Hybrid Fuzzing in Input Generation Logic

Siwei Wei
Key Laboratory of System Software (Chinese Academy of Sciences)
Beijing, China
Institute of Software, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
weisw@ios.ac.cn

Shihao Zhu
Key Laboratory of System Software (Chinese Academy of Sciences)
Beijing, China
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhush@ios.ac.cn

Ruijie Meng
National University of Singapore
Singapore, Singapore
ruijie$_m$eng@u.nus.edu

Yan Cai*
Key Laboratory of System Software (Chinese Academy of Sciences)
Beijing, China
Institute of Software, Chinese Academy of Sciences
Beijing, China
School of Computer Science and Technology, University of Science and Technology of China
Anhui, China
yancai@ios.ac.cn

Abstract

The core challenge of fuzzing is to consistently generate test inputs that are both syntactically valid and semantically diverse. However, mutation-based coverage-guided fuzzing often struggles to achieve this, especially for highly structured formats. Two lines of work have emerged to address this challenge: (1) hybrid fuzzing uses constraint solving to drive execution toward hard-to-cover program states to increase semantic diversity, and (2) generator-based fuzzing relies on predefined or synthesized input-generation programs to ensure syntactic validity. Both approaches, however, have significant limitations. (1) In hybrid fuzzing, solved inputs are fed back into the fuzzing queue for byte-level mutation, which often breaks syntactic structure (e.g., checksums and cross-field dependencies) and causes early rejection. (2) In generator-based fuzzing, semantic diversity is constrained by static generators based on domain knowledge; although recent work introduces generator mutations, they remain blind and therefore have limited adaptability to the target program's logic.

In this paper, we present hybrid generator space fuzzing, a method that combines the strengths of hybrid fuzzing and generator-based fuzzing to improve the syntactic validity and semantic diversity of test inputs. The key idea is to treat input generators, rather than test inputs, as the unit of targeted solving, and to use the LLM as a constraint solver to derive new generators. This design offers two benefits. First, compared with hybrid fuzzing, it preserves syntactic validity because inputs are generated directly from evolving generators rather than through byte-level mutation of solved seeds. Second, compared with generator-based fuzzing, it overcomes the limitations of static generators and blind generator mutations, enabling the fuzzer to adapt to target program logic and explore a broader semantic space. Building on this concept, we implement GenEvo, an LLM-driven hybrid generator space fuzzer. Our evaluation on a large set of real-world programs shows that GenEvo, using a locally deployed medium-sized LLM, significantly outperforms competitive baselines (including general-purpose, hybrid, structure-aware, and LLM-assisted fuzzers) in both branch coverage (with an average improvement of 98.3%) and vulnerability discovery. Furthermore, GenEvo has discovered 25 previously unknown vulnerabilities in these extensively-tested programs.

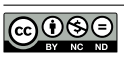
CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Hybrid Fuzzing, Generator-Based Fuzzing, Large Language Models

*Corresponding author.



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834412

ACM Reference Format:
Siwei Wei, Shihao Zhu, Ruijie Meng, and Yan Cai. 2026. Hybrid Fuzzing in Input Generation Logic. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834412

