USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving

Yupeng Yang, Shenglong Yao, Jizhou Chen, and
Wenke Lee, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/yang-yupeng

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving

Yupeng Yang, Shenglong Yao, Jizhou Chen, Wenke Lee
Georgia Institute of Technology

Abstract

Language processors, such as compilers and interpreters, play a crucial role in modern cyberspace. Faulty language processors can lead to severe consequences such as incorrect functionalities or malicious attacks. It is non-trivial to automatically test language processors to detect faulty behaviors, because language processors are multistaged and require various complex constraints to reach deep program states. Existing testing (fuzzing) approaches either fail to effectively generate inputs that satisfy the complex constraints or fail to generalize due to their heavy reliance on target-specific constraint modeling heuristics. In this paper, we explore the potential of using LLMs for constraint solving to address these limitations and identify two challenges regarding constraint prioritization and context construction. To effectively address these challenges, we propose two novel solutions, *hybrid centrality prioritization* and *iterative context construction*. We implement the solutions in a hybrid fuzzing framework, HLPFuzz, which leverages an LLM to overcome complex constraints and reach deep program states. In our evaluation, HLPFuzz successfully discovers 52 bugs in 9 popular language processors, of which 37 are confirmed and 14 are fixed. HLPFuzz also outperforms state-of-the-art solutions by up to 190% in code coverage and discovers 5x more bugs than the second-best fuzzer, with minimal reliance on target-specific heuristics.

1 Introduction

Language processors, such as compilers and interpreters, play a foundational role in modern cyberspace. Faulty language processors can lead to wrongly compiled programs and even malicious attacks. For instance, miscompilation of memory-safe code can produce memory-unsafe binaries [25, 26, 66], and faulty interpreters can lead to attacks such as denial-of-service and remote code execution [65, 92].

Recent works on software testing, particularly greybox fuzzing [21, 28, 36], have thrived on testing various kinds of programs to automatically detect faulty behaviors. Fuzzers generate random test cases and feed them into the target program under test, meanwhile monitoring faulty behaviors and reporting them to human analysts. Guided by coverage feedback, greybox fuzzers use mutations to generate interesting test cases that cover more and more code regions. However, language processors are complex programs that process input in multiple stages, each imposing various complex constraints on the input, such as lexical, syntactic, and semantic constraints (see Fig. 1). It is difficult for pure random mutations to generate test cases satisfying such constraints.

To address the problem of complex input constraints, researchers have proposed two categories of solutions: generic solutions and target-specific solutions. Generic solutions aim at solving complex constraints without target-specific heuristics [3, 5, 57, 71, 90]. The typical approach is to track the relationship between the input and the evaluation of branching conditions within the program. Taint tracking and concolic execution fall into this category. Taint tracking enables fuzzers [24, 31, 51, 63] to learn which parts of the input affect the branching condition checks. Afterward, fuzzers can focus the mutation on the relevant parts of the input to increase the chances of passing such checks. Concolic execution combines a fuzzer with a symbolic solver [9, 12, 59, 78, 90], which replaces the input with symbolic values and then tracks computations stemming from them. Afterward, branching conditions can be represented in SMT formulas and solved by SMT solvers [17, 74]. Unfortunately, these solutions fall short on programs as complex as language processors. Taint tracking would fail after the tokenization stage, which applies constraints on every input byte, rendering taint tracking pointless. Concolic execution would fail due to incomplete semantics modeling or over-constraining [12]. To deal with such limitations, researchers have proposed target-specific solutions for specific language processors [30, 55, 88, 97]. Although these approaches show effectiveness on their respective targets, they require heavy manual effort to embed target-specific heuristics into the fuzzer, making them ungeneralizable to other targets. Further, researchers have proposed annotation-based approaches [11, 89]. Still, they require expert knowledge of

the annotation language and an understanding of the target's input constraints, making them non-trivial to apply to new targets. Semi-generic approaches, such as token-level [67] and grammar-level [4, 80] fuzzing, have been proposed for language processors. Due to the wide availability of tokenizers and grammar, these approaches can be generalized. However, they only aim to overcome lexical and syntax constraints, which are still ineffective in exploring deeper logics (*e.g.*, in optimization and execution stages) behind semantic constraints [11, 55]. To date, we face a dilemma where generic solutions fail to effectively handle language processors while target-specific solutions struggle to generalize.

Recently, large language models (LLMs) have gained vast attention due to their emergent abilities in diverse tasks including code understanding [16, 53] and reasoning [82]. It prompts us to consider whether LLMs can address the aforementioned dilemma by reasoning and solving constraints through code understanding. In this research, we explore the feasibility of utilizing LLMs to solve the constraints that hinder language processor fuzzers from making progress. However, we identify two unique challenges that must be addressed to make an LLM-based approach practical. ***First, it is unclear how to prioritize constraints to maximize coverage gain***. Modern language processors easily contain up to millions of lines of code (see Table 4). Since inference with LLMs is not free [85, 86], it is important to prioritize the constraints to solve. However, constraint prioritization is non-trivial. An ideal prioritization strategy should not only favor those having greater potential to reach more code regions, but also account for the solving difficulty for LLMs. This is because we notice that LLMs may easily solve some constraints, but fail entirely with others due to lacking reasoning capabilities or inherently unsolvable problems (*e.g.*, one-way functions). To date, effectively satisfying these two properties to maximize coverage gain remains an open challenge. ***Second, it is non-trivial to decide the degree of context presentation***. To instruct an LLM to solve a constraint, we need to provide the constraint context (*e.g.*, source code) for it to reason about. Presenting the entire code base is impractical, considering the magnitude of modern language processor code bases. Presenting execution traces or slices is not ideal either because their sizes can still be huge, with total lines of code easily exceeding 10k in our experiments. On the flip side, presenting only a narrow context (*e.g.*, the function enclosing the constraint's evaluation point) can miss necessary information for the LLM to understand the full picture of the constraint. Therefore, it remains unclear how to present sufficient and concise context to achieve both effectiveness and efficiency.

In this work, we systematically analyze these challenges and propose our solutions, *hybrid centrality prioritization* and *iterative context construction*. To solve the first challenge, we combine centrality analysis with a hybrid calibration mechanism based on historical solving feedback. Afterward, the constraints with greater coverage increasing potential and more "solvable" are assigned higher priorities. For the second challenge, we start with a narrow context. Upon failure, we use *divergence analysis* and *language server integration* to iteratively refine the context until reaching a successful solution or exceeding the threshold. We implement our solutions into an end-to-end hybrid fuzzing framework, HLPFuzz, that can effectively utilize an LLM to explore deeper program states. To demonstrate its generalizability and effectiveness, we apply HLPFuzz on 9 mainstream language processors from 7 different languages, discovering 52 bugs, of which 37 are confirmed by the developers, and 14 are fixed. We also compare HLPFuzz with state-of-the-art generic and target-specific solutions. HLPFuzz achieves up to 190% performance in code coverage and finds 5x more bugs when compared with the second-best fuzzer.

In summary, we make the following contributions:

- We systematically analyze the limitations of existing approaches in handling complex constraints when applied to language processor fuzzing.
- We explore the potential of utilizing LLMs to solve the constraints, identify unique challenges, and propose effective solutions.
- We implement our solutions in a hybrid fuzzing framework, HLPFuzz, which effectively leverages an LLM to overcome complex constraints and reach deeper logic.
- We perform extensive evaluations for HLPFuzz on 9 popular language processors of 7 languages. HLPFuzz outperforms existing solutions and has identified 52 bugs in the latest targets we have tested on.

2 Problem

This section begins with an overview of language processors. We then examine existing solutions and their limitations in handling complex constraints while maintaining generalizability. Next, we explore the opportunities and challenges of leveraging LLMs to address these issues. Finally, we introduce our novel approaches to overcome the challenges.

2.1 Scope and Terminology

Language processors in this paper refer to *programming* language processors. A *constraint* is a set of rules the input must follow to trigger the execution of a specific *code region* in the program under test. At the binary level, a code region refers to a basic block [42]. *Solving a constraint* means generating an input that triggers the corresponding code region. Constraints are typically evaluated through branching conditions (*e.g.*, `if-else` conditions). *Complex constraints* are those that cannot be easily solved through pure random mutations.

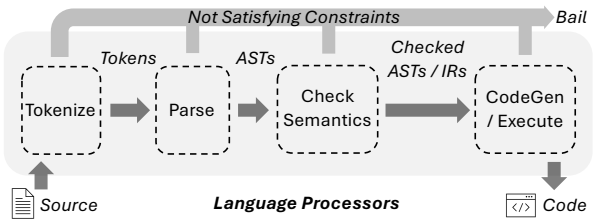

Fig. 1: The General Workflow of Language Processors. Modern language processors consist of multiple stages that demand complex constraints on the input.

2.2 Language Processors

As illustrated in Fig. 1, language processors typically consist of multiple stages, including tokenization, parsing, semantics checking, code generation or execution. Each stage imposes specific constraints on the input. For example, parsing requires syntactically correct input, and semantics checking ensures compliance with language rules (*e.g.*, types and data dependencies). To reach one stage, the input has to satisfy all constraints from prior stages. Inside the stage, finer constraints also exist. For instance, the code generation stage of `Clang++` only executes function optimizations when the input contains functions. Complex input constraints (*e.g.*, specific program structures with correct types and data dependencies) often need to be satisfied to trigger deep code regions.

2.3 Fuzzers and Limitations

Fuzzing feeds random input to a program and checks for faulty behaviors [21, 22, 35, 70, 91]. Modern fuzzers rely on random mutations (*e.g.*, bit-flips) and coverage feedback to produce inputs covering more code regions. However, naive random mutations perform badly at exploring deep logic behind complex constraints. To overcome this problem, researchers have proposed generic and target-specific solutions.

Generic Solutions. Generic solutions aim at solving complex constraints without target-specific heuristics. The typical approach is to track the relationship between the input and the evaluation of the constraints. Taint tracking and symbolic/concolic execution fall into this category. Taint tracking enables fuzzers [24, 31, 51, 63] to learn which parts of the input affect constraint evaluation. Afterward, fuzzers focus their mutations on the relevant parts to increase the chances of solving the constraints. Symbolic execution replaces the input with symbolic values and then tracks computations stemming from them. Then, constraint evaluation can be represented in SMT formulas, and SMT solvers [17, 74] help determine the exact inputs to solve them. In the context of fuzzing, symbolic execution is often enhanced with concolic execution [33, 59, 90], where concrete values and execution paths from fuzzers help speed up SMT solving and path exploration. Additionally, approaches like REDQUEEN [5] and others [10, 57, 81] aim to reduce constraint tracking overhead through approximation.

```
1  // ... skipped header includes
2  extern "C" const TSLanguage *tree_sitter_cpp(void);
3
4  bool check(TSNode &node) {
5    const char *node_type = ts_node_type(node);
6    if (!strcmp(node_type, "struct_specifier"))
7      return true;
8    for (uint32_t i = 0; i < ts_node_child_count(node); i++)
9      if (check(ts_node_child(node, i))) return true;
10   return false;
11 }
12
13 int main(int argc, const char** argv) {
14     char *src = read_input(argc, argv);
15     TSParser *parser = ts_parser_new();
16     ts_parser_set_language(parser, tree_sitter_cpp());
17     TSTree *tree =
18       ts_parser_parse_string(parser, NULL, src, strlen(src));
19     if(!tree || ts_node_has_error(ts_tree_root_node(tree)))
20         return -1; // syntax error
21     TSNode root_node = ts_tree_root_node(tree);
22     for (uint32_t i = 0; i < ts_node_child_count(root_node);
23         i++) {
24       TSNode node = ts_node_child(root_node, i);
25       if(!strcmp(ts_node_type(node), "function_definition")) {
26         if(check(node)) {
27           // code region for processing local structs
28         }
29       }
30     }
31     // ...
32 }
```

Fig. 2: Motivating Example. A simplified C++ processor implemented via Tree-sitter. It has local struct processing logic (line 27) behind complex constraints, simulating compilers that apply certain optimizations only when the input contains local structs.

