OPTFUZZ: RETHINKING LLM-ASSISTED FUZZING FROM STATELESS GENERATION TO CLOSED-LOOP SEMANTIC OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) can generate syntactically valid test cases, but existing LLM-assisted fuzzers mostly treat them as stateless, single-shot generators. For critical infrastructure systems with deep semantic constraints—such as compilers, runtimes, and bytecode interpreters—this design lacks global search memory, architectural state tracking, and iterative error recovery, leaving exploration perpetually stuck on coverage plateaus. In this paper, we propose OPTFUZZ, a novel hybrid fuzzing framework that reframes LLM-assisted fuzzing into hierarchical closed-loop semantic optimization. OPTFUZZ preserves the high throughput of coverage-guided fuzzing (CGF) while selectively invoking the LLM when coverage stagnation is detected. We devise two collaborative optimization engines: the Breadth Optimizer estimates dynamic region values combining static potential metrics and historical solving feedback to steer global state-space exploration toward under-covered logic; the Depth Optimizer models hard bottleneck branch penetration as a contextual multi-armed bandit (LinUCB) sequential decision process over multimodal structural features, iteratively repairing and approximating branch constraints via compilation diagnostics and execution traces. Evaluated across 8 industrial-grade language processors (including Clang, Flang, Hermes, ChakraCore, and Lua), OPTFUZZ improves edge coverage by up to 115.9% over state-of-the-art fuzzers and discovers 74 real-world vulnerabilities (57 confirmed by upstream maintainers), proving the immense security value of closed-loop semantic optimization. Our code and artifacts are publicly available at `https://osf.io/5g2ez/overview?view_only=9a341081ce104a60ad8b3102a7ebb430`.

1 INTRODUCTION

Fuzzing is fundamentally a high-dimensional state space search problem subjected to rigorous syntactic and semantic constraints (Lanzi et al., 2007; He et al., 2024). Modern fuzzers mainly follow two paradigms. Generation-based fuzzers use expert-written grammars or templates to produce structured inputs (Hodován et al., 2018; Aschermann et al., 2019a), but manual grammar engineering rarely captures intricate semantic corner cases. Coverage-guided fuzzing (CGF), represented by AFL++ (Fioraldi et al., 2020) and LibAFL (Fioraldi et al., 2022), explores program state spaces via iterative random mutations. While CGF offers high throughput and broad applicability, it struggles on systems with deep control flows and complex semantic invariants—such as compilers, virtual machines, and bytecode runtimes (Zhu et al., 2025; Ma, 2023). Blind byte-level mutations destroy syntax validity, causing inputs to be rejected early at parser checks (Wang et al., 2024a), while symbolic execution suffers from severe path explosion on large codebases (Zhou et al., 2023; Padhye et al., 2019; Stephens et al., 2016).

Recently, Large Language Models (LLMs) have demonstrated emergent capabilities in code synthesis, internalizing programming language syntax rules and semantic idioms (Vaswani et al., 2017; Zhang et al., 2025b; Ding et al., 2024; Wang et al., 2024b). Current LLM-assisted fuzzing evolves along pure generative fuzzing (Xia et al., 2024; Deng et al., 2023; 2024) and hybrid fuzzing (Huang et al., 2025; Hu et al., 2023; Qiu et al., 2023; Yang et al., 2025a; Li et al., 2024; Yang et al., 2025b). However, existing systems suffer from three fundamental bottlenecks:

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.

1. *Throughput-Intelligence Conflict*: High inference latency and monetary cost restrict pure generative methods from matching the high throughput required by large-scale fuzzing campaigns (Dou et al., 2024).
2. *Stateless Single-Shot Generation*: In current hybrid architectures, LLMs operate as stateless generators lacking persistent search memory and diagnostic feedback, resulting in repeated generation of stagnant tests (Oliinyk et al., 2024; Meng et al., 2024; Kang et al., 2023).
3. *Context Limitations*: Physical token limits prevent injecting entire multi-million LOC codebases, leaving models unable to resolve deep cross-procedural constraints (Ou et al., 2025; Yu et al., 2024; Xu et al., 2025).

To overcome these barriers, we present OPTFUZZ, a hybrid fuzzing framework that converts LLM-assisted fuzzing into hierarchical closed-loop semantic optimization. OPTFUZZ retains high-throughput CGF locally while selectively invoking two collaborative LLM engines upon coverage stagnation: the *Breadth Optimizer* estimates dynamic regional values to steer exploration toward under-covered logic, and the *Depth Optimizer* models bottleneck branch solving as a contextual multi-armed bandit (LinUCB) (Li et al., 2010), iteratively repairing test cases via compilation diagnostics and execution traces.

Evaluated across 8 industrial language processors (Clang, Flang, Hermes, ChakraCore, Lua, solc), OPTFUZZ achieves up to 115.9% higher edge coverage than state-of-the-art fuzzers and uncovers 74 real-world zero-day vulnerabilities (57 confirmed by maintainers).

Contributions: (1) A hierarchical closed-loop optimization paradigm reconciling CGF throughput with selective semantic guidance; (2) A dual-optimizer mechanism featuring region-value breadth exploration and LinUCB depth constraint solving; (3) Extensive empirical validation across 8 industrial targets discovering 74 zero-day vulnerabilities.

2 BACKGROUND, THREAT MODEL, AND PROBLEM FORMULATION

2.1 COVERAGE GUIDED FUZZING AND LLM FUZZING

Coverage-guided fuzzers (Zhou et al., 2023; Fioraldi et al., 2020; 2022) guide seed evolution via coverage bitmaps. On complex language processors, blind mutations fail to generate valid programs, while symbolic execution (Driller (Stephens et al., 2016), QSYM (Yun et al., 2018), DigFuzz (Zhao et al., 2019)) encounters path explosion. Although LLMs generate structured seeds (Deng et al., 2023; Xia et al., 2024; Yang et al., 2024; Zhang et al., 2025a; Chen et al., 2025; Yang et al., 2025b), existing methods treat them as stateless generators lacking historical search context and error recovery.

