ | 0 | 1 | 0 | 1 | 6 | 7 | 1 | 1 | 17 |
| Clang | 0 | 2 | 0 | 1 | 1 | 2 | 1 | 1 | 8 |
| Flang | 10 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 12 |
| Hermes | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| ChakraCore | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 |
| njs | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| Lua | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| solc | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| **Total** | **11** | **8** | **0** | **4** | **8** | **10** | **2** | **2** | **45** |

**Note**: AF: Assertion Failure; SV: Seg Fault; SOF: Stack Overflow; ND: Null Deref; HBOF: Heap Buffer Overflow; HUAF: Heap UAF; FPE: Floating Point Exception; SUAS: Stack Use-After-Scope.

Table 4: Ablation study results comparing DF-Fuzz variants across 24-hour runs (mean values, $N = 5$).

| Variant Configuration | Clang++ (Cov) | Clang++ (Bug) | Flang (Cov) | Flang (Bug) | Hermes (Cov) | Hermes (Bug) | Lua (Cov) | Lua (Bug) |
|---|---|---|---|---|---|---|---|---|
| **DF-Fuzz (Full System)** | **100,153** | **6** | **121,239** | **4** | **26,571** | **1** | **4,897** | **0** |
| ↪ *w/o Semantic Pruning* | 95,824 | 4 | 117,950 | 3 | 25,180 | 1 | 4,410 | 0 |
| ↪ *w/o API Few-shot* | 98,140 | 5 | 119,930 | 4 | 26,050 | 1 | 4,720 | 0 |
| ↪ *w/o Multi-relational Recall* | 97,560 | 5 | 118,940 | 3 | 25,820 | 1 | 4,680 | 0 |
| ↪ *w/o Self-Healing Feedback* | 96,210 | 4 | 117,820 | 3 | 25,340 | 1 | 4,490 | 0 |
| ↪ *w/o RAG (Enclosing Slice)* | 94,810 | 4 | 116,890 | 3 | 24,930 | 1 | 4,260 | 0 |

### 5.3 Real-World Vulnerability Discovery (RQ2)

In extended testing across eight open-source targets, DF-Fuzz uncovered 45 unique vulnerabilities (Table 3).

Among the 45 detected vulnerabilities, 10 are heap use-after-free (HUAF) and 8 are heap buffer overflow (HBOF) bugs. In Clang++, 7 HUAF vulnerabilities were triggered in AST restructuring passes within middle-end optimization pipelines (e.g., loop vectorization and dead code elimination) involving dangling pointer dereferences (Livinskii et al., 2023). Exercising these deep optimizer paths requires syntactically valid and semantically rich test programs that pass all prior frontend static checks. In Flang, DF-Fuzz uncovered 10 assertion failures deep within Fortran generic type resolution modules. Because Flang relies extensively on complex class hierarchies and template abstractions, the polymorphic recall mechanism ($e_{inherits}$) enabled the generation of concrete derived-type instantiations that successfully exercised deeply nested type-checking assertions.

### 5.4 Analytical Experiments

#### 5.4.1 Ablation Study (RQ3)

To evaluate individual design components, we evaluated five DF-Fuzz variants across 24-hour runs (Table 4).

**Coverage.** As shown in Table 4, disabling individual components consistently degrades edge coverage: disabling semantic pruning causes Clang++ coverage to drop to 95,824 (-4.3%) due to long-context prompt noise; disabling self-healing feedback reduces Clang++ coverage to 96,210 (-3.9%) because essential implicit symbols pruned by static scoring cannot be recovered; removing API few-shot exemplars and polymorphic recall leads to moderate coverage drops (Clang++ drops to 98,140 and 97,560); and completely removing RAG causes coverage to degrade to levels near native HLPFuzz (e.g., Lua dropping to 4,260).

**Bug Discovery.** The defect detection capability mirrors the coverage trends: the full DF-Fuzz system detects 11 unique bugs across all targets. Removing semantic pruning or self-healing feedback reduces bugs in Clang++ from 6 to 4, and in Flang from 4 to 3. The variant without RAG discovers only 4 bugs in Clang++ and 3 in Flang, confirming that isolated function slicing lacks external context required for deep defect triggers.