Unfortunately, these solutions fall short when programs become as complex as language processors. Taint tracking fails, because just after the first stage (*i.e.*, tokenization in Fig. 1), every input byte is applied some constraints and gets tainted, rendering taint tracking pointless. Symbolic and concolic fuzzing fail for several reasons. First, they require heavy engineering to model *every* operation semantics in the program (*e.g.*, native instruction semantics [90] or IR semantics [9, 12, 59, 78]) to track symbol propagation correctly. This process also faces difficulties in operations commonly seen in language processors (*e.g.*, symbolized pointers [12]). To demonstrate, Fig. 2 shows a simplified language processor for C++. Line 27 contains the code region that is only triggered when the input source contains structs inside functions. Our experiments show that state-of-the-art symbolic/concolic execution engines, including SymCC [59], Angr [78], and SymQemu [60], cannot track the input constraint leading to line 27 correctly due to incomplete semantics modeling. Second, even when semantics modeling is complete (via heavy manual effort), the enormous number of control paths in early language processor stages can cause path explosion for symbolic execution and over-constraining [12] for concolic execution. Additionally, certain stages might contain logic that defeats symbolic solvers, such as hash-based tokenization that cannot be inverted. Researchers have proposed symbolizing tokens instead of input bytes to mitigate this problem for Javascript [27]. However, its generalizability to other lan-

guages remains unclear [88], and it only mitigates the problem in the first stage, which does not handle constraints in later stages (*i.e.*, semantics checking and code generation).

Target-specific Solutions. Researchers have proposed solutions to overcome complex constraints in specific language processors. Csmith [88] generates C programs avoiding undefined behaviors. DIE [55] and Fuzzilli [30] maintain Javascript semantics to reach deep just-in-time (JIT) regions. These approaches require heavy effort to design target-specific heuristics, making them ungeneralizable. Furthermore, researchers have proposed annotation-based approaches [11, 89]. Still, they require expert knowledge of the annotation language and an understanding of the target constraints, making them non-trivial to apply to new targets. Additionally, token-level fuzzing [67] and grammar-level fuzzing [4, 80] have been proposed. They are semi-generic because they do not target a specific language due to the wide availability of tokenizers and grammar. However, they only try to overcome constraints in the first two stages (see Fig. 1), leaving the later stages unhandled.

To date, we face a dilemma where generic solutions fail to effectively handle language processors, while target-specific solutions struggle to generalize well.

2.4 LLMs and Challenges

Large Language Models. Large language models (LLMs) have gained wide attention in recent years due to their diverse applications. We think LLMs possess the potential to help fuzzers solve complex constraints based on the following observations. First, modern LLMs are trained on a sea of code bases and have shown capabilities in code understanding [16, 53] and reasoning [82]. Given proper context (*e.g.*, comments, relevant code, natural language instructions), LLMs can perform code completion and code refactoring. With such capabilities, LLMs can potentially understand the complex constraints from source code and generate solutions for them. The input to language processors also being in the form of source code further increases LLMs' potential in this task. Second, researchers have shown that pretrained LLMs can learn a new task using a few examples, *i.e.*, few-shot learning [8, 62]. LLMs may utilize the inputs discovered by fuzzers, which are known to cover certain code regions, as examples to learn how to cover new regions. Third, although LLMs are susceptible to reasoning errors [75], they can still be very useful. This is because verifying the LLM's solution through code coverage is cheap and simple. Successful solutions can potentially eliminate fuzzing roadblocks and boost code coverage, while failed ones can simply be discarded. These observations highlight LLMs' potential in helping fuzzers overcome complex constraints, and motivate us to explore deeper. However, working with LLMs differs from traditional program analysis techniques. In practice, we identify two unique challenges that should be addressed.

Constraint Prioritization. The first challenge we identify is *it is unclear how to prioritize unsolved constraints to maximize coverage gain.* Language processors can contain up to millions of lines of code (see Table 4). It is important to prioritize the unsolved constraints to solve (*i.e.*, the uncovered lines to cover), because inference with LLMs is not free [85, 86], and it is by far impractical to ask LLMs to reason about all unsolved constraints. However, constraint prioritization is non-trivial. An ideal prioritization strategy should satisfy the following two properties. First, it should prioritize the constraints having greater potential to reach more code regions if solved. For instance, in Fig. 2, consider a scenario where line 20 and line 27 are both uncovered. Since line 27 can lead to additional uncovered regions, such as deeper functions for processing local structs, while line 20 merely bails out, prioritizing reaching line 27 is more effective. Second, prioritization should account for the solving difficulty. LLMs may easily solve some constraints but fail entirely with others. Solving success rates can drop when constraints involve complex logic beyond the LLM's reasoning capabilities, or when they involve inherently hard-to-solve problems (*e.g.*, one-way functions). To remain cost-effective, constraints that LLMs struggle to solve should be deprioritized, even if they have the potential to cover more code regions. To date, effectively satisfying these two properties to maximize coverage gain remains an open challenge.

Context Granularity. After we decide which constraint to solve, the next step is to collect relevant context, structure it into a prompt, and query the LLM for results. To work with LLMs, we think the best context for a constraint is the relevant source code. However, *it is non-trivial to decide the source granularity to present to the LLM*. Presenting the entire code base is not cost-effective, considering the magnitude of modern language processor code bases (see Table 4). A smarter approach would be using dynamic program slicing [87, 94, 95] to extract only code relevant to the branching condition (which guards uncovered code regions) by setting the branching condition as the slicing criterion. However, we notice this approach is not ideal either. The dynamic slice of a language processor can still be prohibitively huge. In fact, setting any branching condition in one stage (see Fig. 1) as the slicing criterion should result in the majority of all previous stages being included in the dynamic slice. For instance, tokenization and parsing stages are both relevant to any branching condition in the code generation stage, as inputs failing to be tokenized or parsed can never trigger code generation. It is not ideal to present the entirety of the tokenizer and parser as context for constraints in the code generation stage. Additionally, even if the cost budget and LLM context window size support large slices, presenting redundant information can cause more hallucinations [6]. On the flip side, presenting only a narrow context, such as the function containing the uncovered code region (*i.e.*, the enclosing function), can miss necessary information for understanding the constraint

clearly. For instance, prior to reaching the enclosing function, there could be early constraint checks that reject the input. To conclude, how to present sufficient and concise context to achieve both effectiveness and efficiency remains unclear.

2.5 Our Insights and Solutions

We propose two solutions to solve the two aforementioned challenges respectively, *hybrid centrality prioritization* and *iterative context construction*. Next, we discuss our insights.

Hybrid Centrality Prioritization. As discussed in §2.4, an ideal prioritization algorithm should (i) prioritize constraints having greater potential to trigger more code regions and (ii) deprioritize constraints that LLMs may not be able to solve.

To address (i), in the field of fuzzing seed scheduling, researchers have faced similar challenges in measuring the potential to reach more code regions [7, 57, 69]. An effective solution is graph centrality analysis, a method to estimate a node's influence in a graph. Researchers have shown that after building the inter-procedural Control Flow Graph (CFG), centrality analysis can effectively measure one basic block's potential to reach more other basic blocks [69]. Inspired by this, we extend the existing approach and design a centrality analysis mechanism to prioritize unsolved constraints.

To address (ii), we need a way to measure the constraint solving difficulty. The first approach we consider is using graph complexity analysis [98] to approximate constraint complexity, which seems to correlate with solving difficulty. However, in reality, we notice that pretrained LLMs are capable of picking up hints in the context that immediately enable them to solve the constraints, without thoroughly analyzing the constraints in detail. For instance, in Fig. 2, imagine a scenario where lines after line 20 are not covered due to failure to pass the constraints evaluated at line 19. In our experiment, LLMs (*e.g.*, GPT-4o-mini) can draw insights from just the function names (*e.g.*, `ts_node_has_error`, `tree_sitter_cpp`) to realize that the input should be a syntax-correct C++ source, and then proceed to generate an input that successfully triggers line 21. However, graph complexity analysis would consider the parsing stage as a complex operation and neglect such hints. Moreover, asking the LLMs themselves to score each constraint isn't viable because it is neither cost-effective nor reliable (because it is hard to verify the scores). Instead, we propose approximating solving difficulty using past solving history: the more LLMs fail to solve a certain type of constraint, the harder it is for LLMs to solve similar constraints in the future. Finally, we use the approximated difficulty to adjust the centrality score, so that the final scores start high for constraints with higher centrality scores but are scaled down later if the constraints appear hard to solve.

Iterative Context Construction. Considering the "picking up hints" capability discussed above, we deem it impractical to decide how much context is sufficient for an LLM a priori. Instead, we draw insights from recent works on LLMs for software engineering [19, 93], which show that LLMs can improve and correct their code generation results iteratively with feedback information. To provide a sufficient and concise context, we start by presenting the LLM with a narrow context around the constraint evaluation point (*i.e.*, the enclosing function). We then instruct the LLM to either solve the constraint or request additional context (*e.g.*, symbol definitions). This approach allows us to leverage dynamic and static analysis techniques to iteratively expand the context. If the LLM provides a solution, we verify its correctness. In case of failure, the execution trace is analyzed to identify a failing point's context, which is then fed back to the LLM. When the LLM requests more context, source code analysis tools [50] are employed to supply the requested context. Through this process, we gradually construct the context until reaching a successful solution or an iteration threshold.

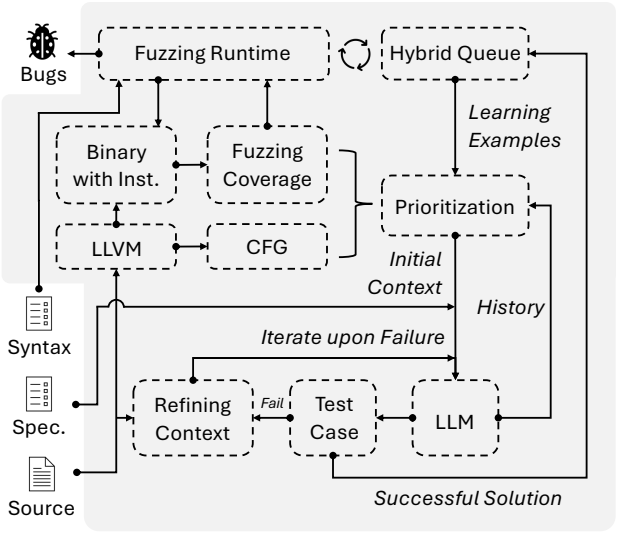

Fig. 3: Overview of HLPFuzz. HLPFuzz takes as input the source code, a short specification in natural language, and a syntax file. Afterward, it performs traditional fuzzing on the language processor, utilizing an LLM to solve complex constraints with advanced prioritization and context construction mechanisms. HLPFuzz outputs the bugs found along the process.