2.2 PROBLEM FORMULATION & THREAT MODEL

Let $\mathcal{P}$ denote the Program Under Test (PUT), with Control Flow Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. An input $x \in \mathcal{X}$ is valid iff it satisfies formal syntax specification $\mathcal{V}_{syn}$ and semantic rules $\mathcal{V}_{sem}$: $\mathcal{X}_{valid} = \{x \in \mathcal{X} \mid x \models \mathcal{V}_{syn} \wedge x \models \mathcal{V}_{sem}\}$. An input x executes path $\pi(x) \subseteq \mathcal{E}$ with outcome $\mathcal{O}(x) \in \{\text{Normal}, \text{Error}, \text{Crash}, \text{UB}\}$, where UB denotes sanitizer violations. Fuzzing maximizes cumulative branch coverage $\text{Cov}(\mathcal{X}_{gen}) = \bigcup_{x \in \mathcal{X}_{gen}} \pi(x)$ and unique vulnerabilities under execution time T and LLM query budget B_{LLM}:

$$\max_{\mathcal{M}} \quad J(\mathcal{M}) = \mathbb{E}_{x \sim \mathcal{M}} \Big[\alpha \cdot |\text{Cov}(\mathcal{X}_{gen})| + \sum_{x \in \mathcal{X}_{gen}} \mathbb{I}(\mathcal{O}(x) \in \{\text{Crash}, \text{UB}\}) \Big] \tag{1}$$

subject to $\sum_{x \in \mathcal{X}_{gen}} \text{Cost}(x) \leq T$ and $\text{Cost}_{\text{LLM}}(\mathcal{X}_{gen}) \leq B_{\text{LLM}}$, where $\mathcal{M}$ is the generation policy.

Threat Model: Language processors (Clang/LLVM, GCC, WebAssembly runtimes, JavaScript JIT engines, smart contract compilers) constitute fundamental trusted infrastructure. An attacker seeks to construct malformed yet syntactically valid source files or scripts that pass compiler frontend checks, penetrate deep optimization passes (e.g., SSA transformation, loop vectorization, constant folding), and trigger critical security failures (heap overflows, use-after-free, assertion failures, or DoS). We assume a standard grey-box setting where the fuzzer tracks edge coverage and detects violations via ASan/UBSan.

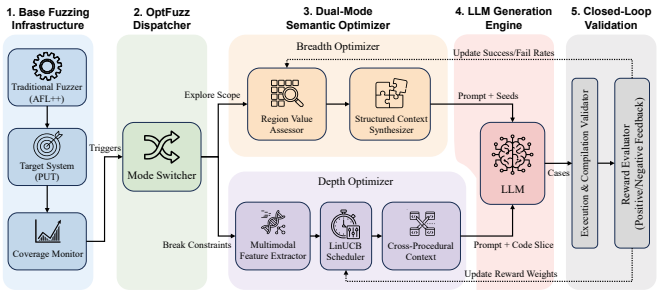


Figure 1: Overall architecture of OPTFUZZ, organizing LLM-assisted fuzzing into a hierarchical closed-loop optimization process combining high-throughput CGF with adaptive breadth/depth semantic optimization.

Algorithm 1 Main Optimization Loop of OPTFUZZ

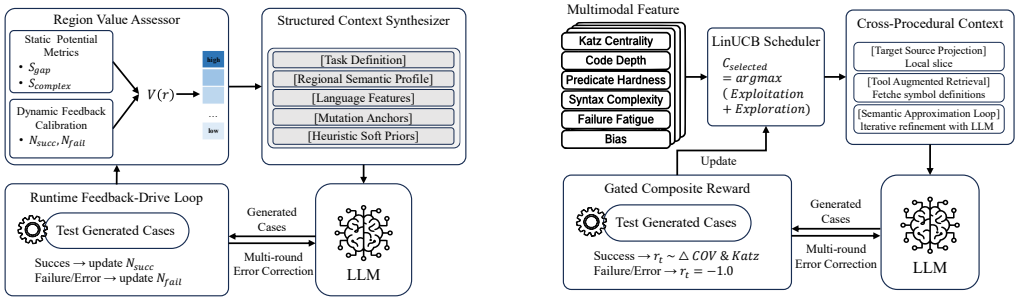
Require: Target Program $\mathcal{P}$, Initial Seed Corpus $\mathcal{S}_0$, LLM Budget B_{LLM}, Stagnation Threshold N_{stag}.
Ensure: Discovered Vulnerabilities $\mathcal{V}_{bugs}$, Coverage Bitmap $\mathcal{C}$.

$\mathcal{Q} \leftarrow \mathcal{S}_0$; $\mathcal{C} \leftarrow$ InitCoverage$(\mathcal{P}, \mathcal{Q})$; $\mathcal{V}_{bugs} \leftarrow \emptyset$; $A \leftarrow I_d$, $b \leftarrow \mathbf{0}_d$; Initialize $\mathcal{D}_{\text{region}}$;
while within time and budget limits **do**
  $s \leftarrow$ ScheduleQueue$(\mathcal{Q})$; $s' \leftarrow$ SyntaxAwareMutate(s); (cov_gain, status) $\leftarrow$ ExecuteTarget$(\mathcal{P}, s')$;
  if status $\in$ {Crash, SanitizerError} **then**
    $\mathcal{V}_{bugs} \leftarrow \mathcal{V}_{bugs} \cup \{(s', \text{status})\}$;
  end if
  if cov_gain > 0 **then**
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{s'\}$; $\mathcal{C} \leftarrow \mathcal{C} \cup$ cov_gain; ResetStagnationCounter();
  else
    IncrementStagnationCounter();
  end if
  if IsStagnated(N_{stag}) **and** LLMBudgetAvailable() **then**
    mode $\leftarrow$ SelectMode$(\mathcal{D}_{\text{region}}, \mathcal{C})$;
    if mode $==$ BREADTH **then**
      $r^* \leftarrow \arg\max_r V(r)$; $P_{\text{breadth}} \leftarrow$ BuildBreadthPrompt$(r^*, \mathcal{Q})$; $s_{\text{gen}} \leftarrow$ QueryLLM(P_{breadth});
      (gain, fb) $\leftarrow$ ValidateAndExecute$(\mathcal{P}, s_{\text{gen}})$; UpdateRegionStats$(\mathcal{D}_{\text{region}}, r^*, \text{gain}, \text{fb})$;
      if gain > 0 **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{s_{\text{gen}}\}$;
    else
      $\mathcal{B} \leftarrow$ ExtractBottlenecks$(\mathcal{P}, \mathcal{C})$; $c^* \leftarrow$ LinUCBSelect$(\mathcal{B}, A, b)$; {Eq. (3)}
      $s_{\text{gen}}, r_t \leftarrow$ DepthOptimizationLoop$(\mathcal{P}, c^*)$; Update LinUCB A, b using x_{c^*}, r_t; {Eq. (3)}
      if $s_{\text{gen}} \neq$ NULL **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{s_{\text{gen}}\}$;
    end if
  end if
end while
return $\mathcal{V}_{bugs}, \mathcal{C}$