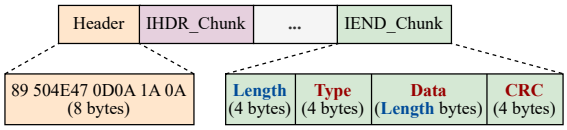


Figure 1: The syntactic structure of PNG format.

```python
import random ...

def create_chunk(type, data):
  length = len(data)
  crc_data = type + data
  crc = zlib.crc32(crc_data)\
     & 0xFFFFFFFF
  return struct.pack(
    '>I4s{}sI'.format(length),
    length, type, data, crc)

def generate_IHDR():...

def generate_PLTE():...

def generate_bKGD(context):...

def generate_pHYs(context):...

def generate_tIME():...

def generate_IDAT(context):...
```

```python
def generate_buf():
  buf = bytearray()
  buf.extend(b'\x89PNG\r\n\x1a\n')      Header
  ihdr_type, ihdr_data, context = \
    generate_IHDR()                     IHDR_Chunk
  ihdr_chunk = \
    create_chunk(ihdr_type, ihdr_data)
  buf.extend(ihdr_chunk)
  plte_chunk = None
  if context['color_type'] == 3: ...
  pre_chunks = [generate_bKGD, generate_pHYs]
  for gen in pre_chunks: ...

  idat_chunk = generate_IDAT(context)
  buf.extend(idat_chunk)
  post_chunks = [generate_tEXt, generate_tIME]
  for gen in post_chunks: ...

  iend_chunk = create_chunk(b'IEND', b'')
  buf.extend(iend_chunk)
  return bytes(buf)
```

Figure 2: A generator for PNG files produced by LLM.

1 Introduction

Coverage-guided fuzzing (CGF) [29, 34, 63] is one of the most widely used techniques for automated vulnerability detection. It is conceptually simple yet highly effective. CGF performs an evolutionary search over input space using predefined low-level mutation operators (e.g., bit-flip and arithmetic operations). This search is guided by coverage feedback, which serves as a fitness metric to select interesting inputs for further mutation. Through this iterative process, CGF progressively explores previously unseen program behaviors. Over the past decade, CGF has achieved great success in software security, leading to the discovery of thousands of vulnerabilities across a wide range of real-world software systems [12, 34].

Despite its success, CGF faces a long-standing challenge: it struggles to generate inputs that are both *syntactically valid* and *semantically diverse*. Syntactically valid inputs are essential to bypass initial parsing checks and reach deep program logic. However, conventional CGF often produces a large number of syntactically invalid inputs [13, 41, 45, 61], as low-level mutation operators applied directly to test inputs easily break syntactic structure. In addition to syntactic validity, semantic diversity is crucial for exploring complex and corner-case program behaviors. However, CGF also falls short in this regard. It relies on blind mutations with predefined operators, which struggle to produce meaningful semantic transformations. Such transformations require coordinated changes across multiple input fields [9, 21, 31, 47, 48], which are difficult to achieve with conventional CGF. As a result, CGF tends to uncover shallow vulnerabilities and struggles to explore deeper program behaviors.

Extensive research has been devoted to addressing this challenge, yet the fundamental problem remains. Hybrid fuzzing [40, 44, 62] integrates constraint solving with fuzzing to generate inputs that satisfy specific program conditions and reach hard-to-cover code regions, thereby improving semantic diversity. By leveraging precise information about the target program's logic, it mitigates the blindness of CGF. However, hybrid fuzzing operates in a seed-enrichment

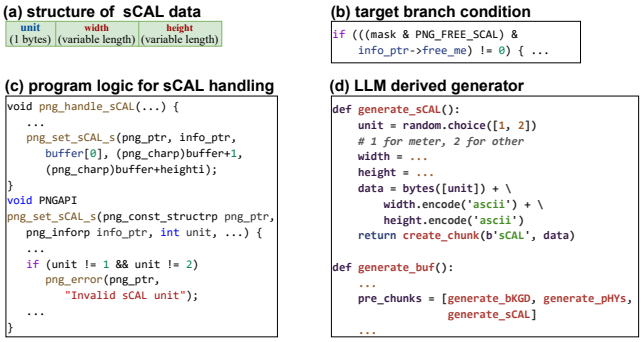


Figure 3: Generating specification-missing semantic features.

paradigm: solved inputs are added back to the fuzzing queue and further mutated at the byte level. These mutations frequently break syntactic structure [13, 45, 61]. For example, in the PNG format (Figure 1), inputs must satisfy cross-field dependencies and checksums. Even if a solved input is initially well-formed, subsequent byte-level mutations will almost certainly break these constraints, leading to early rejection during parsing. As a result, hybrid fuzzing struggles to consistently generate syntactically valid inputs.