3 Overview of HLPFuzz

We integrate our solutions in a hybrid fuzzing framework, HLPFuzz (**H**ybrid **L**anguage **P**rocessor **Fuzz**er). Fig. 3 shows its overview. Overall, HLPFuzz extends a traditional greybox fuzzing runtime (containing the target executor, seed scheduler, mutator, and bug oracle) with a whitebox LLM-based solver. Guided by coverage feedback, the greybox fuzzing runtime quickly explores shallow code regions with syntax-aware mutations. The LLM-based solver then *HeLP*s overcome complex constraints to reach deeper code regions. To solve the challenges discussed in §2.4, HLPFuzz uses a novel constraint prioritization mechanism to select the con-

straint that maximizes coverage gain. Afterward, HLPFuzz iteratively constructs a sufficient and concise constraint context to promote a successful constraint solution. We discuss the details of our approaches in the following sections.

4 Hybrid Centrality Prioritization

In this section, we discuss how we design a constraint prioritization algorithm aiming to maximize the benefit from an LLM-based constraint solver.

4.1 Centrality Analysis

First, we want to measure an unsolved constraint's potential to reach more code regions if solved. To this end, we take insights from `K-Scheduler` [69], a centrality-based fuzzing seed scheduler for picking up the most rewarding seed for mutation. We use the same centrality measurement as the basis and extend it to suit our purposes. Next, we introduce the background and describe our approach.

Graph Influence Analysis. In graph analysis, it is a common task to identify the most influential nodes inside a graph. Centrality analysis aims to accomplish this by ranking nodes in a graph based on their positions. Multiple centrality analysis techniques exist [83]. Degree centrality measures the local influence of a node by counting its direct neighbors. Eigenvector centrality measures the global influence across the entire graph, but can lose accuracy on directed graphs (*e.g.*, CFGs) [46, 52]. Variants of eigenvector centrality, including pagerank centrality [84] and Katz centrality [34], can address this problem. Katz centrality proves to be more useful when applied to program CFGs, because pagerank centrality dilutes node influence by the number of its direct neighbors and can cause undesirable side effects [69]. The calculation of Katz centrality scores involves neighboring nodes. Depending on whether we use in-neighbors or out-neighbors, there are in-degree and out-degree Katz centrality. In-degree Katz centrality measures the influence from other nodes. Out-degree Katz centrality measures the influence exerted on other nodes, which can approximate the potential to reach more nodes. Next, we describe out-degree Katz centrality.

Katz Centrality. Let A denote the adjacency matrix of a directed graph G of n nodes. That is, $A_{ij} = 1$ when there is an edge from node i to j, and $A_{ij} = 0$ otherwise. Let C denote the Katz centrality vector of size n for all nodes in G. Then, C can be calculated as follows:

$$C_i = \alpha \sum_{j=1}^{n} A_{ij} C_j + \beta_i \tag{1}$$

where α is a weight $\in [0,1]$ and β_i can be used as a bias for each node. From the equation, we can see that C_i captures an equally weighted sum of its neighbors' centrality scores. The α term is a decay factor, so that nodes farther down the path receive less influence from the current node. The bias vector β can adjust each node's score based on external information. To efficiently calculate the scores, an iterative approach called the *power method* can be used under reasonable assumptions about the graph topology (*e.g.*, α being less than the multiplicative inverse of the largest eigenvalue [46, 52]). The power method initializes elements in C to 1, iterates them, and gradually converges to the final solution with $O(n)$ complexity.

Centrality for Coverage Potential Measurement. After building the inter-procedural CFG, we can use out-degree Katz centrality to measure one node's influence on other downstream nodes. The greater the influence, the more likely the node can reach more other nodes. Next, to measure the potential for a fuzzing mutator to gain more coverage after covering one node, we can utilize feedback from the fuzzer to adjust the centrality calculation. First, we can use the coverage feedback to remove all covered nodes from the CFG, so that the centrality score measures the influence only on the uncovered nodes, which are the targets for the mutator to cover next. Second, since not all nodes are equally difficult for a mutator to explore, historical mutation success rates from the fuzzer can be used to set the bias vector β, lowering C_i when node i appears hard to reach by the mutator. Following this approach, the resulting centrality score can well approximate the coverage increasing potential of an uncovered node, as evaluated by prior work [69].

Our goal is to maximize the coverage gain that the LLM can bring to the fuzzer by solving complex constraints. As previously discussed in §2.4, this includes two steps: (i) selecting the unsolved constraint with the most coverage increasing potential for the fuzzer and (ii) deprioritizing the constraints that appear hard for the LLM. We notice that step (i) can be solved using the centrality score of the basic block guarded by the constraint. However, step (ii) remains unhandled. Note that the bias vector β mentioned a priori does not solve this problem, because it measures how hard it is to solve the constraint for the mutator, not for the LLM. For step (ii), we propose using the LLM solving history to perform hybrid calibration.

4.2 Hybrid Calibration

As discussed in §2.5, it is challenging to predict how likely an LLM can solve a constraint beforehand. Instead, we calibrate the centrality score based on the LLM solving history. In general, if the LLM keeps failing to solve one constraint, then this constraint, along with similar ones, should be considered hard for the LLM and be deprioritized later.

Solution Categories. First, we conceptually divide a constraint of a target basic block into two parts. Part one is from the program entry to the basic block's enclosing function, and part two is from the beginning of the enclosing function to the basic block. We refer to part one as the *pre-constraint*, and part one+two as the *full constraint*. When verifying the solutions, we categorize them as follows:

- SC1: Does not solve the pre-constraint.
- SC2: Solves the pre-constraint, but not the full constraint.
- SC3: Solves the full constraint.

This categorization strategy is based on the following observation. When the solving history shows the LLM struggles with the pre-constraint for one basic block, it often suggests the enclosing function is hard to reach, and therefore the LLM is likely to face the same challenge solving the pre-constraints of other basic blocks in the same enclosing function. Upon failure, this observation enables us to calibrate the score for not only the target basic block itself, but also *similar* targets that share the same pre-constraints.

History Maps. We use history maps to store past solving history. Specifically, we use a map $PC_{success} : Func \rightarrow Int$ to track the successful solution count of the pre-constraint *w.r.t.* each function, $PC_{failure} : Func \rightarrow Int$ to track the failing count, and $FC_{failure} : ID \rightarrow Int$ to track the failing solution count of the full constraint *w.r.t.* the ID of the target basic block in the CFG. When a solution falls into SC1, $PC_{failure}$ and $FC_{failure}$ are increased by one. When a solution falls into SC2, $PC_{success}$ and $FC_{failure}$ are increased by one. For SC3, $PC_{success}$ is increased by one. We will soon explain why the successful count $PC_{success}$ is also stored.

Calibration. We perform centrality score calibration based on the history maps. Specifically, we introduce a scaling factor vector SF of size n for all nodes in the CFG. We let F_i denote the enclosing function of basic block i, and compute SF by setting $SF_i =$

$$\frac{1+\gamma \cdot PC_{success}(F_i)}{1+\gamma \cdot PC_{success}(F_i)+FC_{failure}(i)+PC_{failure}(F_i)} \quad (2)$$

where γ is a positive constant number. Afterward, SF is used to calibrate the original centrality score C in (1) to get the final score $Score_i = SF_i \cdot C_i$ used for prioritization.

We compute SF to achieve the following goals. First, it should keep the centrality score unchanged upon solving success, and scale down the centrality score upon failure. Therefore, when there is no failing solution, $SF_i = 1$, and when failing solutions appear, $SF_i \in (0,1)$. Second, $PC_{failure}(F_i)$ should affect all uncovered basic blocks in the function F_i. As discussed, this is because all uncovered basic blocks in F_i share the same pre-constraint. Therefore, as $PC_{failure}(F_i)$ increases, SF_j approaches zero for every node j in the same enclosing function $F_i (= F_j)$. Third, failure to solve the full constraint indicates the LLM's potential lack of ability to solve it again in the future. Therefore, as $FC_{failure}(i)$ increases, SF_i approaches zero. Finally, the LLM may sometimes fail to trigger one target basic block but succeed in triggering others in the same enclosing function, because slightly different prompts are provided (*e.g.*, different target lines and learning examples, which we will detail in §5.1). When this happens, we still treat the basic blocks in that enclosing function as promising, because the pre-constraint is successfully solved. Therefore, we weaken the failure count's influence on reducing SF by assigning more weight to the successful solving attempts on the pre-constraint (*i.e.*, $PC_{success}$) than to the failed ones (*e.g.*, $PC_{failure}$), using a constant $\gamma > 1$ determined based on empirical results.

Ranking. HLPFuzz uses the calibrated score to rank the uncovered basic blocks immediately following all *executed* constraint evaluation points (*e.g.*, `if-else` conditions, `switch` conditions). Focusing on executed constraint evaluation points comes with two benefits. First, it filters out the constraints that are non-solvable due to non-reachable code. For instance, if we set up a fuzzer to fuzz `Clang++` with the `-O1` optimization level, then uncovered `-O3` optimization regions are not reachable. Since they are never executed, they are filtered out and HLPFuzz does not waste time covering them. Second, HLPFuzz can locate a test case that triggers the constraint evaluation point. This test case is a solution that solves the pre-constraint, making it a great learning example for the LLM (more in §5.1). After ranking the targets, HLPFuzz uses the LLM to solve the top-n targets before waiting for the next round. We select $n = 10$ empirically for timely coverage synchronization and balancing the ranking computation cost.

To conclude, HLPFuzz calculates the centrality score and then utilizes solving history from the LLM for calibration. The resulting hybrid score not only captures the coverage increasing potential, but also accounts for the LLM's solving capability. HLPFuzz uses the hybrid score to rank the constraints to maximize the benefit of the LLM. We provide the pseudocode for the prioritization algorithm in Alg. 1.

5 Iterative Context Construction

In this section, we present an iterative algorithm to address the context granularity challenge discussed in §2.4. We first establish an initial context, and then use divergence analysis and language server integration to iteratively refine it.

5.1 Initial Context

The initial context includes a task description, a narrow context around the constraint evaluation point, and a learning example. We show its structure in Fig. 8.

Task Description. The task description clarifies the goal and rules of the constraint solving task. It includes the system prompt, the overall goal, the response format requirement, and the language processor specification. The language processor specification is an input to HLPFuzz. It is a *short* description of the target that can help the LLM understand the background better. This includes the expected input language (*e.g.*, C++), the harness setup (*e.g.*, the compilation options), and so on. In our evaluation, the specification contains 6 lines on average.

Narrow Context. We start with the enclosing function as the narrow context. We consider it an ideal option, because

functions provide the local context and are typically of manageable sizes based on our experiments (*e.g.*, C++ functions in the LLVM project have 19.24 LoC on average). We present the source code with line numbers, and highlight the target line, *i.e.*, the line containing the target uncovered basic block.

Learning Example. As discussed in §2.4, we can benefit from few-shot learning by showing the LLM an example solution. Modern fuzzers keep all test cases that can trigger new coverage in a fuzzing queue. Since HLPFuzz's prioritization targets only basic blocks behind executed constraint evaluation points, there must exist a test case in the fuzzing queue that can trigger the constraint evaluation point. This test case can be used as a great learning example, because it already solves the pre-constraint (*i.e.*, it triggers the enclosing function). To retrieve this test case, HLPFuzz maintains a data structure to track the first queue entry that triggers each basic block. After retrieving the test case, HLPFuzz performs test case minimization on it. This is because we notice mutators can generate large test cases that involve irrelevant information. We use `C-Reduce`'s method [64] to minimize the test case's size while maintaining its ability to trigger the constraint evaluation point. Finally, HLPFuzz includes the minimized test case at the end of the initial context and clarifies the line it can trigger.