3 OVERVIEW OF OPTFUZZ

OPTFUZZ organizes LLM-assisted fuzzing into a hierarchical closed-loop optimization process (Fig. 1): (1) *Bottom-Layer High-Throughput Loop*: Executes continuous byte mutations and syntax-aware AST recombination, tracking edge bitmaps and sanitizer traps at hundreds of execs/sec; (2) *Top-Layer Semantic Hub*: Operates asynchronously, intervening upon coverage stagnation (N_{stag} iterations without new coverage) to invoke Breadth or Depth Optimizers; (3) *Closed-Loop Validation & Reward Updating*: LLM-generated tests undergo compilation and instrumented execution, feeding diagnostics back into the region value database and LinUCB parameters.

Algorithm 1 formalizes the complete control flow. The Breadth Optimizer expands global frontiers in underexplored modules based on dynamic region values, while the Depth Optimizer performs



(a) Breadth Optimizer architecture. (b) Depth Optimizer architecture.

Figure 2: Architectures of OPTFUZZ's Breadth Optimizer (region value assessment and prompt synthesis) and Depth Optimizer (LinUCB target scheduling and tool-augmented semantic approximation).

targeted constraint penetration at bottleneck branches. By default, OPTFUZZ enforces a *breadth-first, depth-assisted* scheduling policy, dynamically decaying hard targets via failure penalties to balance exploration and exploitation.

4 BREADTH OPTIMIZER

The Breadth Optimizer (Fig. 2a) expands the global search frontier when local mutation produces redundant trials, allocating limited LLM calls to program areas with high expected marginal coverage.

4.1 REGION VALUE ASSESSMENT

To achieve precise resource allocation, OPTFUZZ partitions the PUT into logical regions by directory hierarchy (e.g., `clang/lib`, `llvm/IR`), avoiding function-level overhead. For each region $r \in \mathcal{R}$, expected value $V(r)$ combines static potential with dynamic feedback:

Static Potential Metrics: (1) *Coverage Gap*: $S_{gap} = 1 - R_{cov}$, with edge coverage ratio R_{cov}; (2) *Code Complexity*: $S_{complex} = N_{bb}/M_{bb}$, normalizing basic block count N_{bb} against maximum count M_{bb}.

Dynamic Feedback Calibration: Dynamic factor λ calibrates value using historical solving outcomes:

$$\lambda = \max\left(\frac{1 + \gamma \cdot N_{succ}}{1 + N_{fail} + \gamma \cdot N_{succ}}, 0.1\right), \qquad V(r) = (w_1 \cdot S_{gap} + w_2 \cdot S_{complex}) \cdot \lambda \quad (2)$$

where N_{succ}, N_{fail} record solving successes/failures in a sliding window, $\gamma = 3.0$ amplifies valid trials, and 0.1 prevents starvation.

4.2 CONTEXT-AWARE CLOSED-LOOP GENERATION

For target region r^*, OPTFUZZ builds a 5D structured prompt: (1) *Task Definition*: target binary, compiler version (Clang++ 22.1.1), standards (C++20/23), and format rules; (2) *Regional Semantic Profiling*: architecture intent extracted from developer manuals; (3) *Language Feature Injection*: advanced syntax specs (template specializations, array slicing); (4) *Mutation Anchors*: $K = 3$ diverse queue seeds with maximal Hamming distance; (5) *Heuristic Soft Priors*: structural substitution, dependency coupling, and boundary stress.

Feedback Loop: If a seed triggers new edges (ΔCov > 0), N_{succ} and λ rise and the seed is enqueued. If compilation fails, compiler diagnostics trigger multi-round interactive repair. Stagnation beyond $M_{retry} = 3$ increments N_{fail}, penalizing regional priority.

5 DEPTH OPTIMIZER

When exploration is blocked by deep bottleneck branches, the Depth Optimizer (Fig. 2b) models constraint penetration as contextual sequential decision-making.

5.1 Feature-Based Sequential Decision Model

To solve the constraint scheduling problem in depth exploration, OptFuzz formalizes it as a contextual sequential decision-making process over candidate bottleneck constraints $\mathcal{C}_t = \{c_1, \ldots, c_N\}$. We map scheduling to a contextual multi-armed bandit (LinUCB) observing a 6-dimensional multimodal feature vector $x \in \mathbb{R}^6$: (1) x_1: *Dynamic Katz Centrality*: $C_{\text{Katz}}(v_i) = \alpha \sum_{j=1}^{n} Z_{ij} C_{\text{Katz}}(v_j) + \beta_i$ ($\alpha = 0.1$); (2) x_2: *Normalized Code Depth*: $D(v)/D_{\max}$; (3) x_3: *Predicate Hardness*: $\mathbb{I}(T(c) \in \mathcal{T}_{hard})$ for equality checks and multi-variable logic; (4) x_4: *Syntax Complexity*: $L_{nest} + N_{var}$; (5) x_5: *Failure Fatigue*: $1 - 1/\log_2(2 + F_c)$ on consecutive failures F_c; (6) x_6: *Bias*: 1.0 (model intercept).

Gated Composite Reward: Truncated to $r_t = -1.0$ upon failure; on success, aggregates centrality and coverage: $r_t = B_{base} + v_1 \cdot \text{Score}_{Katz} + v_2 \cdot \log(1 + \Delta\text{Cov})$ ($B_{base} = 1.0$).