Table 5: Throughput, API costs, pruning sensitivity ($K$), and backend generalization on Clang++ (24h).

| (a) Throughput & Cost | | | (b) Pruning Retention $K$ vs Coverage | | | | | (c) LLM Backend Generalization | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Fuzzer** | **Throughput** | **Cost/h** | $K=2$ | $K=4$ | $K=6$ **(Def)** | $K=8$ | $K=10$ | **Backend** | **Coverage** | **Bugs** |
| AFL++ | 177.8/s | $0.00 | 92.1k | 96.9k | **100.2k** | 98.9k | 95.4k | Qwen3.5-Flash | 89,451 | 3 |
| HLPFuzz | 44.2/s | $0.58 | – | – | – | – | – | Qwen3.5-27B | 95,182 | 4 |
| **DF-Fuzz** | **56.1/s** | **$0.23** | – | – | – | – | – | **GPT-5 (Def)** | **100,153** | **6** |

### 5.4.2 Cost, Throughput, Sensitivity, and Generalization (RQ4–RQ6)

**Cost & Throughput Efficiency (RQ4):** Pre-building dependency graphs offline reduces hourly API operational costs to $0.23 (-60.3% compared to HLPFuzz's $0.58/h), while eliminating multi-turn RPC communication delays increases test execution throughput to 56.1 execs/sec (+26.9% over HLPFuzz's 44.2 execs/sec).

**Pruning Retention Sensitivity (RQ5):** Setting $K = 6$ achieves peak edge coverage (100,153). Lower thresholds ($K \leq 4$) suffer from missing external context, while higher thresholds ($K \geq 8$) introduce redundant definitions that dilute the LLM's attention.

**Model-Agnostic Generalization (RQ6):** When driven by lightweight `Qwen3.5-Flash` and open-source `Qwen3.5-27B`, DF-Fuzz sustains substantial coverage gains over AFL++ (+59.7% and +69.9%), confirming the framework's broad applicability across proprietary and open-source model backends.

## 6 Related Work

**Language Processor Fuzzing:** Generative grammar fuzzers (Grammarinator (Hodován et al., 2018), YARPGen (Livinskii et al., 2020), NAUTILUS (Aschermann et al., 2019a)) generate syntactically valid inputs but lack target guidance to penetrate deep optimization paths. Coverage-guided greybox fuzzers (AFL++ (Fioraldi et al., 2020), LibAFL (Fioraldi et al., 2022), CollAFL (Gan et al., 2018), MOPT (Lyu et al., 2019), POLYGLOT (Chen et al., 2021)) lack semantic type awareness, stalling at compiler frontends (Padhye et al., 2019; Lemieux & Sen, 2018). LLM-guided fuzzers (HLPFuzz (Yang et al., 2025b), WhiteFox (Yang et al., 2024), Clozemaster (Gao et al., 2025), Fuzz4All (Xia et al., 2024), Fuzzilli (Groß et al., 2023), TitanFuzz (Deng et al., 2023), FuzzGPT (Deng et al., 2024), KernelGPT (Yang et al., 2025a), ELFuzz (Chen et al., 2025)) leverage LLM reasoning but suffer from semantic scarcity in isolated slices. DF-Fuzz addresses this by dependency-guided context retrieval and self-healing.

**RAG in Software Engineering:** RAG and context graph modeling have been explored in repository completion (RepoCoder (Shrivastava et al., 2023)), agentic workflows (RepoAgent (Chen et al., 2024)), cross-file generation (RepoGen (Shen et al., 2024)), exemplar selection (Nashid et al., 2023), representation learning (Galla (Zhang et al., 2025b)), fuzz driver synthesis (CKGFuzzer (Xu et al., 2025)), and syscall retrieval (Zhang et al., 2025a). Unlike interactive multi-round developer RAG, DF-Fuzz delivers high-throughput single-round pre-injection, scope-complete namespace wrapping, and zero-token local error self-healing.