After establishing the initial context, HLPFuzz sends it to the LLM and fetches the response. Upon receiving the response, HLPFuzz extracts the solution and verifies it by running the target with the solution and checking code coverage. If the target basic block is triggered, the solution is verified to be successful and is synchronized to the hybrid fuzzing queue, as shown in Fig. 3. If the target basic block is not triggered, it means the LLM fails to solve the constraint given the initial context. HLPFuzz then employs two strategies to refine the context. Next, we discuss the two strategies.

5.2 Divergence Analysis

When the solution falls into SC1 (see §4.2), HLPFuzz uses the stack trace of the learning example to locate the (diverging) function where the solution's execution diverges, because we notice that the diverging function can help the LLM capture missing information to solve the pre-constraints.

HLPFuzz achieves this by executing the target in debug mode. First, HLPFuzz sets a breakpoint BP_e at the beginning of the enclosing function. Then, HLPFuzz executes the target with the learning example, which is guaranteed to trigger the enclosing function. When BP_e is hit, HLPFuzz extracts the stack frames, which reveal a chain of functions from the program entry function to the enclosing function. We denote these functions as $F_0 ... F_{n-1}$, where F_0 is the program entry function and F_{n-1} is the enclosing function. Next, HLPFuzz restarts the debugging session and sets breakpoints $BP_0 ... BP_{n-1}$ at the beginning of $F_0 ... F_{n-1}$. Afterward, HLPFuzz re-executes the target with the failing solution provided

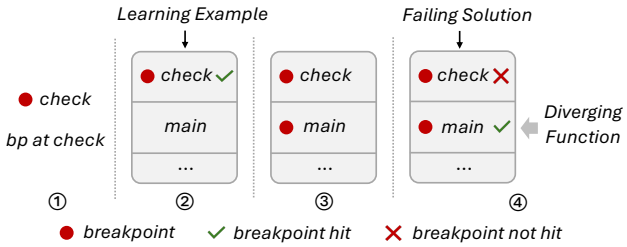

Fig. 4: Divergence Analysis Running Example. HLPFuzz uses divergence analysis to locate the diverging function, which is used to refine the initial context.

by the LLM, and monitors $BP_0 ... BP_{n-1}$ for their hitting status. When the target exits, HLPFuzz locates the first breakpoint BP_i that is not hit, and considers the function F_{i-1} as the diverging function (note the index difference). Lastly, HLPFuzz re-executes the target twice more: once with the learning example and once with the LLM's solution. The goal is to analyze the diverging line(s) in F_{i-1} by setting breakpoints on all its lines and comparing their hit status between the learning example and the solution. To illustrate, we show a running example of divergence analysis for the previous motivating example shown in Fig. 2. We assume a scenario where line 7 is the target uncovered line. HLPFuzz first creates an initial context using the function `check()` as the narrow context with a learning example that triggers the execution of `check()`. The learning example contains a `function`, but does not contain a `struct` within the function. After getting the initial context, the LLM infers that the argument `node` must contain a `struct`. However, since the initial context does not capture any pre-constraint, the LLM references the learning example, replaces the `function` with a `struct` and returns it as the solution, without realizing that the `function` has to exist to encapsulate the `struct`. Consequently, the LLM's solution does not pass verification, and HLPFuzz starts divergence analysis. As shown in Fig. 4, this involves four steps. ①: Setting a breakpoint BP_e at the enclosing function `check()`. ②: Executing the target with the learning example; Retrieving the functions F_0 = `main()`, F_1=`check()` from the stack frames when BP_e is hit. ③: Setting the breakpoints BP_0 at F_0 and BP_1 at F_1. ④: Re-executing the target with the LLM's solution and monitoring the breakpoint hitting status. In this case, HLPFuzz detects that BP_1 on `check()` is not hit, and therefore considers F_0 as the diverging function. Next, HLPFuzz provides the LLM with the source code of F_0. HLPFuzz also highlights the diverging line (*i.e.*, line 25) in F_0. Afterward, the LLM sees the pre-constraint at line 25, and corrects its solution by including a `struct` inside the `function`. Finally, HLPFuzz verifies the new solution as successful.

Additionally, HLPFuzz deduplicates the functions in the stack frames to handle recursive calls commonly seen in language processors. Using dynamic program analysis, divergence analysis helps HLPFuzz find the missing information to refine the initial context iteratively.

5.3 Language Server Integration

When the missing context does not reside in the functions from the stack frames, divergence analysis falls short. For instance, consider the scenario where line 27 in Fig. 2 is the target line to cover. For this scenario, HLPFuzz includes the source code of function `main()` in the initial context. However, without the implementation detail of function `check()`, the LLM cannot figure out the correct solution to cover line 27. To solve this problem, we gain insights from what a programmer would do to understand program context: looking for symbol definitions. Specifically, we provide the LLM access to a tool that can fetch the definition of a symbol *upon request*. This method avoids including all the referenced symbols as program slices do, which is impractical as previously discussed in §2.4. Instead, the LLM decides which symbol to look for. To realize this, HLPFuzz integrates a language server (*e.g.*, `clangd` [43]) for the source code via the Language Server Protocol (LSP) [50]. LSP is commonly used by modern program editors and IDEs to power language features such as auto-complete. Language server integration allows the LLM to actively search for missing context that divergence analysis cannot provide. In the aforementioned scenario, when provided access to the symbol lookup tool along with the initial context, an LLM (*e.g.*, GPT-4o-mini) can realize it should look for the definition of `check()` in order to solve the constraint for line 27. The LLM then requests to use the symbol lookup tool, and HLPFuzz uses LSP to talk to the language server to fetch the definition of `check()`. Afterward, the LLM has the sufficient context needed to solve the constraint.

Language server integration uses static analysis to complement divergence analysis and refine the initial context. With these two approaches, HLPFuzz iteratively constructs a sufficient and concise context until reaching a successful solution or a maximum number of iterations threshold. HLPFuzz currently uses a threshold of 5 based on empirical results. We show the effectiveness of the approaches in §7.3.

6 Implementation

We implement HLPFuzz with 4,885 lines of code in Python and C/C++. Table 3 shows a breakdown. HLPFuzz reuses most of AFL++ [21]'s fuzzing runtime, including the target executor, seed scheduler, mutator, and bug oracle, while modifying the coverage tracing part to realize queue entry tracking discussed in §5.1. HLPFuzz uses a syntax-aware mutator, which is an effective and generic solution adopted by many fuzzers [11, 32, 80, 89] to solve syntactic constraints. The mutator uses ANTLR4 [1] grammars due to ANTLR4's wide grammar availability [2]. CFG Extraction is implemented via LLVM Analysis Passes [61]. To ensure correct mapping from AFL++'s coverage bitmap to the CFG nodes, HLPFuzz uses the SanitizerCoverage [44] edge instrumentation mode, which offers collision-free coverage tracking. Using collision-free instrumentation enables HLPFuzz to efficiently track CFG coverage by directly mapping the bits in the AFL++ bitmap to nodes in the CFG. To translate CFG coverage to source code coverage, which is needed for context presentation, HLPFuzz utilizes the debug information that is by default available in AFL++ instrumented binaries. HLPFuzz uses `graph-tool` [56] to compute Katz centrality due to its great performance on large graphs. We set the constant $\alpha = 0.5$ in (1) and $\gamma = 3$ in (2) from empirical results. Similar to prior hybrid fuzzers [33, 90], we implement the `WaitForNoCovGain` function in Alg. 1 by checking coverage in a 90s window. HLPFuzz uses LLDB [73] for divergence analysis, which is around 10x faster than GDB on large binaries such as `Clang++`. HLPFuzz uses `clangd` [43] as the language server, and we evaluate it only on targets written in C/C++. HLPFuzz uses OpenAI's API for LLM access.

7 Evaluation

We evaluate HLPFuzz to answer the following questions:

- Can HLPFuzz be applied to different real-world programming languages and effectively detect new bugs in their language processors?
- How do the solutions proposed in §2.5 contribute to the performance of HLPFuzz?
- How does HLPFuzz compare to state-of-the-art tools?

7.1 Evaluation Setup

Benchmark. To evaluate the generalizability and effectiveness of HLPFuzz, we test it on 9 popular language processors of 7 programming languages according to their popularity and diversity in domains (*e.g.*, Javascript for web services, Fortran for scientific computing), aiming to hunt for real-world bugs. We test their latest versions and provide the details in Table 4. To understand the contributions of our solutions proposed in §2.5, we perform an in-depth ablation study on two compilers and two interpreters for four different languages, `Clang++` for C++, `Flang` for Fortran, `Hermes` for Javascript, and `PHP` for PHP. We continue to use the four targets to conduct thorough experiments comparing HLPFuzz with six state-of-the-art solutions, including generic constraint solving solutions `SymQEMU` [60] and REDQUEEN [5], generic syntax-aware fuzzers POLYGLOT [11] and Grammarinator [32], and target-specific fuzzers `YARPGen` [39] and `Fuzzilli` [30] that are specific to C/C++ and Javascript, respectively. For all the fuzzers we evaluate, we feed them with the same input (if they need one) collected from open-source repositories.

Runtime Setup. We perform the evaluations on a machine running Ubuntu 22.04.2 LTS with two AMD EPYC 7452 32-core Processors and 1,024GB RAM. For LLM usage, we use the GPT-4o-mini model to remain cost-efficient [54] and set the model's temperature to zero via the OpenAI API to reduce

```
subroutine pack_example()
  integer(8), parameter :: a(10) = 100
  integer(kind=16) :: mask(4) &
                    = (/ .true., .false., .true., .false. /)
  integer :: test_i1(2)
  b = pack(a, mask)
end subroutine
```

(a) Assertion Failure in `Flang` when processing `pack-mask`

```
template<typename... Args>
void func(Args... args) {
    // auto lambda = [&]() { // <- LLM's version
    auto lambda = [&](struct main* main) {
        return (args + ...);
    };
    lambda(); // Call the lambda to ensure it is invoked
}
int main() {
    // func(1, 2, 3); // <- LLM's version
    func(nullptr);
    return 0;
}
```

(b) Stack Overflow in `Clang++` when processing template parameters

Fig. 5: Case studies. Examples demonstrating HLPFuzz's ability to find real-world bugs. The examples are manually reduced for demonstration.

randomness. For the real-world bug-hunting evaluation, we tested 9 targets over varying durations, totaling approximately three months due to resource limits. For all remaining experiments, we run them for 24 hours, repeat them five times, and report the average results. Each experiment is conducted in a separate Docker container, limited to a single CPU core.

7.2 Generalizability and Effectiveness

HLPFuzz finds bugs in 9 language processors. We present the full bug list in Table 7. As of writing, HLPFuzz has discovered 52 bugs in the latest versions of the language processors, out of which 37 are confirmed by the developers, and 14 are fixed. HLPFuzz achieves these promising results with minimal target-specific heuristics, relying solely on specifications averaging six lines of natural language descriptions. The average cost of LLM querying is around $0.2/hour.

Next, we conduct case studies for real-world bugs found by HLPFuzz to understand its capabilities further. We have manually reduced the test cases for demonstration.