Another line of work, generator-based fuzzing [22, 25, 28, 37], employs input generation programs (e.g., Figure 2) to produce syntactically valid inputs by construction. While this approach effectively ensures syntactic validity, its ability to improve semantic diversity is limited. In particular, traditional generators are often static and derived from specifications or domain knowledge, making them difficult to adapt based on program feedback. Recent works [7, 10, 60] attempt to address this limitation by applying rule-based or LLM-based mutations to generators. However, these mutations remain blind to the target program's internal logic, relying on heuristics or LLM's prior knowledge. Consequently, they have limited ability to introduce program-specific semantic variations. For instance, consider the sCAL chunk in PNG (Figure 3 (a)), whose structure is not documented in the official specification[1]. Generators derived from the specification do not include this chunk, and blind mutations, whether rule-based or LLM-based, are unlikely to synthesize it correctly. This derivation instead requires reasoning about the target program's implementation.

To address the limitations of existing approaches, we present hybrid generator space fuzzing, which combines the key ideas of hybrid fuzzing and generator-based fuzzing. The central design is to shift the unit of targeted solving from test inputs to input generators. In this setting, input generators play a role analogous to seeds in conventional hybrid fuzzing, while an LLM is used as a constraint solver to derive new generators that satisfy specific program conditions. This design provides two advantages. First, unlike hybrid fuzzing, which relies on byte-level mutation of solved inputs, our approach evolves input-generation logic. As inputs are produced by generators rather than mutated from existing seeds, syntactic validity is likely to be preserved. For instance, when fuzzing libpng, inputs produced by generators (e.g., Figure 2 and Figure 3 (d)) naturally maintain structural constraints such as chunk ordering and checksums. Second, compared with generator-based

[1] https://www.w3.org/TR/png-3/

fuzzing, our approach improves semantic diversity by enabling program-aware adaptation of generators. Instead of relying on static generators or blind mutations, it performs constraint-guided generator solving [53] based on the target program's logic using the LLM. For example, to satisfy a target branch condition (Figure 3 (b)), the LLM can analyze the relevant implementation code (Figure 3 (c)) and synthesize a generator (Figure 3 (d)) that introduces the required semantic feature, the undocumented sCAL chunk (Figure 3 (a)). This capability allows the fuzzer to explore program-specific behaviors that are otherwise difficult to reach, thereby increasing semantic diversity.

Based on this concept, we develop GenEvo, a hybrid generator space fuzzer that evolves a corpus of input generators with LLM-based generator solving. At the beginning of fuzzing, GenEvo prompts the LLM to synthesize initial generators for the target input format, which serve as the starting population. During fuzzing, GenEvo performs targeted generator solving: given an uncovered branch, it uses the LLM as a constraint-guided generator synthesizer to derive new generators that are more likely to produce inputs exercising the desired program behavior. To improve the effectiveness and efficiency of this solving process, GenEvo augments the prompt with two forms of additional guidance. First, it injects actionable specification knowledge by providing specification-compliant reference implementations, thereby reducing LLM errors caused by incomplete or inaccurate format knowledge. Second, it applies context retrieval to include relevant code fragments from the target program, obtained via RAG, so that the LLM can reason over implementation-specific semantics. In addition, to keep the search efficient and reserve targeted solving for hard-to-reach branches, GenEvo also applies lightweight random mutations to input generators as a complementary exploration mechanism. The resulting generators are evaluated using coverage feedback, and only the fittest ones are retained for subsequent rounds.

To evaluate the performance of GenEvo, we selected 49 real-world programs that are widely used as fuzzing benchmarks. We locally deployed Qwen3-32B [51], a medium-sized model, as the backend LLM. For comparison, we selected 13 representative fuzzers as baselines, including general-purpose, hybrid, structure-aware, generator-based, and LLM-assisted approaches. The evaluation results show that GenEvo significantly outperforms the baselines, achieving an average increase of 98.3% in branch coverage. In total, GenEvo detected 25 previously unknown bugs in these extensively-tested programs, with one CVE assigned at the time of writing. These results demonstrate the effectiveness of GenEvo in fuzzing real-world programs.