## 7 Conclusion

We presented DF-Fuzz, a dependency-guided context retrieval framework for language processor fuzzing. DF-Fuzz constructs a Dual-Flow Hybrid Dependency Graph (DF-HDG) to unify control-flow path constraints and cross-file semantic dependencies. By combining offline graph construction, namespace wrapping, semantic pruning, and a compiler-feedback-driven self-healing loop, DF-Fuzz provides self-contained context in single-round prompts. Evaluations on eight production language processors demonstrate that DF-Fuzz achieves significant coverage improvements over AFL++ and HLPFuzz, reduces API operational costs by 60.3%, and detected 45 previously unknown real-world vulnerabilities.

**AI Use Statement** In this work, we used generative AI assistance for grammar checking and text polishing of the manuscript. We have not used generative AI for core scientific ideation, algorithm design, or data collection. All results were strictly verified by the authors, who take full responsibility for this work.

**Ethics Statement** This study complies with all ethical standards. All vulnerability discoveries in open-source language processors were responsibly disclosed to respective security teams following coordinated disclosure protocols. No proprietary or sensitive personal data was collected.

**Reproducibility Statement** All experimental artifacts, including graph construction toolchains, LSP bridges, seed corpora, and evaluation harnesses, are available in supplementary materials and detailed in Appendix A. Statistical breakdowns are provided in Section 5.2 and Appendix D.

## REFERENCES

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019a.

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019b.

Max Brunsfeld and Others. Tree-sitter: An incremental parsing system for programming tools. *Software Manual*, 2023.

Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. ELFuzz: Efficient input generation via LLM-driven synthesis over fuzzer space. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*, pp. 6279–6298, 2025.

Qinyu Chen, Jidong Ge, Chuanyi Li, and Vincent Ng. RepoAgent: An llm-powered agent for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.

Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yueh-Hsun Lin, and Wenke Lee. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 1894–1911, 2021.

Yi Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 423–435, 2023.

Yi Deng, Chunqiu Steven Xia, Chenyuan Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1–13, 2024.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, and Others. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4571–4585, 2024.

Andrea Fioraldi, Dominik Maier, Heiko Eissfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1051–1065, 2022.

Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuetao Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*, pp. 679–696, 2018.

Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. Clozemaster: Fuzzing rust compiler by harnessing llms for infilling masked real programs. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1422–1435, 2025.

Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for javascript jit compiler vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *Proceedings of the 2021 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 647–657, 2021.

Renáta Hodován, Ákos Kiss, and Tamás Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST)*, pp. 45–48, 2018.

Jiawei Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 491–503, 2023.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys (CSUR)*, 55(12):1–38, 2023.

Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pp. 475–485, 2018.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CODA-MOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 919–931, 2023.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing of compilers with YarpGen. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 1–25, 2020.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 1826–1847, 2023.

Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pp. 1949–1966, 2019.

Haoyang Ma. A survey of modern compiler fuzzing. *ACM Computing Surveys (CSUR)*, 2023.

Valentin J. M. Manès, HyungSeok Han, Cheolwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 47(11):2312–2331, 2019.

Jonathan Metzman, László Szekeres, Laurent Simon, Terry Read, and Abhishek Arya. FuzzBench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1656–1660, 2021.

László Nagy. Bear: Build ear. `https://github.com/rizsotto/Bear`, 2024.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 2450–2462, 2023.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 329–340, 2019.

Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymQEMU: Compilation-based qemu execution for fast symbolic emulation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 727–741, 2021.

LLVM Project. Clangd: language server that understands C++. `https://clangd.llvm.org/`, 2024.

Da Shen, Xinyi Hou, Yanjie Zhao, and Haoyu Wang. Cross-file code completion via retrieval-augmented generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024.

Disha Shrivastava, Hailey Schoelkopf, Yuanzhi Li, Jerry Wei, Sheng Shen, Baolin Peng, Vincent Zhao, Sanjeev Satheesh, and Colin Raffel. Repository-level prompt generation for large language models of code. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 31693–31715, 2023.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

Junming Wang, Yuchao Huang, Chunrong Chen, and Others. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering (TSE)*, 50 (4):911–936, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 24824–24837, 2022.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1–13, 2024.

Hongliang Xu, Wanlun Ma, Tao Zhou, and Yang Xiang. CKGFuzzer: LLM-based fuzz driver generation enhanced by code knowledge graph. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 243–254, 2025.