Case Study A. Fig. 5a shows a Fortran program that crashes `Flang` with an assertion failure. Assertion failures in language processors can lead to denial of service or miscompilation in non-debug versions [15], resulting in harmful misbehaving programs. The crash is related to constant processing in the `pack-mask` operation. Forming this operation is hard through pure syntax-aware mutation, because it requires a dependency relation and correct argument types. This makes it a complex constraint to solve. During fuzzing, this constraint is deemed important by hybrid centrality prioritization, and HLPFuzz successfully solves it with two iterations of context construction. Afterward, the fuzzer performs several mutations on the solution and quickly finds the crash. In our experiments, pure syntax-aware mutation cannot easily solve this constraint and consequently does not discover this bug. We show the detailed solving process in Fig. 11 and Fig. 12.

Case Study B. Fig. 5b shows a test case that crashes `Clang++` with a stack overflow. The bug originates in the code that jointly handles the parameter of C++ template functions and lambda functions. To trigger this bug, a lambda function has to be inside a template function with a parameter pack, which is a non-trivial constraint for traditional mutation to solve. During fuzzing, HLPFuzz's prioritization picks up the constraint that guards the processing logic for template parameter packs. After three iterations of context construction, the LLM obtains sufficient context and creates a test case that contains a template function enclosing a lambda function, and calls the template function with a parameter pack (see lines 2, 3, and 10). After a few mutations on the solution, HLPFuzz arrives at the final test case triggering the crash.

> In conclusion, HLPFuzz successfully finds 52 bugs in 9 popular real-world targets from 7 programming languages. Its effectiveness is acknowledged by both its strong bug-finding capability and bug confirmations.

7.3 Contributions of the Solutions

To understand the contributions of the solutions discussed in §4 and §5, we make five variations of HLPFuzz to perform ablation studies. We make HLPFuzz$^{!HCP}$ by replacing hybrid centrality prioritization with random selection, HLPFuzz$^{!ICC}$ by stripping iterative context construction and presenting only the initial context, and HLPFuzz$^{!!}$ by stripping both hybrid centrality prioritization and iterative context construction while keeping the LLM-based solver. Additionally, we make HLPFuzz$^{!HC}$ by stripping only hybrid calibration (§4.2), and HLPFuzz$^{!LSI}$ by stripping only language server integration (§5.3). We compare these variations with HLPFuzz, the full-fledged version, and AFL++ (with the same syntax-aware mutator), which is the baseline. We evaluate the fuzzers on the four real-world targets chosen in §7.1, and compare the fuzzing performance differences in terms of coverage, bug-finding capabilities, and solving success rates.

Coverage. We run all the fuzzer variants with the same input for 24 hours and compare their coverage to understand their abilities to discover program states. We report the number of edges tracked by the collision-free SanitizerCoverage instrumentation mode. As shown in Fig. 6, HLPFuzz finds on average 109%, 45.8%, 34.3%, 31.0% more edges than the baseline fuzzer AFL++ (syntax) on `Clang++`, `Flang`, `Hermes`, and `PHP`, respectively. In general, the edge discovering capability of HLPFuzz degrades when we strip off each solution, showing the contribution of each solution to HLPFuzz in unblocking the fuzzer and discovering more program states.

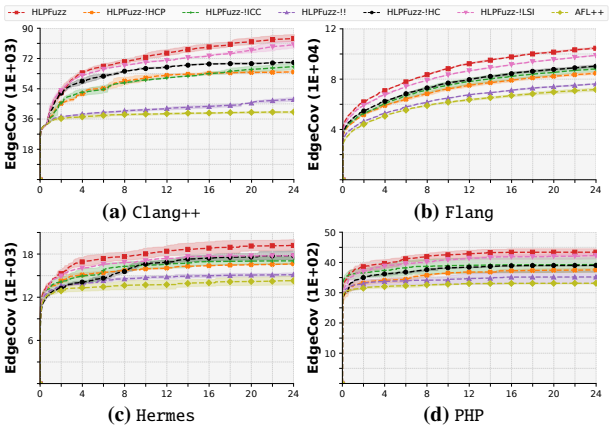

Fig.6: Edge coverage found by each version of HLPFuzz in 24h. We repeat the experiments five times and report the average results.

Specifically, by comparing HLPFuzz and HLPFuzz$^{!LSI}$, it shows that language server integration brings a coverage increase from 2.65% (on PHP) to 8.01% (on Hermes). Comparing HLPFuzz$^{!LSI}$ and HLPFuzz$^{!ICC}$ shows that divergence analysis brings a coverage increase from 4.21% (on Hermes) to 19.3% (on Clang++). Comparing HLPFuzz and HLPFuzz$^{!HC}$ shows that when centrality analysis is turned on, hybrid calibration brings a coverage increase from 8.02% (on Hermes) to 20.51% (on Clang++). Comparing HLPFuzz$^{!HC}$ and HLPFuzz$^{!HCP}$ shows that without hybrid calibration, centrality analysis brings a coverage increase from only 3.87% (on PHP) to 8.81% (on Clang++). This demonstrates that centrality analysis alone is limited in effectiveness for language processors due to its inability to account for the LLM's solving capabilities. Moreover, HLPFuzz$^{!!}$ only improves over AFL++ from 5.82% (on Hermes) to 18.57% (on Clang++), which is 28.48% to 90.43% lower than the coverage improvement HLPFuzz has over the baseline AFL++. This demonstrates that using LLMs to randomly solve constraints with the initial context brings limited coverage improvement, due to the challenges analyzed in §2.4.

Bug Finding. As shown in Table 1, HLPFuzz finds 10 bugs within 24 hours across the four targets, the highest among all the variants. In general, the bug-finding capability decreases when we strip off each solution. Without hybrid calibration, HLPFuzz$^{!HC}$ finds only 4 bugs, because the LLM can waste time solving constraints that are beyond its capability, leading to limited coverage gain. HLPFuzz finds two more bugs than HLPFuzz$^{!LSI}$, which is without language server integration. HLPFuzz$^{!ICC}$ finds only 2 bugs without the entire iterative context construction, because it fails to capture the sufficient context for the LLM to solve the constraint. Note that HLPFuzz$^{!!}$ finds the same number of bugs (two) as the baseline fuzzer AFL++, showing that the two challenges discussed in §2.4 hinder the effectiveness of the LLM.

Solving Success Rate. An interesting metric we also measure is the LLM's solving success rate. We calculate the solving success rate by dividing the number of successfully solved constraints by the total number of constraints the LLM has attempted to solve. As shown in Table 1, HLPFuzz achieves the highest success rates between 23.1% (on PHP) and 33.5% (on Clang++) among all the variants. HLPFuzz improves the success rate over HLPFuzz$^{!LSI}$ from 4.7% (on PHP) to 8.5% (on Hermes). This shows that language server integration helps improve the LLM's solving success rate, but has varying effectiveness on different targets. Similarly, HLPFuzz$^{!LSI}$ improves the success rate over HLPFuzz$^{!ICC}$ from 8.3% (on Hermes) to 18.4% (on Clang++), showing that divergence analysis helps improve the solving success rate as well. HLPFuzz improves the success rate over HLPFuzz$^{!HC}$ from 8.7% (on PHP) to 17.4% (on Clang++). This is because hybrid calibration learns from the LLM's solving history and actively avoids those constraints that are hard for the LLM to solve, thereby increasing the solving success rate. Finally, without our proposed solutions, HLPFuzz$^{!!}$ achieves the lowest success rates, showing the importance of both hybrid centrality prioritization and iterative context construction.

Table 1: The unique number of bugs found in 24 hours and the solving success rate of each fuzzer on four language processors. We run the fuzzers five times for 24h and report the average number of bugs (#) they find during this period. The bugs are manually deduplicated. The solving success rate (SR) is measured by the number of successfully solved constraints divided by the total number of constraints that the LLM has tried to solve.

	Clang++		Flang		Hermes		PHP	
	#	SR	#	SR	#	SR	#	SR
HLPFuzz	5	33.5%	4	26.7%	1	24.2%	0	23.1%
HLPFuzz$^{!HCP}$	2	17.8%	2	16.0%	0	16.1%	0	12.5%
HLPFuzz$^{!ICC}$	1	9.6%	1	8.0%	0	7.4%	0	7.1%
HLPFuzz$^{!!}$	1	8.4%	1	6.7%	0	6.5%	0	6.6%
HLPFuzz$^{!HC}$	2	16.1%	2	17.3%	0	15.1%	0	14.4%
HLPFuzz$^{!LSI}$	5	28.0%	2	19.1%	1	15.7%	0	18.4%
AFL++	1	⊖	1	⊖	0	⊖	0	⊖
Grammarin.	0	⊖	0	⊖	0	⊖	0	⊖
PolyGlot	1	⊖	1	⊖	0	⊖	0	⊖
Redqueen	0	⊖	0	⊖	0	⊖	0	⊖
SymQEMU	0	⊖	0	⊖	0	⊖	0	⊖
Fuzzilli	⊖	⊖	⊖	⊖	0	⊖	⊖	⊖
YARPGen	0	⊖	⊖	⊖	⊖	⊖	⊖	⊖

Internals. To understand how hybrid constraint prioritization works in practice, we randomly sample and examine 50 constraints selected by HLPFuzz$^{!HCP}$ (*i.e.*, no prioritization) and 50 selected by HLPFuzz. We find that 37 (74%) of the constraints selected by HLPFuzz$^{!HCP}$ guard regions that are CFG leaf nodes, which have limited potential[1] for improving coverage. An example is shown in Fig. 9a. In contrast, the constraints selected by HLPFuzz all guard CFG subtrees (*e.g.*, containing other branches and function calls), offering greater potential for coverage improvement. An example is shown in Fig. 9b. Thanks to prioritization and higher central-

[1] Excluding potential coverage gain from interprocedural data flows, a limitation discussed in §8.

ity scores, HLPFuzz focuses on constraints more likely to yield high returns. We also analyze the number of iterations required to solve constraints. On average, HLPFuzz takes 2.64 iterations to reach a successful solution, with a median of 2 to 3 across different targets. Additionally, only 5% (`Flang`) to 9.1% (`Hermes`) of the solutions succeed in a single iteration, indicating that iterative context construction improves solving success. Further details are provided in Table 6.

LLM Generalizability. We are also interested in how HLPFuzz performs across different LLMs. To evaluate this, we run HLPFuzz with two additional LLMs: llama3.1-70b-instruct and llama3.1-8b-instruct, which are two popular open-source models with different sizes and capabilities. According to community benchmarks [40, 41], the 70b model performs comparably to GPT-4o-mini, while the 8b model lags behind. The results (Table 5) show that HLPFuzz achieves similar, though slightly lower, performance improvement with llama3.1-70b-instruct (24.7% to 85.1% increase over the baseline), but yields more limited gains with llama3.1-8b-instruct (6.1% to 49.3%). Through inspection, we observe that the 8b model more frequently fails to follow instructions (e.g., returning responses in incorrect formats) and hallucinates (e.g., disregarding feedback from previous iterations), ultimately resulting in a lower success rate. Overall, the results show that HLPFuzz's effectiveness generalizes to different LLMs. However, a reasonably capable LLM is recommended, as failures to follow instructions and a higher likelihood of hallucinations can impair solution performance.

> In conclusion, the effectiveness of each solution varies on different language processors. Overall, HLPFuzz's success comes from all the proposed solutions.

7.4 Comparison with Existing Tools

In this section, we compare HLPFuzz with six state-of-the-art works regarding code coverage and bug detection capabilities. We compare HLPFuzz to two generic constraint solving solutions: `SymQEMU` [60], which employs concolic execution, and RedQueen [5], which uses the input-to-state heuristic to solve constant-comparison-based constraints. We also compare HLPFuzz with two generic syntax-aware fuzzers, PolyGlot [11], which is also mutation-based, and Grammarinator [32], which is a generation-based fuzzer that utilizes ANTLR4 grammars. Note that PolyGlot employs human-written semantics annotations for Javascript. Additionally, on `Clang++`, we compare HLPFuzz to `YARPGen` [39], a random program generator that embeds heavy human-written heuristics for C++. On `Hermes`, we compare HLPFuzz to `Fuzzilli` [30], a fuzzer that specializes in Javascript engine fuzzing. We provide the fuzzers with the same initial corpus (if required) collected from the Internet.