In summary, our contribution is three-fold:

- We introduce hybrid generator space fuzzing, a novel paradigm that combines the strengths of hybrid fuzzing and generator-based fuzzing, to overcome the long-standing challenge of generating both syntactically valid and semantically diverse inputs.
- We implement GenEvo, a hybrid generator space fuzzer. The code of GenEvo is available [2].
- We evaluate GenEvo on 49 real-world programs. The results demonstrate that GenEvo significantly outperforms state-of-the-art CGF baselines in both code coverage and bug discovery. In total, GenEvo discovers 25 previously unknown vulnerabilities in these subjects.

[2] https://github.com/genevo-fuzz/genevo, https://zenodo.org/records/19244111

2 Background and Related Work

2.1 Hybrid Fuzzing

Hybrid fuzzing [40, 44, 62] aims to improve the semantic diversity of test inputs by combining CGF with symbolic reasoning techniques such as concolic execution. The key idea is to leverage constraint solving to systematically explore hard-to-reach program paths that are unlikely to be discovered through random mutation alone. In practice, a concolic engine analyzes program executions to extract path constraints and solves them to generate concrete inputs satisfying specific branch conditions. Hybrid fuzzers adopt a seed enrichment paradigm, where constraint-solved inputs are added to the seed corpus and subsequently processed by the underlying CGF. This design combines precise, goal-directed exploration with the efficiency and scalability of mutation-based fuzzing. Prior work has shown that hybrid fuzzing is effective in bypassing complex checks such as magic values, checksums, and structured input validations, thereby improving code coverage and enabling exploration of deeper program behaviors [23, 42, 49, 57].

Despite these advantages, hybrid fuzzing does not address the challenge of maintaining syntactic validity during input generation. Under the seed enrichment paradigm, solved inputs are added to the seed corpus and subject to the same byte-level mutations as ordinary seeds. For programs that process highly structured inputs, such mutations frequently violate format constraints, including cross-field dependencies and integrity checks, causing inputs to be rejected early during parsing. As a result, many enriched seeds fail to produce follow-up inputs that remain valid beyond shallow execution stages. This reduces exploration efficiency, as substantial computational effort is wasted on inputs that never reach deeper program logic.

2.2 Generator-Based Fuzzing

Generator-based fuzzing [22, 25, 28, 37] aims to ensure syntactic validity by constructing inputs through input-generation programs rather than mutating raw byte sequences. These approaches [11, 55] typically rely on manually designed generators derived from domain knowledge to produce well-formed inputs. This paradigm has been shown to be effective for fuzzing programs that process highly structured inputs. Recent works extend this paradigm by introducing generator mutation. For instance, Fuzztruction [7] injects faults into existing generation programs to perturb their behavior, G2Fuzz [60] employs feature-based mutations guided by documents, and ELFuzz [10] evolves generators by applying structured mutation operators. Overall, these approaches are effective at preserving syntactic correctness and improving fuzzing performance on structured input formats.

Despite these advances, generator-based fuzzing remains limited in achieving high semantic diversity. Static generators derived from specifications are fixed, may be incomplete, and require substantial human effort to construct; they cannot adapt to program-specific

behaviors observed during fuzzing. While recent approaches introduce generator mutation to improve flexibility, these mutations are typically blind to the internal logic of the target program. Rule-based mutations operate according to predefined heuristics without awareness of program semantics, and LLM-based mutations largely rely on prior knowledge of input formats rather than reasoning over the target program's implementation. Consequently, such methods struggle to introduce semantic variations that require coordinated, program-specific modifications. For example, generating an input that triggers an undocumented behavior in libpng, such as the insertion of an sCAL chunk (see Figure 3), requires reasoning about how the program processes inputs internally. Blind mutations, whether rule-based or LLM-driven, are unlikely to synthesize such behaviors correctly. Without program-aware guidance, generator-based fuzzing remains limited in its ability to explore deeper and more complex semantic behaviors.