LinUCB Decision & Update: The scheduler selects constraint maximizing upper confidence bound:

$$c_{selected} = \arg\max_{c_i \in \mathcal{C}_t} \left(x_{c_i}^\top \hat{\theta}_t + \alpha \sqrt{x_{c_i}^\top A_t^{-1} x_{c_i}} \right), \quad A_{t+1}^{-1} = A_t^{-1} - \frac{A_t^{-1} x_t x_t^\top A_t^{-1}}{1 + x_t^\top A_t^{-1} x_t} \tag{3}$$

updating $b_{t+1} = b_t + r_t x_t$ and $\hat{\theta}_{t+1} = A_{t+1}^{-1} b_{t+1}$ in $O(d^2)$ time via Sherman-Morrison rank-1 update.

5.2 Cross-Procedural Context and Semantic Approximation

To prevent attention dilution from million-line codebases, the Depth Optimizer uses on-demand slicing: (1) *Target Source Projection*: slices target functions, annotating boundaries with `[Covered Execution Flow]` and `[Target Unreached Branch]`; (2) *External Tool Augmentation*: LLM emits structured JSON tool queries (e.g., {`"tool_call": "lookup_symbol", "symbol": "TypeInfo"`}) resolved via fast `ripgrep` symbol lookup; (3) *Semantic Approximation Loop*: code generation, compilation diagnostics, and runtime traces are fed back iteratively until the branch is solved or budget expires.

6 Implementation

We implemented OptFuzz in Python 3.10 and C++ (∼5,000 LOC), layered atop AFL++ (v4.08c) via a modular, loosely-coupled scheduler. OptFuzz supports flexible LLM backends (OpenAI GPT-5, Qwen3.5-27B, Qwen3.5-Flash). For underlying seed generation, we integrate an ANTLR4-based syntax-aware mutator (ANTLR, 2024) to manipulate structured ASTs across target languages without manual prompt engineering. Precise edge-to-source mapping is achieved using LLVM SanitizerCoverage collision-free bitmaps coupled with DWARF debug info.

Breadth Optimizer: Real-time coverage R_{cov} is parsed via `gcov`/`lcov`, while basic block counts N_{bb} are extracted at compile time using `llvm-profdata`. Historical solving outcomes (N_{succ}, N_{fail}) are logged into an SQLite database to update the dynamic calibration factor λ. Context prompts retrieve high-Hamming-distance seeds from the AFL++ queue alongside automated doc summaries, and generated tests are enqueued asynchronously.

Depth Optimizer: We extend AFL++ with lightweight bottleneck address dumping. An LLVM Analysis Pass and `graph-tool` compute CFG Katz centrality (x_1), while `libclang` extracts AST predicate features (x_3, x_4). The LinUCB scheduler is implemented in NumPy using Sherman-Morrison $O(d^2)$ rank-1 matrix updates. External symbol dependencies in LLM queries are dynamically retrieved via fast `ripgrep` symbol lookup and reinjected into iterative repair turns.

7 Evaluation

We evaluate the practical efficacy of OptFuzz in breaking through coverage bottlenecks and discovering vulnerabilities in semantically constrained systems. The experiments are designed to address the following research questions (RQs): **RQ1** (Exploration Capability & Baseline Comparison), **RQ2** (Real-World Vulnerability Discovery Efficacy), **RQ3** (Ablation & Policy Alternatives), and **RQ4** (Generality across LLM Backends, Cost & Throughput).

Table 1: Code Coverage and Vulnerability Comparison (24h Mean).

Tool	Clang++		Flang		Hermes		Lua	
	Cov	Bug	Cov	Bug	Cov	Bug	Cov	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
OPTFUZZ	**104,537**	**8**	**128,714**	**5**	**35,482**	**2**	**5,645**	**0**

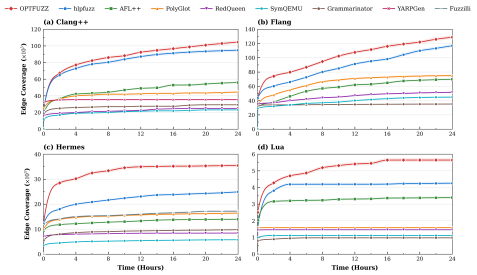


Figure 3: Edge coverage comparison over 24 hours.

7.1 Experimental Setup and Baselines

Experimental Environment. All experiments were deployed on an Ubuntu 22.04.5 LTS server equipped with dual Intel Xeon Platinum 8462Y+ processors (64 physical cores / 128 logical threads in total) and 1024 GB of physical memory. Each fuzzing task was executed within an independent Docker container pinned to dedicated CPU cores to reduce interference. The large language model backend uniformly employed the OpenAI GPT-5 API for interactions. For fairness, all seed-dependent baselines received initial corpora of identical origin and comparable scale, collected from open-source repositories and official language test suites. Controlled quantitative experiments used a 24-hour execution budget per tool-target pair, independently repeated 5 times.

Targets & Baselines. We selected 8 language processors across 6 programming languages (Clang/-Clang++ 22.1.1, Hermes 0.13.0, ChakraCore 97eu8, njs 0.9.6, Flang 22.1.1, Lua 5.5.0, and solc 0.8.34). Controlled 24-hour comparisons focused on Clang++, Flang, Hermes, and Lua. Baselines include 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., 2023), Fuzzilli (Groß et al., 2023), and HLPFUZZ (Yang et al., 2025b).

7.2 Comparison with Baselines (RQ1)

Code Coverage Evaluation. OPTFUZZ consistently achieves the highest code coverage across all evaluated targets, shattering the exploration bottlenecks of existing tools as depicted in Table 1 and Fig. 3. Compared to traditional CGF (AFL++) and constraint-solving methods (SymQEMU, REDQUEEN), OPTFUZZ demonstrates a substantial performance leap across diverse language paradigms:

- *Static Compilers (Clang++, Flang)*: On Clang++, OPTFUZZ improves branch coverage by 86.6% over AFL++ and by over 300% compared to SymQEMU. On Flang, OPTFUZZ achieves 128,714 coverage (+84.3% over AFL++).
- *Dynamic Engines (Hermes, Lua)*: On Hermes, OPTFUZZ exceeds Fuzzilli by 106.5% and AFL++ by 154.4%. On Lua, OPTFUZZ reaches 5,645 edge coverage (+66.3% over AFL++).
- *Advantage over SOTA LLM Baseline (HLPFUZZ)*: Against HLPFUZZ, OPTFUZZ achieves additional coverage gains from 10.2% (Clang++) to 42.4% (Hermes), confirming the efficacy of closed-loop dual optimization.