Coverage. We run all the fuzzers for 24 hours and use the same instrumented binaries to measure the edge coverage of their fuzzing queues. As shown in Fig. 7, HLPFuzz achieves the best edge coverage in 24 hours in all targets. HLPFuzz achieves 90.34%, 40.71%, 35.28%, and 7.18% more edge coverage than the second-best fuzzer on `Clang++`, `Flang`, `PHP`, and `Hermes` respectively. This performance gain demonstrates HLPFuzz's contribution to the language processor fuzzing venue. Note that HLPFuzz improves over `Fuzzilli` and PolyGlot on `Hermes` by only 7.18% and 14.53%. This is because PolyGlot employs human-written annotations and `Fuzzilli` embeds hard-coded JavaScript semantics, which all help them overcome many semantic constraints to reach more program states. `SymQEMU` does not show promising results on the four language processors. Our manual investigation shows that its symbolic executor can barely generate any solutions for the four targets, likely due to its inability to perfectly model the operation semantics, as discussed in §2.3. In conclusion, HLPFuzz outperforms existing solutions in terms of code coverage by varying margins.

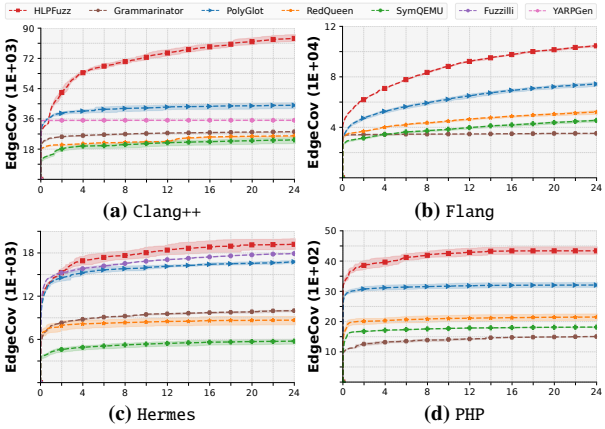

Fig. 7: Edge coverage comparison with existing fuzzers in 24h.

Bug Finding. We also compare HLPFuzz with the six fuzzers in terms of bug-finding capabilities. All fuzzers are run for 24 hours, and we manually deduplicate the bugs they find. As shown in Table 1, HLPFuzz discovers 10 bugs in total, with 5, 4, and 1 bug(s) in `Clang++`, `Flang`, and `Hermes`, respectively. Among existing fuzzers, only PolyGlot finds one bug in `Clang++` and one in `Flang`, both of which are also detected by HLPFuzz and the baseline AFL++ (syntax-aware). Our manual investigation reveals that these bugs occur in the parsing stage, which requires satisfying only syntax constraints. Therefore, these syntax-aware fuzzers detect them. The remaining fuzzers fail to detect any bugs within 24 hours, likely due to limited ability to effectively explore program states, as illustrated in Fig. 7. Leveraging the two proposed solutions, HLPFuzz utilizes the LLM to explore deeper states and uncover more bugs compared to existing tools.

Internals. We evaluate whether the constraints selected by HLPFuzz (i.e., the 50 randomly sampled in §7.3) can be solved by existing tools. Specifically, we assess: (1) how

Table 2: Solving details on the randomly sampled constraints. #1: the number of constraint evaluation points reached by the fuzzers (*e.g.*, reaching the if-else condition check). #2: the number of constraints solved by the fuzzers (e.g., reaching the if-else body).

	Clang++		Flang		Hermes		PHP		%
	#1	#2	#1	#2	#1	#2	#1	#2	
HLPFuzz	13	4	13	3	13	3	11	2	24%
Grammarin.	3	0	2	0	2	0	2	0	0%
PolyGlot	12	1	11	0	11	1	10	0	4%
Redqueen	9	0	8	0	6	0	7	0	0%
SymQEMU	9	0	8	0	6	0	7	0	0%
Fuzzilli	⊖	⊖	⊖	⊖	7	2	⊖	⊖	15.4%
YARPGen	6	1	⊖	⊖	⊖	⊖	⊖	⊖	7.7%

many constraint evaluation points each fuzzer reaches, and (2) how many constraints are solved. Table 2 summarizes the results. PolyGlot, Redqueen, and SymQEMU reach a similar number of constraint evaluation points as HLPFuzz, likely because they share the same initial corpus. However, all three underperform in solving the constraints. The rest fuzzers reach fewer evaluation points, likely due to their generative nature without an initial corpus (e.g., Grammarinator and YARPGen) or the use of a different embedded corpus (e.g., Fuzzilli). Overall, existing tools struggle to solve the constraints identified by HLPFuzz. Those that solve some constraints do so at lower rates than HLPFuzz.

Case Studies. We analyze why the case studies in Fig. 5 are missed by existing tools. The baseline fuzzer (AFL++ syntax-aware) fails to uncover the first bug (Fig. 5a) due to the roadblocking (*i.e.*, unsolved) `pack-mask` constraint. Because the operation depends on argument types and a specific dependency relation, random syntax mutations are ineffective in generating a valid test case to trigger the bug. The second bug (Fig. 5b) is missed by both the baseline fuzzer and YARPGen. This constraint requires a lambda function enclosed within a template function that uses a parameter pack. Since the baseline fuzzer's initial corpus lacks lambda functions, syntax-based mutations fail to generate a satisfying test case. YARPGen misses this bug as it focuses on loop constructs and does not model templates or lambda expressions. Redqueen and SymQEMU miss both bugs due to their heuristics (*i.e.*, input-to-state and symbolic execution) being limited in solving constraints in the later stages of a language processor.

Mutation Guidance v.s. Bugs. Similar to concolic fuzzers, HLPFuzz solves roadblocks to achieve higher coverage. However, this approach may be limited in discovering bugs that require guided mutation, such as preserving semantics correctness [11, 30] or targeting specific structures (*e.g.*, loops [39]). We study two cases for illustration. First, HLPFuzz does not find any bug in the latest V8 [76], a popular JavaScript engine. Since none of the evaluated tools uncover bugs in the latest version, we downgrade V8 to earlier versions, run each tool for 24h until bugs appear, and compare the results. In one earlier version [13], Fuzzilli discovers a legacy bug that HLPFuzz misses. The triggering test case requires a sequence of correctness-preserving mutations (see Fig. 10a). Given the initial test case (*i.e.*, line 1), HLPFuzz utilizes the LLM to generate a class inheritance structure (mutation 1) based on prioritization but does not treat further steps as roadblocks. HLPFuzz then falls back on its syntax mutator, which has a much lower correctness rate (4.7%) compared to Fuzzilli's IR-based approach (60.54%). Although HLPFuzz achieves comparable code coverage (EdgeCov 125483 vs. Fuzzilli's 135347.5), it fails to find this intricate bug due to the lack of guided mutation. Second, HLPFuzz does not detect optimization errors in Clang++, a bug type that YARPGen is designed to uncover. After reverting Clang++ to an old version [45], YARPGen finds a loop optimization error (see Fig. 10b) that HLPFuzz misses. HLPFuzz misses it due to: (1) lacking a logic error oracle; (2) mutations being not correctness-preserving (required to generate executable C++ programs); and (3) not focusing on loop structures. Therefore, although roadblock-solving is effective in generating complex structures that boost coverage and expose bugs, it is insufficient in finding certain bugs that require more targeted/guided mutation. On this matter, target-specific solutions with deeper domain knowledge continue to have an advantage.

> In conclusion, HLPFuzz generally outperforms existing constraint-solving solutions in terms of code coverage and bug-finding capabilities, despite limitations arising from the lack of mutation guidance. Meanwhile, future practitioners should consider combining LLMs with domain knowledge, rather than blindly relying on them, to achieve more intricate bug detection.

8 Discussion

8.1 Static Analysis Limitations

Although HLPFuzz achieves decent fuzzing performance, it is limited by the current static analysis approach. For example, the lack of interprocedural dataflow awareness can lead to imprecise prioritization scores. This could be improved by incorporating dataflow information into prioritization. Moreover, the current design of divergence analysis provides context only along the execution path of the learning example. If other valid paths exist to satisfy the constraint, divergence analysis cannot capture them. We believe this can be improved by applying reachability analysis to identify all potential paths from the program entry to the enclosing function. HLPFuzz could then perform divergence analysis along these paths to include their contexts, potentially improving performance.

8.2 Application to Other Targets

HLPFuzz is designed to address challenges specific to programming language processors. Its applicability to other sys-

tems that consume textual input has remained to be explored. For targets that consume binary input and involve less complex constraints (*e.g.*, `Binutils` [23]), HLPFuzz may not achieve the same satisfying results. Since modern LLMs are not trained on binary inputs, it becomes non-trivial to provide LLMs with learning examples, and it is unclear if we can effectively retrieve solutions in binary form. However, recent advancements in fine-tuning and adapting LLMs to specialized domains [20, 37, 72] have shed some light on LLMs' potential to handle binary formats, which may help HLPFuzz generalize to targets consuming binary input as well. We leave this as a future research direction.

9 Related Work

9.1 General Fuzzing Techniques

Fuzzing techniques include generation-based techniques and mutation-based ones. Generation-based ones create test cases according to a pre-defined input format [58, 68, 77], which does not require a seed corpus. Mutation-based ones mutates existing test cases [21, 36, 91], which rely on an initial seed corpus and uses code coverage feedback as guidance to cover more code. To improve mutation-based fuzzing, researchers have explored ways to generate better initial seed corpora. Skyfire [79] generates seeds from a grammar learned from data. TensileFuzz [38] generates seeds based on the relation among input fields. Better feedback mechanisms are also proposed. DDFuzz [48] incorporates coverage in the data dependency graph to guide fuzzing. MemFuzz [14] utilizes memory accesses. Traditional mutations may fall short in overcoming complex constraints, and hybrid fuzzing [59, 60, 90] is proposed to help the fuzzer make progress with external tools. HLPFuzz is a mutation-based hybrid fuzzer that uses an LLM to address the challenges in language processor fuzzing. It benefits from advancements in seed generation and feedback mechanisms as well.

9.2 Language Processor Fuzzing

Language processors are unique targets for fuzzing. They process inputs in multiple stages, each imposing different constraints on the input. To solve these constraints, there are generic solutions mentioned earlier, and target-specific solutions. Target-specific solutions model the target's semantics to generate better test cases. `DIE` [55] embeds heuristics to test Javascript just-in-time (JIT) engines. `YARPGen` [39] models C++ semantics to produce high-quality test cases for C++ compilers. These solutions show promising results on their respective targets but lack generalizability to other targets due to their reliance on target-specific heuristics. Annotation-based solutions [11, 89] aim to mitigate this problem by introducing an annotation language to specify the semantics. However, they still require expert knowledge of both the target's semantics and the annotation language itself, limiting their generalizability. Token-level fuzzing [67] and grammar-level fuzzing [4, 32, 80] are semi-generic solutions because they do not target a specific language. However, they only aim to solve constraints in the tokenization and parsing stages. HLPFuzz is designed to break this dilemma. By utilizing an LLM, HLPFuzz can solve complex constraints with minimal reliance on target-specific heuristics.