Chenyuan Yang, Yi Deng, Runbu Lu, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 8(OOPSLA2):709–735, 2024.

Chenyuan Yang, Zheng Zhao, and Lingming Zhang. KernelGPT: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 560–573, 2025a.

Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. Hybrid language processor fuzzing via LLM-based constraint solving. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*, 2025b.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pp. 745–761, 2018.

Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. Unlocking low frequency syscalls in kernel fuzzing with dependency-based RAG. *Proceedings of the ACM on Software Engineering (ISSTA)*, 2(ISSTA):848–870, 2025a.

Ziyin Zhang, Hang Yu, Sage Lee, Peng Di, Jianguo Li, and Rui Wang. Galla: Graph aligned large language models for improved source code understanding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 13784–13802, 2025b.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

## A Detailed Reproducibility and Artifact Setup

To ensure the scientific reproducibility and objectivity of our experimental evaluations, we implemented rigorous environment and configuration management protocols across all experiments:

- *Containerized Execution and Core Pinning:* All fuzzing campaigns and baseline comparative runs were deployed within dedicated, isolated Docker containers running Ubuntu 22.04.5 LTS. To eliminate performance interference, thread contention, and cache pollution arising from concurrent execution on multi-core servers, each containerized fuzzing instance was strictly pinned to dedicated physical CPU cores with isolated memory allocations.
- *Standardized Benchmarks and Identical Corpora:* All evaluated language processor targets were built using identical commit versions and configuration flags (as cataloged in Section 5.1). For all baseline tools requiring initial seed inputs, we uniformly supplied identical initial test corpora harvested from official test suites and open-source benchmark repositories, ensuring an equitable baseline across all evaluated paradigms.
- *Statistical Rigor across Repeated Trials:* To eliminate the confounding effects of fuzzing non-determinism and random seed variance, all 24-hour comparative evaluations and ablation experiments were independently repeated across 5 distinct runs with different pseudorandom seeds ($N = 5$). We report both the sample mean, standard deviation (SD), and 95% confidence intervals (CI) in accordance with the established empirical guidelines of FuzzBench (Metzman et al., 2021).
- *Open-Source Artifact Availability:* To facilitate independent replication and community verification, the complete source code of the DF-Fuzz dependency graph construction engine (including the Tree-sitter AST parser, Clangd LSP JSON-RPC bridge, and graph serialization scripts), experimental configurations, seed corpora, and evaluation log parsers will be made publicly available in an open-access replication package upon publication.

## B Extended System Limitations

While DF-Fuzz demonstrates substantial empirical advantages over existing fuzzers, we identify several boundary conditions and limitations that provide avenues for future exploration:

- *Commercial API Dependency and Cost Constraints:* By default, DF-Fuzz utilizes commercial large language model backends (OpenAI GPT-5) for semantic mutation and constraint solving. Although DF-Fuzz reduces API operational costs by 60.3% compared to interactive RAG baselines, reliance on commercial cloud APIs remains subject to network latency, rate limits, and monetary costs. While our RQ6 experiments confirm that open-source models (such as Qwen3.5-27B) achieve competitive coverage, deploying quantized local models optimized specifically for code synthesis represents an important direction to enable fully offline, air-gapped fuzzing deployments.
- *Polymorphic Search Breadth under Complex Class Hierarchies:* In large object-oriented C++ codebases with deeply nested class inheritance structures (e.g., Clang's AST hierarchy containing hundreds of derived node classes), the current polymorphic recall mechanism ($e_{inherits}$) relies on global LSP indexing coupled with top-$K$ pruning. Under constrained prompt budgets ($K = 6$), the system can only retrieve the top 1–2 concrete subclasses. When a target branch requires an uncommon derived type, the necessary subclass may be omitted during pruning. Future enhancements could incorporate static type-hierarchy frequency scoring or def-use reachability analysis to dynamically rank candidate derived classes.
- *Sequential Self-Healing on Multi-Error Candidates:* The current diagnostic error extractor captures and parses the primary compiler error emitted during local syntax checking (e.g., a missing macro identifier), resolving one missing symbol per iteration. In scenarios where an over-pruned test case simultaneously triggers multiple independent compilation errors (such as lacking both an external macro and an auxiliary struct definition), DF-Fuzz requires sequential repair rounds. Future work can construct a multi-error diagnostic dependency tree to parse and exempt multiple missing symbols concurrently, further accelerating the self-healing convergence rate.