24-Hour Vulnerability Discovery Comparison. In 24-hour controlled trials (Table 1), OPTFUZZ triggered 15 unique vulnerabilities (8 Clang++, 5 Flang, 2 Hermes), whereas traditional fuzzers found ≤ 1 and HLPFUZZ found 8. Extended 72-hour runs on Clang++ confirmed that baselines (AFL++: 59k, HLPFUZZ: 96k) still failed to trigger deep defects without feedback approximation.

Temporal Exploration Dynamics. Early phase (0–4h) sees shallow parser saturation; middle phase (4–12h) baselines plateau while OPTFUZZ's Breadth Optimizer unlocks unreached modules; late phase (12–24h) Depth Optimizer solves narrow bottleneck branches with LinUCB, maintaining continuous growth.

Table 2: Vulnerability Distribution of OPTFUZZ in Real-World Discovery Testing. AF: Assertion Failure; SV: State Violation; SOF: Stack Overflow; ND: Null Dereference; HBOF: Heap Buffer Overflow; HUAF: Heap Use After Free; FPE: Floating Point Exception; SUAS: Shift/Unsigned/Arithmetic/Sign issues.

Target	AF	SV	SOF	ND	HBOF	HUAF	FPE	SUAS	Total
Clang++	0	2	0	1	11	13	1	1	29
Clang	0	4	0	2	1	4	1	3	15
Flang	18	1	1	0	0	1	0	0	21
Hermes	0	2	0	0	0	0	0	0	2
ChakraCore	0	0	0	2	0	0	0	0	2
njs	0	2	0	0	0	0	0	0	2
Lua	0	1	0	0	1	0	0	0	2
solc	1	0	0	0	0	0	0	0	1
Total	**19**	**12**	**1**	**5**	**12**	**18**	**2**	**4**	**74**

```
struct S {
  struct {} arr[1][1ULL<<40];
} a, b = a;
```

(a) AST eval (Clang++)

```
struct S2 {
  unsigned a : 10;
  unsigned c : 1 << 26;
} arr2[2] = {1};
```

(b) CodeGen (Clang++)

```
const crash =
  ({ [i]: a }, i) => {};
```

(c) TDZ forward ref (Hermes)

Figure 4: Reduced crash triggers discovered by OPTFUZZ in Clang++ and Hermes.

7.3 REAL-WORLD VULNERABILITY DISCOVERY (RQ2)

During one-month stress testing across all 8 targets, OPTFUZZ discovered 74 real-world zero-day vulnerabilities (57 confirmed by official developers; Table 2). The confirmed issues span frontend parsing, AST construction, type checking, constant expression evaluation, IR code generation, and runtime execution passes.

Case Studies & Mechanistic Insights (Fig. 4):

1. *Case A (Clang++ AST Constant Expression Evaluation, Fig. 4a)*: Directly declaring an extreme array (e.g., `int arr[1ULL << 40]`) is immediately caught and rejected by the frontend semantic analyzer (`Sema`) as "size out of bounds," preventing it from reaching the backend. OPTFUZZ's Depth Optimizer captured this compiler rejection as negative feedback. Guided by this feedback, the LLM executed "Semantic Camouflage": it nested the extreme array bound inside empty structs and activated it via global implicit copy initialization (`b = a`). This legally disguised code bypassed frontend validation and deferred calculations to the backend APValue evaluator, triggering integer truncation and a heap buffer overflow.
2. *Case B (Clang++ CodeGen Pointer Out-of-Bounds, Fig. 4b)*: Triggering this crash requires simultaneously compiling an out-of-bounds bit-field and aggregate initialization. Traditional fuzzers face massive probability barriers in assembling these distinct syntactic elements while satisfying structural constraints. When OPTFUZZ detected coverage stagnation in Clang++'s `AppendBitField` function, the Depth Optimizer nested a bit-field exceeding 67 million bits in width inside a struct and appended an array initialization list (`arr2[2] = {1}`), causing an out-of-bounds pointer calculation during constant struct emission.
3. *Case C (Hermes SSA Dominator Tree Null Dereference, Fig. 4c)*: Arises from a mismatch between JavaScript's dynamic evaluation order and static optimizer assumptions. The combination of object destructuring and computed properties can create a forward reference inside the Temporal Dead Zone (TDZ). Under control-flow perturbation feedback, the Depth Optimizer guided the LLM to invert variable lifecycles, constructing the parameter pattern `({ [i]: a }, i) => {}`. When entering Memory-to-Register Promotion (Mem2Reg), this pattern violated topological assumptions regarding basic block predecessors, causing a null pointer dereference during SSA Dominator Tree construction.

Deep Mechanistic Insights: Why Traditional & Stateless LLM Fuzzers Fail. Existing fuzzers suffer from two fundamental failure modes on complex language processors: (1) *Probabilistic*

Table 3: Ablation Study Comparison (24-Hour Test).

Variant	Clang++		Flang		Hermes		Lua	
	Cov	Bug	Cov	Bug	Cov	Bug	Cov	Bug
OPTFUZZ	**104,537**	**8**	**128,714**	**5**	**35,482**	**2**	**5,645**	**0**
Breadth-only	92,184	5	114,473	2	25,174	1	4,386	0
-w/o VA	78,436	2	97,852	1	18,924	0	4,017	0
-w/o IO	70,918	2	88,647	1	15,307	0	3,781	0
Depth-only	95,874	5	118,093	3	25,300	1	4,497	0
-w/o DS	81,263	3	100,574	2	19,842	1	4,163	0
-w/o IO	73,517	2	91,638	1	16,284	0	3,946	0

Table 4: Constraint Schedulers.

Scheduler	Coverage	Bug
LinUCB	**104,537**	**8**
Thompson	99,504	6
ϵ-greedy	94,419	4
Failure-Only	90,102	3
Katz-Only	89,436	3

Vacuum in Traditional CGF: Defects like Cases A and B require extreme scalar expressions and compound struct initializations. For byte-level mutators, the joint probability of generating such valid combinations while maintaining strict grammar validity is virtually zero ($p \to 0$); (2) *Safe Code Bias in Stateless LLMs*: Pre-trained LLMs possess a statistical bias toward idiomatic, benign patterns. In single-shot prompting without feedback, LLMs generate standard syntax constructs that pass frontend checks but fail to stress backend evaluators. By capturing intermediate compiler diagnostics as negative feedback, OPTFUZZ forces the LLM to perform semantic camouflage and control-flow inversion, driving tests directly into fragile optimizer logic.