9.3 LLM-Assisted Techniques

Recently, LLMs have been utilized to handle various fuzzing tasks. TitanFuzz [18] uses LLMs to generate seeds for fuzzing deep learning libraries. Fuzz4All [85] combines both heavy and light LLMs to perform seed generation and mutation. ChatAFL [49] uses LLMs to analyze protocol specifications and generate machine-readable information that can be used for protocol fuzzing. PromtFuzz [47] and OSS-Fuzz [29] use LLMs to construct fuzzing harnesses for library code. HLPFuzz, on the other hand, focuses on utilizing an LLM to solve hard-to-solve constraints, which is orthogonal research that can potentially benefit from the above approaches. Additionally, LLMs have been used to aid other software analysis tasks. LLM4Decompile [72] uses LLMs to assist binary code decompilation. LDB [96] uses LLMs as debuggers to improve the quality of code generation. The thriving trend of such types of work shows the promising direction of utilizing LLMs to renovate traditional software analysis tasks.

10 Conclusion

In this work, we systematically analyze the limitations of existing constraint-solving approaches in language processor fuzzing. We explore the potential of utilizing LLMs for constraint solving, where we identify unique challenges and propose effective solutions in a hybrid fuzzing framework, HLPFuzz, utilizing an LLM to solve complex constraints to make progress. HLPFuzz achieves up to a 190.34% coverage improvement compared to state-of-the-art fuzzers and discovers 52 real-world bugs in 9 popular language processors, highlighting its generalizability and effectiveness.

11 Acknowledgment

This material was supported in part by the National Science Foundation (NSF) under Grant No. 2229876, the Defense Advanced Research Projects Agency (DARPA) under Contract No. N6600121-C-4024, and by funds provided by the NSF, the Department of Homeland Security, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DARPA, or their federal and industry partners.

12 Ethics considerations

The authors conducted all experiments to evaluate HLPFuzz without other human subjects. All the bugs found by HLPFuzz were discovered in a local environment without affecting any real-world services. We have responsibly reported all the bugs to the developers of each language processor by following their security reporting procedures.

13 Open Science

We make the following artifacts available: the source code of HLPFuzz, an example Dockerfile that instruments the target and configures HLPFuzz, and documentation for running HLPFuzz end-to-end with a specification, grammar, and seeds collected from the Internet. These artifacts are available at: https://doi.org/10.5281/zenodo.15606060.

References

[1] ANTLR. ANTLR v4. https://github.com/antlr/antlr4, 2024. Accessed: 2024-12-30.

[2] ANTLR. Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4, 2024. Accessed: 2024-12-40.

[3] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[6] Catarina G Belem, Pouya Pezeskhpour, Hayate Iso, Seiji Maekawa, Nikita Bhutani, and Estevam Hruschka. From single to multi: How llms hallucinate in multi-document summarization. *arXiv preprint arXiv:2410.13961*, 2024.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM, 2017.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 711–725. IEEE Computer Society, 2018.

[11] 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 *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 642–658. IEEE, 2021.

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278. ACM, 2011.

[13] Chromium. Fix Fuzzilli integration when Workers are created. https://chromium.googlesource.com/v8/v8/+/e0399e4394c4179fc24eae038f43f4ac5e3c1b62, 2025. Accessed: 2025-05-14.

[14] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 48–58. IEEE, 2019.

[15] The MITRE Corporation. CWE-617: Reachable Assertion. https://cwe.mitre.org/data/definitions/617.html, 2024. Accessed: 2024-12-30.

[16] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. Github copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734, 2023.

[17] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[18] Yinlin 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 René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 423–435. ACM, 2023.

[19] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 31–53. IEEE, 2023.

[20] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do llms really do their job? In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[21] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop*

on Offensive Technologies, WOOT 2020, August 11, 2020. USENIX Association, 2020.

[22] 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, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1051–1065. ACM, 2022.

[23] Inc. Free Software Foundation. GNU Binutils. https://www.gnu.org/software/binutils/, 2024. Accessed: 2024-12-30.

[24] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[25] go. Miscompilation: broken write barrier. https://github.com/golang/go/issues/71228, 2024. Accessed: 2024-12-30.

[26] go. Miscompilation: for range loop reading past slice end. https://github.com/golang/go/issues/40367, 2024. Accessed: 2024-12-30.

[27] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 206–215. ACM, 2008.

[28] Google. Honggfuzz. https://google.github.io/honggfuzz/, 2016. Accessed: 2023-09-14.

[29] Google. Leveling Up Fuzzing: Finding more vulnerabilities with AI. https://security.googleblog.com/2024/11/leveling-up-fuzzing-finding-more.html, 2024. Accessed: 2024-12-30.

[30] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[31] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 49–64. USENIX Association, 2013.

[32] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir, editors, *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pages 45–48. ACM, 2018.

[33] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. Evaluating and improving hybrid fuzzing. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 410–422. IEEE, 2023.

[34] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[35] Yu Liang, Song Liu, and Hong Hu. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 4309–4326. USENIX Association, 2022.

[36] libFuzzer. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2015. Accessed: 2023-09-14.

[37] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. Harnessing the power of LLM to support binary taint analysis. *CoRR*, abs/2310.08275, 2023.

[38] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. Tensilefuzz: facilitating seed input generation in fuzzing via string constraint solving. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 391–403. ACM, 2022.

[39] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proc. ACM Program. Lang.*, 7(PLDI):1826–1847, 2023.

[40] LLMStats. GPT-4o mini vs Llama 3.1 70B Instruct. https://llm-stats.com/models/compare/gpt-4o-mini-2024-07-18-vs-llama-3.1-70b-instruct, 2025. Accessed: 2025-05-14.

[41] LLMStats. GPT-4o mini vs Llama 3.1 8B Instruct. https://llm-stats.com/models/compare/gpt-4o-mini-2024-07-18-vs-llama-3.1-8b-instruct, 2025. Accessed: 2025-05-14.

[42] LLVM Project. Basic Block Detailed Description — Clang Documentation. https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html#details, 2024. Accessed: 2024-12-30.

[43] LLVM Project. Design of clangd. https://clangd.llvm.org/, 2024. Accessed: 2024-12-30.

[44] LLVM Project. SanitizerCoverage — Clang Documentation. https://clang.llvm.org/docs/SanitizerCoverage.html, 2024. Accessed: 2024-12-30.

[45] LLVM Project. Commit 2d77b27. https://github.com/llvm/llvm-project/commit/2d77b272a8f9b5b89b022628ca30b6b896a8f725, 2025. Accessed: 2025-05-14.

[46] Linyuan Lü, Duanbing Chen, Xiao-Long Ren, Qian-Ming Zhang, Yi-Cheng Zhang, and Tao Zhou. Vital nodes identification in complex networks. *CoRR*, abs/1607.01134, 2016.

[47] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3793–3807. ACM, 2024.

[48] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with Data Dependency Information. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pages 286–302. IEEE, 2022.

[49] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[50] Microsoft. Language Server Protocol. https://microsoft.github.io/language-server-protocol/, 2024. Accessed: 2024-12-30.

[51] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 67–82. USENIX Association, 2009.

[52] Eisha Nathan and David A. Bader. A dynamic algorithm for updating katz centrality in graphs. In Jana Diesner, Elena Ferrari, and Guandong Xu, editors, *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, Sydney, Australia, July 31 - August 03, 2017*, pages 149–154. ACM, 2017.

[53] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May*

23-24, 2022, pages 1–5. ACM, 2022.

[54] OpenAI. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, 2024. Accessed: 2024-12-30.

[55] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1629–1642. IEEE, 2020.

[56] Tiago P. Peixoto. The graph-tool python library. *figshare*, 2014.

[57] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 697–710. IEEE Computer Society, 2018.

[58] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based Whitebox Fuzzing for Program Binaries. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 543–553. ACM, 2016.

[59] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 181–198. USENIX Association, 2020.

[60] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[61] LLVM Project. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#introduction, 2024. Accessed: 2024-12-30.

[62] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[63] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[64] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012.

[65] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[66] Rust. Miscompilation: Memory unsafety problem in safe Rust. https://github.com/rust-lang/rust/issues/69225, 2024. Accessed: 2024-12-30.

[67] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-level fuzzing. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2795–2809. USENIX Association, 2021.

[68] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. SQLsmith. https://github.com/anse1/sqlsmith, 2016. Accessed: 2023-09-14.

[69] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2194–2211. IEEE, 2022.

[70] SQLancer. https://github.com/sqlancer/sqlancer, 2020.

[71] 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 *NDSS*, volume 16, pages 1–16, 2016.

[72] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 3473–3487. Association for Computational Linguistics, 2024.

[73] The LLDB Team. The LLDB Debugger. https://lldb.llvm.org/index.html, 2024. Accessed: 2024-12-30.

[74] The STP Project. The Simple Theorem Prover. https://stp.github.io/, 2024. Accessed: 2024-12-30.

[75] Armin Toroghi, Willis Guo, Ali Pesaranghader, and Scott Sanner. Verifiable, debuggable, and repairable commonsense logical reasoning via llm-based theory resolution. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 6634–6652. Association for Computational Linguistics, 2024.

[76] V8. What is V8? https://v8.dev/, 2025. Accessed: 2025-05-14.

[77] Spandan Veggalam, Sanjay Rawat, István Haller, and Herbert Bos. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 581–601. Springer, 2016.

[78] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 8–9. IEEE Computer Society, 2017.

[79] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[80] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 724–735. IEEE / ACM, 2019.

[81] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 497–512. IEEE Computer Society, 2010.

[82] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[83] Wikipedia. Centrality — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Centrality&oldid=1259333256, 2025. [Online; accessed 14-January-2025].

[84] Wikipedia. PageRank — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=PageRank&oldid=1268143105, 2025. [Online; accessed 14-January-2025].

[85] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 126:1–126:13. ACM, 2024.

[86] Chunqiu Steven Xia and Lingming Zhang. Automated program repair

via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 819–831. ACM, 2024.

[87] Aashish Yadavally, Yi Li, and Tien N. Nguyen. Predictive program slicing via execution knowledge-guided dynamic dependence learning. *Proc. ACM Softw. Eng.*, 1(FSE):271–292, 2024.

[88] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

[89] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. Towards generic database management system fuzzing. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[90] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.

[91] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, 2019. Accessed: 2023-09-14.

[92] ZERO DAY INITIATIVE. THE STORY OF TWO WINNING PWN2OWN JIT VULNERABILITIES IN MOZILLA FIREFOX. https://www.thezdi.com/blog/2019/4/18/the-story-of-two-winning-pwn2own-jit-vulnerabilities-in-mozilla-firefox, 2019. Accessed: 2024-12-30.

[93] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787. Association for Computational Linguistics, 2023.

[94] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In William W. Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 94–106. ACM, 2004.

[95] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 319–329. IEEE Computer Society, 2003.

[96] Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 851–870. Association for Computational Linguistics, 2024.

[97] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 955–970. ACM, 2020.

[98] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In *ISSRE 2007, The 18th IEEE International Symposium on Software Reliability, Trollhättan, Sweden, 5-9 November 2007*, pages 227–236. IEEE Computer Society, 2007.