## C Edge Coverage Growth Curves over 24 Hours

Figure 2 illustrates the cumulative edge coverage trajectories over 24-hour testing campaigns across Clang++, Flang, Hermes, and Lua ($N = 5$).

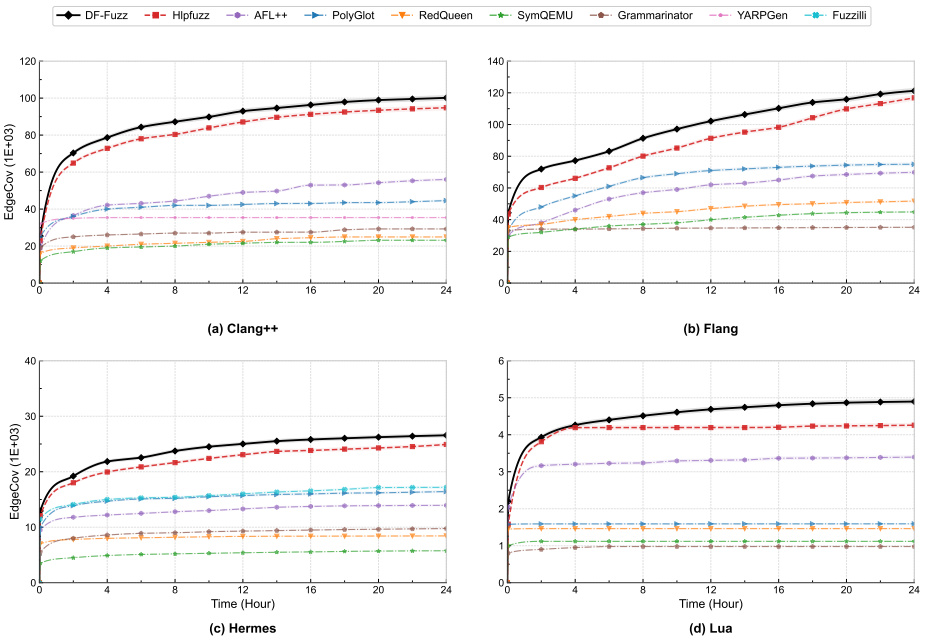


Figure 2: Edge coverage growth curves over 24 hours comparing DF-Fuzz against baseline fuzzing tools. Curves depict mean cumulative edge coverage across 5 independent runs. DF-Fuzz exhibits steep early growth and sustains upward progress throughout the 24-hour period, successfully overcoming coverage plateaus encountered by traditional fuzzers.

## D Coverage Statistics and Confidence Intervals

Table 6 summarizes the edge coverage mean, standard deviation (SD), and 95% confidence intervals (CI) for the primary evaluation targets ($N = 5$):

Table 6: Statistical breakdown of 24-hour edge coverage (mean, SD, and 95% CI).

| Target Subject | DF-Fuzz (Mean ± SD [95% CI]) | HLPFuzz (Mean ± SD [95% CI]) | AFL++ (Mean ± SD [95% CI]) |
|---|---|---|---|
| **Clang++** | 100.2k ± 3.1k [97.5k, 102.9k] | 94.8k ± 3.0k [92.2k, 97.4k] | 56.0k ± 2.1k [54.2k, 57.8k] |
| **Flang** | 121.2k ± 3.8k [117.9k, 124.5k] | 116.8k ± 3.9k [113.4k, 120.2k] | 69.8k ± 2.5k [67.6k, 72.0k] |
| **Hermes** | 26.6k ± 0.8k [25.9k, 27.3k] | 24.9k ± 0.9k [24.1k, 25.7k] | 13.9k ± 0.6k [13.4k, 14.4k] |
| **Lua** | 4.9k ± 0.15k [4.75k, 5.05k] | 4.3k ± 0.2k [4.13k, 4.47k] | 3.4k ± 0.1k [3.31k, 3.49k] |