7.4 ABLATION STUDY (RQ3)

To dissect the concrete contributions of OPTFUZZ's core components and validate our design decisions, we conduct controlled 24-hour ablation experiments across four representative targets (Clang++, Flang, Hermes, and Lua). We evaluate six ablated variants by systematically isolating or removing key modules: (1) **Breadth-only**: Disables the Depth Optimizer entirely, relying solely on region-value-guided breadth exploration to steer the LLM toward underexplored modules upon stagnation; (2) **Breadth *-w/o VA***: Replaces dynamic Region Value assessment ($V(r)$ in Eq. (2)) with uniform random directory selection to assess the impact of value estimation; (3) **Breadth *-w/o IO***: Strips the compiler-diagnostic repair loop, reducing prompt queries to stateless, single-shot generation without feedback retries; (4) **Depth-only**: Disables the Breadth Optimizer entirely, relying solely on LinUCB-guided depth constraint solving to penetrate bottleneck branches; (5) **Depth *-w/o DS***: Replaces the multimodal LinUCB bandit scheduler (Eq. (3)) with uniform random selection among candidate bottleneck branches; (6) **Depth *-w/o IO***: Eliminates the interactive semantic approximation loop, issuing only a single-turn prompt without multi-round execution trace feedback.

Component Contributions & Empirical Findings. Table 3 summarizes the comparative results:

1. *Dual-Optimizer Synergy*: Isolating either optimizer alone (*Breadth-only* vs. *Depth-only*) decreases coverage by 11.8%–29.0% and halves bug discovery. Breadth expansion uncovers diverse architectural frontiers but stalls when encountering rigid predicate gates; conversely, depth solving penetrates narrow paths but lacks global context to navigate broader subsystems. Their synergy produces optimal exploration.

2. *Role of Strategic Value Assessment and Dynamic Scheduling*: Disabling Region Value assessment (*-w/o VA*) or LinUCB scheduling (*-w/o DS*) causes severe coverage degradation (22.2%–46.7%). Without quantitative ranking, the LLM squanders queries on trivial boilerplate files or mathematically impossible branch guards.

3. *Indispensability of Closed-Loop Iterative Repair*: Disabling closed-loop feedback (*-w/o IO*) causes the sharpest decline across both engines (up to 56.9% coverage loss on Hermes). This proves that single-shot generative LLM fuzzing cannot overcome deep semantic constraints without runtime error diagnostics.

Alternative Scheduling Policy Comparison. To rigorously justify LinUCB over other exploration-exploitation heuristics, we evaluate four alternative schedulers on Clang++ (Table 4): Thompson Sampling, ϵ-greedy, Failure-penalized ranking, and pure Katz centrality. LinUCB achieves the highest coverage ($104,537$) and bug yield (8) by effectively combining topological reward potential with historical solving resistance.

Table 5: LLM Backend Generalization and Cost/Throughput Profile (Clang++, 24-Hour Test).

Fuzzer / Configuration	LLM Backend	Throughput (exec/s)	Edge Coverage	Unique Bugs	Cost / Hour
Baseline (AFL++)	None	∼**177.8**	56,029	1	**$ 0.00**
HLPFUZZ	SOTA LLM	∼44.2	94,782	4	$ 0.58
OPTFUZZ	Qwen3.5-Flash (Light)	∼46.1	92,241 (+64.6%)	4	$ 0.08
OPTFUZZ	Qwen3.5-27B (Open)	∼44.9	97,346 (+73.7%)	6	Self-Hosted
OPTFUZZ	**GPT-5 (Default)**	**∼45.6**	**104,537 (+86.6%)**	**8**	**$ 0.55**

7.5 MECHANISTIC, GENERALITY & OVERHEAD ANALYSIS (RQ4)

To understand the internal mechanisms and efficiency of OPTFUZZ, we examine model-agnostic generalization across backends, cost-throughput metrics, and prompt sensitivity.

Backend Generalization. As shown in Table 5, OPTFUZZ achieves 92,241 coverage (+64.6%) with lightweight Qwen3.5-Flash, and 97,346 coverage (+73.7%) with open-source Qwen3.5-27B, demonstrating robust model-agnostic generality.

Cost and Throughput. OPTFUZZ runs native CGF $> 99\%$ of the time. Under semantic generation benchmarks (Table 5), OPTFUZZ attains higher throughput (∼45.6 execs/s) and lower cost ($0.55/h) than HLPFUZZ ($0.58/h) while finding twice as many bugs, as failure-fatigue suppresses futile token spend.

Prompt Sensitivity. Evaluating reference seed count (K) reveals an inverted U-curve: $K = 1$ $(95, 412)$, $K = 3$ $(\mathbf{104}, \mathbf{537})$, $K = 5$ $(104, 416)$, $K = 7$ $(99, 350)$. $K = 3$ balances cross-over synergy with context dilution (Lost in the Middle).

8 LIMITATIONS AND THREATS TO VALIDITY

Limitations and Future Work. (1) *Source Code Dependency*: OPTFUZZ leverages AST and CFG analysis to construct structured prompt contexts for the Depth Optimizer, making it best suited for grey-box or white-box environments. Black-box binary fuzzing with binary-level disassembly analysis remains future work. (2) *Cross-File Context*: While tool augmentation recovers local struct and symbol definitions, complex multi-file data flows and distributed dependencies can still exceed context window constraints. (3) *Multi-Agent Architecture*: Decomposing optimization into specialized agent teams (planning, mutation, review, retrieval) represents a promising direction for future research.

Threats to Validity. (1) *Internal Validity*: Controlled by using industry-standard foundations (AFL++ v4.08c, ANTLR4, LLVM passes) and repeating 24h experiments 5 times with 95% confidence intervals. (2) *External Validity*: Validated across 8 industrial language processors in 6 programming languages and 3 diverse LLM backends. (3) *Construct Validity*: Evaluated on cumulative edge coverage and ASan/addr2line-deduplicated unique bugs (57 confirmed).