2.3 Structure-Aware Fuzzing

Structure-aware fuzzing aims to improve the effectiveness of mutation-based fuzzing by incorporating knowledge of input structure. This is typically achieved either through domain-specific specifications (e.g., input grammars or format-aware generators), or through heuristics that capture structural dependencies. This line of work partially overlaps with generator-based fuzzing but focuses on enhancing mutation. A class of fuzzers, including AFLSmart [41] and Gramatron [48], relies on manually crafted input specifications to guide input generation and mutation. While effective in preserving structural validity, their performance depends heavily on the completeness and accuracy of the provided specifications, which are often labor-intensive to construct and prone to errors. To reduce manual effort, Weizz [15] infers byte-level dependency from existing inputs using heuristics and incorporates them into mutation strategies. However, such inferred models are approximate and may fail to capture subtle structural constraints, limiting their generality. Overall, existing structure-aware fuzzers are limited in increasing semantic diversity, as they generally lack mechanisms to introduce program-specific, high-level semantic variations.

2.4 LLM-Based Constraint Solving

Recently, several works have explored using LLMs as constraint solvers in fuzzing and concolic testing. HyLLfuzz [35] constructs dynamic slices for uncovered branches and prompts an LLM to patch concrete inputs without translating path conditions into SMT formulas. HLPFuzz [56] targets language processors by prioritizing promising semantic constraints and iteratively expanding program context until a satisfying test case is solved. Cottontail [52] integrates an LLM-driven solver into concolic execution to generate syntax-valid inputs satisfying selected path constraints, while ConcoLLMic [32] employs LLM agents to summarize execution traces and solve constraints across programming languages and constraint theories.

Despite their effectiveness, these approaches formulate the solving target in input space: the result is ultimately one concrete test inputs. Consequently, the semantic knowledge inferred through an expensive LLM query is tied to those inputs rather than retained as reusable generation logic; in hybrid settings, subsequent input-level mutations may again violate structural dependencies. In contrast, GenEvo performs constraint solving in generator space. By synthesizing generation logic instead of a single satisfying input, one LLM query can produce many structurally valid inputs, and the derived generator can be evolved throughout fuzzing.

3 Method

In this paper, we present hybrid generator space fuzzing, a method that combines the strengths of hybrid fuzzing and generator-based fuzzing to improve both syntactic validity and semantic diversity of generated inputs. Similar to generator-based fuzzing, GenEvo uses generators to produce syntactically valid inputs. Similar to hybrid fuzzing, GenEvo introduces goal-directed solving at the level of generation logic, where an LLM is used to derive new generators that satisfy specific program conditions to increase semantic diversity. The overall workflow of GenEvo is shown in Figure 4.

The process begins with an initialization phase, where GenEvo prompts the LLM to synthesize a set of seed generators for the target input format, forming the initial population (①). During fuzzing, the core mechanism is LLM-based generator solving (②). Given a previously uncovered branch, the LLM synthesizes a generator that is more likely to produce inputs exercising the branch. This process is analogous to the constraint-guided exploration in hybrid fuzzing and serves as the primary driver for increasing semantic diversity. However, applying generator solving to all branches would be computationally expensive and unnecessary, as many program paths can be explored without precise reasoning. To address this, GenEvo incorporates a lightweight LLM-based random mutation (③). Instead of solving every branch, GenEvo first relies on inexpensive random modifications to diversify generator behaviors and cover easy-to-reach paths, reserving generator solving for branches that remain uncovered. To improve efficiency, GenEvo manages LLM interactions through a request pool that enables asynchronous and controlled query execution (④). GenEvo maintains a corpus of candidate generators throughout the fuzzing process. In each round, newly generated mutants are added to the corpus, and each generator is executed multiple times to produce concrete inputs, which are then fed to the instrumented program. Coverage feedback is attributed to the generator that produced each input, and a fitness score is assigned to reflect its effectiveness in discovering new behaviors (④). Based on this score, only the top-N generators are retained for the next round, while the rest are discarded. This iterative loop of solving-driven evolution, assisted by lightweight mutation, progressively drives the system toward increasingly effective generators that produce both syntactically valid and semantically diverse inputs. We describe ① in Section 3.3, ② in Section 3.1, ③ in Section 3.2, ④ in Section 3.4, and ⑤ in Section 3.5.