A Appendix

Algorithm 1: Hybrid Centrality Prioritization

Data: CFG G, Coverage Cov, PCs, $FC_{success}$

$FC_{success} \leftarrow PC\text{s} \leftarrow Zeros$;
WaitForNoCovGain(Cov);
while *FuzzerIsRunning()* **do**
  G' = RemoveCoveredNodes(G, Cov);
  C = ComputeKatzCentrality(G');
  $Score$ = Calibrate(C, $FC_{success}$, PCs);
  LLM_Solve ← Rank(G, G', $Score$);
  // Solution Verification and Synchronization;
  // Updating $FC_{success}$ and PCs;
  WaitForNoCovGain(Cov);

System Prompt: *You are an experienced security engineer.*

Goal: I'm fuzzing a language processor, but the fuzzer is blocked by complex constraints. Your goal is to generate an input that satisfies the constraint to unblock the fuzzer.

Response Format: You should provide the input in the format of ...

Language Processor Description: It is a compiler that accepts C++ source as input. The enabled compilation options are ...

Task Description

This is the source code to the function containing the evaluation of the complex constraint. The target line is <LINE_NUM> ...

Narrow Context

This is a test case that can drive the execution to the evaluation point at line <LINE_NUM> ...

Learning Example

Fig. 8: Initial Context. The initial context consists of three parts: the task description, a narrow context, and a learning example. The prompts are simplified for illustration.

Table 3: Lines of code for a breakdown of HLPFuzz, summing up to 4,885 lines. As we integrate with AFL++, for Fuzzing Runtime, we only count the code that we add into AFL++.

Module	Language	LOC
Prioritization + Context Construction	Python	1,564
Hybrid Synchronization	Python	611
CFG Extraction (LLVM Passes)	C++	501
Fuzzing Runtime (Modified AFL++)	C	162
Unit Tests	Python	1,026
Utils	Python	1,021
Total	Python/C++/C	**4,885**

```
function a() { }
// mutation 1
class b extends a {
  constructor() {
    super()
    // mutation 2
    function c() {
        // mutation 3
        f = new BigUint64Array(256)
        // mutation 4
        for (d in f) {
            async function e() { g }
            e()
        }
    }
    // mutation 5
    new Worker(c, { type: "function" })
  }
}
// mutation 6
new b
// mutation 7: CRASH!
new b
```

(a) `DCHECK Failure` in `V8` in worker termination logic found by `Fuzzilli`. The test case was generated via multiple correctness-preserving mutations.

```
#include <stdio.h>
int var_3 = 24; bool var_23 = 1;
int arr_12 [25]; unsigned short arr_13 [25];

void test() {
#pragma clang loop vectorize_predicate(enable)
  for (char a = 4; a < var_3; a++) {
    arr_13[a] = arr_12[a - 3];
    var_23 = arr_12[a - 1];
  }
}

int main() {
  for (size_t i_0 = 0; i_0 < 25; ++i_0)
    arr_12 [i_0] = 1;
  test();
  // the printed values differ between -O0 and -O1
  printf("%d\n", (int)var_23);
}
```

(b) Loop optimization error in `Clang++` found by `YARPGen`. The test case was generated by maintaining semantics correctness and focusing on loops.

Fig. 10: Mutation Guidance Case Studies. Legacy bugs found by existing tools but missed by HLPFuzz demonstrate the utility of mutation guidance. The examples are reformatted for demonstration.

Table 4: Real-world targets we test HLPFuzz on. # is the TIOBE index ranking that measures language popularity. The Version column is either the release version or the git commit hash.

#	Language	Target	Version	LoC(K)
2	C++	`Clang++`	19.1.0	5247
4	C	`Clang`	19.1.0	5247
6	Javascript	`Hermes`	0.13.0	3019
		`ChakraCore`	13358c6	878
		`njs`	0.8.8	115
10	Fortran	`Flang`	19.1.0	6660
13	PHP	`PHP`	8.4.2	1310
27	Lua	`lua`	5.4.7	25
39	Solidity	`solc`	0.8.28	189

Table 5: HLPFuzz's performance across different LLMs. G-4-M: gpt-4o-mini. L-70b: llama3.1-70b-instruct. L-8b: llama3.1-8b-instruct. SR: solving success rate.

Target		G-4-M	L-70b	L-8b
`Clang++`	EdgeCov	83908	74435	60037
	SR	33.5%	22.4%	16.3%
	#Bugs	5	4	1
`Flang`	EdgeCov	104573	93461	81724
	SR	26.7%	18%	15.3%
	#Bugs	4	4	2
`Hermes`	EdgeCov	19198	18001	16524
	SR	24.2%	17.5%	15.8%
	#Bugs	1	1	0
`PHP`	EdgeCov	4339	4131	3514
	SR	23.1%	15.7%	6.3%
	#Bugs	0	0	0

Table 6: Number of iterations for successful solutions.

	`Clang++`	`Flang`	`Hermes`	`PHP`
Average	2.75	2.8	2.48	2.63
Median	3	3	2	2.5

```
void Parser::ParseDirectNewDeclarator(Declarator &D) {
  // ...
  ExprResult Size =
    First ? (Tok.is(tok::r_square) ?
        ExprResult() : ParseExpression())
    : ParseConstantExpression();  // <--
        // -- the randomly selected constraint
  // ...
}
```

(a) An example of a randomly picked constraint that isn't rewarding to solve. The second expression in a ternary statement is selected as the target constraint to solve, but it has limited potential for increasing coverage.

```
if (isa<CXXDestructorDecl>(M)) { // <---- the prioritized constraint
  auto Check = [](QualType T, auto &&Check) -> bool {
    const CXXRecordDecl *RD =
        T->getBaseElementTypeUnsafe()->getAsCXXRecordDecl();
    if (!RD || !RD->isCompleteDefinition())
      return true;
    if (!RD->hasConstexprDestructor())
      return false;
    QualType CanUnqualT = T.getCanonicalType().getUnqualifiedType();
    for (const CXXBaseSpecifier &B : RD->bases()) {
      if (B.getType().getCanonicalType().getUnqualifiedType() !=
              CanUnqualT &&
          !Check(B.getType(), Check))
        return false;
    }
    // ... branches, loops, and calls to other functions
  };
  EffectivelyConstexprDestructor =
      Check(QualType(Record->getTypeForDecl(), 0), Check);
  // ...
}
```

(b) An example of a prioritized constraint. It guards many nodes down the CFG (*e.g.*, branches, loops, and calls to other functions), making it a rewarding target to solve.

Fig. 9: Constraint Examples. Examples of constraints before and after hybrid centrality prioritization.

```
// ...
if (const auto *intrinsic{
                    call.proc().GetSpecificIntrinsic()}) {
    if (intrinsic->name == "shape" ||
        // ...
    } else if (intrinsic->name == "pack") {
        < The target line to cover >
        // ...
// ...
```

(a) An excerpt of the enclosing function

```
subroutine spread_example()
  integer :: a(2) = (/1, 2/)
  integer :: b(2, 4)
  b = spread(a, 1, 2)
end subroutine
```

(b) A learning example that triggers line 10 in Fig. 11a

```
  ### Analysis of The Context
  The context include checks for various intrinsic functions.
  ...
  ### Strategy
  To cover the target, we can create a test case that uses
  the `pack` intrinsic function. Here's a modified test case that
  utilizes the `pack` intrinsic function:
  ```
  subroutine pack_example()
  integer :: a(2) = (/1, 2/)
  integer :: b(2, 4)
  b = pack(a, 1, 2)
  end subroutine
  ```
```

(c) LLM's response

```
//...
ArgumentAnalyzer analyzer{*this};
const auto &variable{std::get<parser::Variable>(x.t)};
analyzer.Analyze(variable);
analyzer.Analyze(std::get<parser::Expr>(x.t));
std::optional<Assignment> assignment;
if (!analyzer.fatalErrors()) { // <--- Your test case diverges here
    auto restorer{
        GetContextualMessages().SetLocation(
            variable.GetSource())
    };
// ...
```

(d) An excerpt of the diverging function

```
It seems that the test case contains an argument error.
### Key Points to Consider
To reach the target function and invoke the intrinsic logic that
is currently not covered, it requires specific argument structures.
### Strategy
To cover the target, we can create a test case that:
- Uses the `pack` intrinsic function, which requires at least three
arguments: the array, a mask, and an optional dimension argument.
- Ensures that the arguments passed to `pack` are valid and meet
the function's requirements.
### Revised Test Case
Here's a modified test case:
```
subroutine pack_example()
  integer :: a(4) = (/1, 2, 3, 4/)
  logical :: mask(4) = (/ .true., .false., .true., .false. /)
  integer :: b(2)
  b = pack(a, mask) ! Pack elements of a where mask is true
end subroutine
```
```

(e) LLM's response

Fig. 11: Case study A solving process details (1/2) HLPFUZZ starts by presenting the initial context, including the enclosing function shown in Fig. 11a and a learning example shown in Fig. 11b that triggers the constraint evaluation point at line 10. The LLM receives the context, analyzes it, and deems it necessary to include a `pack` operation in the test case to solve the constraint.

Fig. 12: Case study A solving process details (2/2) As shown in Fig. 11c, the LLM replaces the `spread` operation with `pack` and gives HLPFUZZ the new test case. However, the new test case does not pass early argument type checks, meaning that it does not satisfy the pre-constraint. Therefore, divergence analysis kicks in, which locates the diverging function shown in Fig. 11d. Note that HLPFUZZ also highlights the line where the execution diverges (*i.e.*, line 7 in Fig. 11d). Upon receiving the feedback from divergence analysis, the LLM realizes that the test case should use `pack` with the correct argument types. Therefore, it proceeds to revise the previous test case and gives HLPFUZZ a successful solution, as shown in lines 14 to 19 in Fig. 11e. This solution triggers the target line shown in Fig. 11a, and thereby gets synchronized to the hybrid queue. After a few syntax-aware mutations, HLPFUZZ quickly arrives at the test case shown in Fig. 5a, which triggers the crash in `Flang`. In our experiments, pure syntax-aware mutation cannot easily solve this constraint and consequently cannot discover this bug.

Table 7: Real-world bugs found by HLPFUZZ. HLPFUZZ has found 52 bugs in the latest versions of the language processors during evaluation. Status: **R**eported, **C**onfirmed, **F**ixed. * means HLPFUZZ found the bug in the latest target, but it was already known by the vendor when we reported it. **AF**: Assertion Failure; **SV**: SEGV; **SOF**: Stack Overflow; **ND**: Null-pointer Dereference; **HBOF**: Heap Buffer Overflow; **HUAF**: Heap Use After Free.

Target	ID	Status	Type	Target	ID	Status	Type
Clang++	1	C	AF	njs	27	C	HBOF
	2	F*	AF		28	R	HUAF
	3	C*	AF		29	C*	SV
	4	C*	AF		30	R	SV
	5	C	AF	Flang	31	F	AF
	6	C	SV		32	F	AF
	7	C	SV		33	F	AF
	8	C*	AF		34	F	AF
	9	C	SOF		35	F	HUAF
	10	C	SV		36	F	SOF
Clang	11	C	AF		37	F	AF
	12	C	AF		38	F	AF
	13	C	AF		39	F	AF
	14	C	SV	PHP	40	F	AF
	15	C	ND	lua	41	C	SV
	16	C	AF		42	R	SV
	17	C*	AF		43	R	SV
Hermes	18	F	SV		44	C	HBOF
	19	C*	ND	solc	45	F	SV
ChakraCore	20	R*	AF		46	F	AF
	21	R*	AF		47	R	AF
	22	R*	AF		48	R	AF
	23	R	AF		49	R	AF
	24	R	AF		50	R	AF
njs	25	C	ND		51	R	AF
	26	C*	ND		52	R	AF