9 CONCLUSION

OPTFUZZ reframes LLM-assisted fuzzing from a stateless generation paradigm into a hierarchical closed-loop semantic optimization process. By reconciling high-throughput coverage-guided fuzzing with dynamic region-value breadth exploration and LinUCB depth constraint solving, OPTFUZZ continuously adapts its generation trajectory using execution diagnostics and compiler feedback. Extensive evaluation across 8 industrial-grade language processors across 6 distinct programming languages demonstrates that OPTFUZZ achieves substantial coverage improvements over state-of-the-art fuzzers and uncovers 74 real-world zero-day vulnerabilities (57 confirmed by upstream maintainers), proving the profound efficacy of closed-loop semantic optimization for critical software security testing.

AI USE STATEMENT

In this work, generative AI tools (OpenAI GPT-5 and Qwen models) were used as core components of the automated fuzzing framework for test case generation, mutation, and constraint repair. Generative AI was also used for grammar checking and polishing manuscript presentation. All AI-assisted

artifacts and text were rigorously reviewed and verified by the authors, who take full responsibility for the content of this paper.

Ethics Statement

This research adheres to ethical vulnerability disclosure practices. All 74 discovered vulnerabilities were responsibly reported to upstream maintainers (LLVM Project, Hermes, Lua, etc.) via private security advisories prior to submission. No private user data was involved. The public artifact repository contains only anonymized test suites, minimized bug triggers, and evaluation scripts.

Reproducibility Statement

Detailed mathematical formulations for Region Value assessment, multimodal features, composite rewards, and LinUCB updates are provided in Sections 2.2, 4, and 5. Complete prompt templates are in Appendix A. All target versions are specified in Section 7.1. The anonymized prototype implementation, fuzzing harnesses, docker environments, and raw data are publicly available at `https://osf.io/5g2ez/overview?view_only=9a341081ce104a60ad8b3102a7ebb430`.

References

ANTLR. Grammars written for antlr v4. `https://github.com/antlr/grammars-v4`, 2024. Accessed: 2024-12-15.

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Nautilus: Fishing for deep bugs with grammars. In *2019 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 *NDSS*, volume 19, pp. 1–15, 2019b.

Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. ELFuzz: Efficient input generation via LLM-driven synthesis over fuzzer space. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 6279–6298, Seattle, WA, August 2025. USENIX Association. ISBN 978-1-939133-52-6. URL `https://www.usenix.org/conference/usenixsecurity25/presentation/chen-chuyang`.

Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 642–658. IEEE, 2021.

Y. Deng, C. S. Xia, H. Peng, and et al. 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*, pp. 423–435, 2023.

Yi Deng, C. S. Xia, Chenyuan Yang, and et al. 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*, pp. 1–13, 2024.

Yifeng Ding, Jiawei Liu, Yuxiang Wei, and Lingming Zhang. Xft: Unlocking the power of code instruction tuning by simply merging upcycled mixture-of-experts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12941–12955, 2024.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024.

Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou (eds.), *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020. URL `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (eds.), *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pp. 1051–1065. ACM, 2022. doi: 10.1145/3548606.3560602. URL https://doi.org/10.1145/3548606.3560602.

Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

Guofeng He, Yichen Xin, Xiuchuan Cheng, and G. Yin. Afl++: A vulnerability discovery and reproduction framework. *Electronics*, 13:912, 02 2024. doi: 10.3390/electronics13050912.

Renáta Hodován, Ákos Kiss, and Tibor 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*, pp. 45–48, 2018.

J. Hu, Q. Zhang, and H. Yin. Augmenting greybox fuzzing with generative ai. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 491–503, 2023.

Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. On the challenges of fuzzing techniques via large language models, 2025. URL https://arxiv.org/abs/2402.00350.

S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2312–2323. IEEE, 2023.

Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007)*, pp. 7–7. IEEE, 2007.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pp. 661–670, 2010.

Z. Li, J. Wu, X. Ling, and et al. The seeds of the future sprout from history: Fuzzing for unveiling vulnerabilities in prospective deep-learning libraries. *arXiv preprint arXiv:2412.01317*, 2024. Preprint.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for c++ and data-parallel languages. *Proceedings of the ACM on Programming Languages*, 7(PLDI): 1826–1847, 2023.

Haoyang Ma. A survey of modern compiler fuzzing. *CoRR*, abs/2306.06884, 2023. doi: 10.48550/ARXIV.2306.06884. URL https://doi.org/10.48550/arXiv.2306.06884.

R. Meng, M. Mirchev, M. Böhme, and et al. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

Y. Oliinyk, M. Scott, R. Tsang, and et al. Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 883–900, 2024.

Shiwen Ou, Yuwei Li, Lu Yu, Chengkun Wei, Tingke Wen, Qiangpu Chen, Yu Chen, Haizhi Tang, and Zulie Pan. Mirrorfuzz: Leveraging llm and shared bugs for deep learning framework apis fuzzing. *IEEE Transactions on Software Engineering*, pp. 1–16, 2025. doi: 10.1109/TSE.2025.3619966.

Rohan Padhye, Caroline Lemieux, Koushik Sen, et al. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 329–340, 2019.

Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Ndss 2021, network and distributed system security symposium*. Internet Society, 2021.

F. Qiu, P. Ji, B. Hua, and et al. Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pp. 103–112. IEEE, 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 *23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, volume 30, 2017.

P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu. The progress, challenges, and perspectives of directed greybox fuzzing. *Software Testing, Verification and Reliability*, 24(2):e1869, 2024a.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4706–4721, 2024b.

C. S. Xia, M. Paltenghi, J. Le Tian, and et al. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

H. Xu, W. Ma, T. Zhou, and et al. Ckgfuzzer: Llm-based fuzz driver generation enhanced by code knowledge graph. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 243–254. IEEE, 2025.

C. Yang, Y. Deng, R. Lu, and et al. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735, 2024.

C. Yang, Z. Zhao, and L. 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, Volume 2*, pp. 560–573, 2025a.

Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. *Hybrid language processor fuzzing via LLM-based constraint solving*. USENIX Association, USA, 2025b. ISBN 978-1-939133-52-6.