3.1 LLM-Based Generator Solving

The goal of generator solving is to derive input generators that produce test inputs capable of exercising previously uncovered branches, thereby improving semantic diversity. To support this, the target program is instrumented using Clang [27] to collect coverage information. Conditional branches identified at the intermediate representation (IR) level are mapped back to corresponding source-level conditional statements using debug location metadata [5]. A

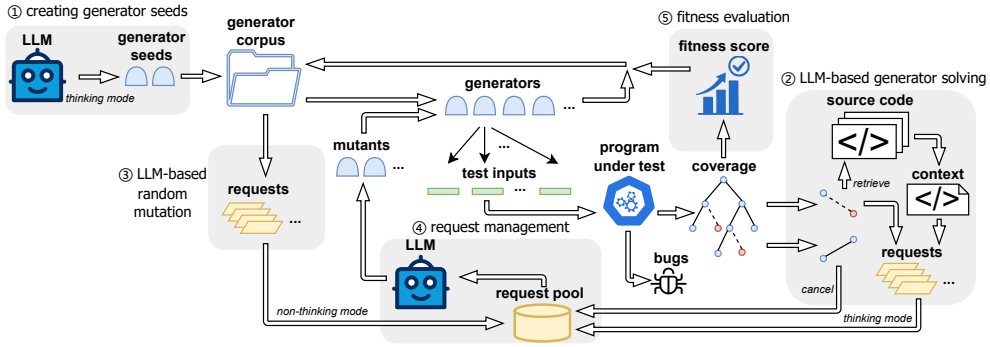


Figure 4: The workflow of GenEvo

targeted solving task is triggered when a conditional statement has been reached during execution, but only one of its branches has been covered. The objective is then to generate inputs that exercise the opposite, currently uncovered branch. Unlike conventional hybrid fuzzing, which solves for concrete inputs, GenEvo performs solving at the level of input-generation logic. Given a target branch, the LLM is prompted to synthesize a generator whose outputs are likely to satisfy the branch condition. To improve this process, we first conduct an empirical analysis of LLM-based generator solving to understand its reasoning behavior and common failure modes. Based on this, we design strategies to improve the quality of synthesized generators.

3.1.1 Empirical Analysis. To better understand the capabilities of LLM-based generator solving, and to guide prompt design, we conducted an empirical analysis on 65 roadblock branches encountered while fuzzing libpng. For each branch, we prompted the LLM to synthesize a generator for the uncovered branch, and manually examined its reasoning process and synthesis outcome. Our analysis shows that, when solving branches in real-world repository-scale programs, the LLM typically follows a two-stage process rather than performing precise low-level symbolic reasoning.

Stage 1: Constraint modeling. The LLM abstracts the target branch condition into a high-level input requirement. Rather than reasoning over symbolic variables, it attempts to infer what semantic feature must be present in the input for the branch to be taken. For structured formats, this often corresponds to identifying required input components or structural relationships.

Stage 2: Generator implementation. The LLM then translates the high-level requirement into concrete generator logic. This involves implementing generators that can produce inputs satisfying the inferred condition while being format-compliant.

Figure 3 illustrates this on libpng. Given a target branch related to the handling of sCAL chunk (Figure 3 (b)), the LLM first infers, from the relevant implementation code (Figure 3 (c)), that reaching the branch requires the generated PNG to contain a valid sCAL chunk with specific structure requirements. This corresponds to the constraint-modeling stage. It then synthesizes a corresponding generator function and integrates it into the PNG generation pipeline (Figure 3 (d)), which corresponds to the implementation stage. This suggests that LLM-based generator solving is better understood as a form of high-level semantic reasoning followed by code realization, rather than exact symbolic constraint solving. We found that this process exposes two recurring failure modes.

(1) Incorrect constraint modeling. In some cases, the LLM fails to correctly interpret the target branch in the broader code context. As a result, it derives an incorrect or incomplete high-level constraint. This issue is particularly common when the meaning of a branch depends on code outside the local context initially provided to the model.

(2) Incorrect generator implementation. In other cases, the LLM correctly infers the required high-level condition, but fails to implement it in a specification-compliant manner. That is, the synthesized generator does not correctly realize the intended input structure, often due to mistakes in format-specific details such as field encoding and ordering constraints. As a result, the generated inputs can remain invalid and fail to trigger the intended behavior despite targeting the correct semantic feature.

3.1.2 Prompt Strategy. The empirical analysis above suggests that the effectiveness of LLM-based generator solving is limited by two issues: *insufficient format-specific implementation knowledge*, which leads to incorrect generator implementation, and *insufficient program context*, which contributes to incorrect constraint modeling. To address these limitations, we design a prompting strategy that augments solving requests with two forms of guidance: actionable domain knowledge and context retrieval.

Our design is intentionally lightweight. Unlike recent LLM-based program analysis and solving frameworks that rely on multi-step agentic interaction with iterative tool use [32, 59], our method is designed for fuzzing, where a large number of branch-solving tasks can be issued. In this setting, solving overhead must remain low to avoid becoming the bottleneck of the fuzzing loop. Accordingly, GenEvo improves solving quality by enriching a single prompt with carefully selected supporting information.

Actionable domain knowledge. To reduce implementation errors caused by incomplete or inaccurate format knowledge, GenEvo provides the LLM with specification-compliant reference generator code as in-context examples. The key idea is that the LLM is more likely to synthesize format-compliant generators when grounded in executable reference implementations, rather than relying solely on textual format descriptions. This design is particularly important for structured binary formats, where many validity constraints are procedural rather than easily expressed in natural language,

Algorithm 1: Creating Reference Generator

Input: Specification F, Validator Program P_v, *LLM*
Output: Reference Generator g

Generator $g \leftarrow$ `InitialGenerator`(*LLM*, F)
repeat
  ValidationErrors $E \leftarrow \emptyset$
  for sample $\in$ {1, ..., MaxValidateSamples} **do**
    Input $i \leftarrow$ `SampleProgramInput`(g)
    Result $r \leftarrow$ `ExecuteValidator`(P_v, i)
    $E \leftarrow E \cup$ {`ErrorMessage`(r)}
  if $E = \emptyset$ **then**
    break
  $g \leftarrow$ `RepairGenerator`(*LLM*, F, g, E)
until maximum refinement rounds reached
return g

Algorithm 2: Context Retrieval

Input: SourceCode S, TargetBranch b_t, TokenBudget T
Output: RetrievedContext C_r

SyntaxUnits $U \leftarrow$ `ParseAndSegmentWithClang`(S)
TargetStmt $s_t \leftarrow$ `GetSourceStmt`(b_t)
LocalCtx $c_t \leftarrow$ `GetSurroundingCode`(s_t)
EntrySig $e \leftarrow$ `GetEntrySignature`(S)
Query $q \leftarrow$ `BuildQuery`(s_t, c_t, e)
RankedUnits $R \leftarrow$ `BM25Retrieve`(q, U)
RetrievedContext $C_r \leftarrow \emptyset$
UsedTokens $n \leftarrow 0$
for $u \in R$ **do**
  Tokens $t \leftarrow$ `CountTokens`(u)
  if $n + t > T$ **then**
    break
  RetrievedContext $C_r \leftarrow C_r \cup \{u\}$
  $n \leftarrow n + t$
return C_r

such as field encoding rules, ordering constraints, and checksum construction. Reference generator code exposes these constraints in an operational form that the LLM can directly imitate and adapt during synthesis.