J. Yu, Y. Shao, H. Miao, and et al. Promptfuzz: Harnessing fuzzing techniques for robust testing of prompt injection in llms. *arXiv preprint arXiv:2409.14729*, 2024. Preprint.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 745–761, 2018.

Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. Low-cost and comprehensive non-textual input fuzzing with llm-synthesized input generators. In Lujo Bauer and Giancarlo Pellegrino (eds.), *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pp. 6999–7018. USENIX Association, 2025a. URL `https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-kunpeng`.

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 (Volume 1: Long Papers)*, pp. 13784–13802, 2025b.

Lei Zhao, Yue Duan, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Network and Distributed System Security Symposium (NDSS), 2019.

Chen Zhou, Qi Zhang, Lei Guo, et al. Towards better semantics exploration for browser fuzzing, 2023.

Xiaogang Zhu, Wei Zhou, Qing-Long Han, Wanlun Ma, Sheng Wen, and Yang Xiang. When software security meets large language models: A survey. *IEEE/CAA Journal of Automatica Sinica*, 12(2): 317–334, 2025.

A Prompt Templates and Construction Details

This appendix provides the prompt templates utilized by OptFuzz's Breadth and Depth Optimizers (Listings 1 and 2).

Listing 1: Prompt Template for Breadth Optimizer.

```
[SYSTEM INSTRUCTION] Synthesize novel, compilable source code to explore unreached logic
     in {{TARGET_REGION_PATH}}.
[TARGET CONTEXT] Target: {{TARGET_NAME}} ({{VERSION}}) | Standard: {{LANGUAGE_STANDARD}}
     | Profile: {{REGIONAL_SEMANTIC_DESCRIPTION}}
[LANGUAGE FEATURES] {{LANGUAGE_FEATURE_INJECTION}}
[MUTATION ANCHORS]
- Seed 1: {{SEED_1_CONTENT}} | Seed 2: {{SEED_2_CONTENT}} | Seed 3: {{SEED_3_CONTENT}}
[HEURISTIC PRIORS] 1. Syntactic Structural Substitution 2. Data-Flow Dependency Coupling
     3. Control-Flow Inversion & Boundary Stresses
[FEEDBACK HISTORY] {{PREVIOUS_DIAGNOSTIC_FEEDBACK}}
```

Listing 2: Prompt Template for Depth Optimizer.

```
[SYSTEM INSTRUCTION] Mutate/repair input to force execution through [TARGET UNREACHED
    BRANCH]. Emit JSON tool call {"tool_call": "lookup_symbol", "symbol": "<name>"} if
    missing definitions, else output updated code.
[BOTTLENECK TARGET] File: {{SOURCE_FILE}} | Function: {{FUNCTION_NAME}} | Predicate: {{
    BRANCH_PREDICATE_EXPR}}
[SLICED SOURCE CODE] {{SLICED_SOURCE_CODE_WITH_COVERAGE_ANNOTATIONS}}
[CURRENT SEED] {{CURRENT_SEED_CODE}}
[EXECUTION FEEDBACK] Exit: {{EXIT_STATUS}} | Trace: {{TRACE_DIAGNOSTIC}} | Error: {{
    COMPILER_OR_SANITIZER_LOG}}
```

B Discovered Vulnerabilities & Statistical Confidence Analysis

Table 6 details representative real-world vulnerabilities discovered by OptFuzz across all 8 target language processors, alongside statistical confidence metrics across 5 independent 24-hour trials.

Table 6: Representative Real-World Vulnerabilities & Statistical Confidence (5 Runs).

Discovered Vulnerabilities			**Coverage 95% Confidence Intervals (5 Runs)**			
Target	Type	Subsystem	Target	Tool	Mean ± SD	95% CI
Clang++	Heap Buffer Overflow	`AST/ExprConstant`	Clang++	AFL++	$56,029 \pm 2,134$	$[53,938, 58,120]$
Clang++	Segmentation Fault	`CodeGen/CGExpr`		HLPFUZZ	$94,782 \pm 3,025$	$[91,817, 97,747]$
Clang++	Heap Use After Free	`Sema/SemaTemplate`		**OptFuzz**	$\mathbf{104,537 \pm 3,382}$	$[\mathbf{101,523}, \mathbf{107,551}]$
Clang	Assertion Failure	`CodeGen/CGCall`	Flang	AFL++	$69,843 \pm 2,510$	$[67,383, 72,303]$
Clang	State Violation	`StaticAnalyzer`		HLPFUZZ	$116,843 \pm 3,892$	$[113,029, 120,657]$
Flang	Assertion Failure	`Evaluate/fold`		**OptFuzz**	$\mathbf{128,714 \pm 4,215}$	$[\mathbf{125,000}, \mathbf{132,428}]$
Flang	Stack Overflow	`Parser/builder`	Hermes	AFL++	$13,947 \pm 582$	$[13,377, 14,517]$
Hermes	Null Dereference	`Optimizer/Mem2Reg`		HLPFUZZ	$24,917 \pm 914$	$[24,021, 25,813]$
Hermes	State Violation	`VM/Runtime`		**OptFuzz**	$\mathbf{35,482 \pm 1,120}$	$[\mathbf{34,484}, \mathbf{36,480}]$
ChakraCore	Null Dereference	`ByteCodeGen`	Lua	AFL++	$3,395 \pm 124$	$[3,273, 3,517]$
njs	State Violation	`src/njs_vm`		HLPFUZZ	$4,258 \pm 185$	$[4,077, 4,439]$
Lua	Heap Buffer Overflow	`src/lvm`		**OptFuzz**	$\mathbf{5,645 \pm 208}$	$[\mathbf{5,441}, \mathbf{5,849}]$
solc	Assertion Failure	`libsolidity/Yul`	All gains significant at $p < 0.001$			

Statistical Variance & Significance Analysis. Across all evaluation targets, the lower bound of OptFuzz's 95% confidence interval strictly exceeds the upper bound of the strongest baseline (101.5k $>$ 97.7k on Clang++, 125.0k $>$ 120.7k on Flang, 34.5k $>$ 25.8k on Hermes, and 5.4k $>$ 4.4k on Lua). Two-tailed Student's t-tests confirm that all improvements are statistically significant at $p < 0.001$.