To construct such references, GenEvo first prompts the LLM to generate an initial input generator from the format specification. The generated program is then validated by executing the corresponding parser. If the generated inputs trigger format-related parsing failures, the observed error messages are fed back to the LLM for repair. This refinement loop is repeated until the resulting generator consistently produces syntactically valid inputs. The final refined generator is then used as a reference for subsequent solving tasks. Algorithm 1 summarizes this process.

Context retrieval. To reduce failures in constraint modeling caused by incomplete program context, GenEvo augments each solving request with retrieved code fragments from the target program. The motivation is that the local branch condition alone is often insufficient to reveal the semantic requirement needed to exercise the uncovered branch. In many cases, the meaning of a branch depends on helper functions or format-handling logic located elsewhere in the codebase. For example, in the libpng case shown in Figure 3, the target conditional statement alone does not explicitly reveal how to construct the sCAL chunk. Retrieving the relevant implementation code allows the LLM to infer the required structure of that chunk.

To provide such information efficiently, GenEvo adopts retrieval-augmented generation (RAG) [14, 17], a standard technique for enabling LLMs to reason over large codebases [24, 38]. Specifically, we use Clang [27] to parse the target program and segment it into fine-grained syntactic units. For each target branch, GenEvo constructs a retrieval query using the source-level conditional statement and its surrounding code context. It then applies BM25 [43] to retrieve the most relevant code units, under a fixed token budget. The retrieved code is appended to the solving prompt as additional context. Algorithm 2 summarizes this process.

Final prompt. Each generator-solving request in GenEvo is formulated as a single prompt with four components: (a) a task instruction specifying the branch-solving objective; (b) the target conditional branch; (c) retrieved implementation context relevant to the branch; and (d) specification-compliant reference generator code. Figure 5 shows the prompt structure (the full prompt is provided in the supplementary material).

```
You are an expert in software analysis and fuzz testing. Your task
is to write a Python function named `generate_buf` that produces
random byte sequences as test inputs for a program
...                                                  (a) Task Instruction

The target conditional statement is the following:
```cpp
[CONDITIONAL_STATEMENT]
```                                                  (b) Target Branch
Your goal is to make the program take the [BRANCH_TO_TAKE] branch
of the target conditional statement.

Here is an additional related code context:
```cpp                                               (c) Retrieved Context
[CODE_CONTEXT]
```

The reference specification-compliant generator:
```python                                            (d) Reference Generator
[GENERATOR_CODE]
```
```

Figure 5: The prompt for generator solving.

3.2 LLM-Based Random Mutation

Directly applying LLM-based generator solving to every uncovered branch is inefficient, as many branches can be reached without precise, goal-directed reasoning. Conventional CGF has shown that numerous program behaviors can be discovered through inexpensive, unguided exploration. To address this, GenEvo introduces LLM-based random mutation as a lightweight auxiliary mechanism. Its goal is to reduce the number of branches requiring targeted generator solving. If a branch can already be covered through random generator perturbation, there is no need to allocate a solving request. Given an existing generator, GenEvo prompts the LLM to produce random modifications to its code so as to alter its output behavior while preserving the overall generation structure. Because the mutation target is the generator code rather than input bytes, much of the structural regularity of the format is retained.

```
You are an expert in software analysis and fuzz testing.
### Your task:
- Generate at least `n` distinct mutated versions of the provided
code snippet.
...
```
(a) Task Instruction

```
### Format requirements:
- Instead of the raw code blocks, provide all mutations as a
**sequence of diff patch blocks** using the following format:
...
```
(b) Format Requirement

```
Here is the generator code that needs to be mutated:
```python
[GENERATOR_CODE]
```
```
**(c) Generator Code**

**Figure 6: The prompt for random mutation.**

GenEvo adopts several optimizations to make random mutation efficient. First, instead of requesting a single mutant per LLM call, GenEvo asks the model to return multiple mutations in a diff-patch format [39]. Second, since random mutation does not require deep branch-specific reasoning, the model is used in its non-thinking mode for these requests. The structure of random mutation prompt is shown in Figure 6, it contains: (a) a task instruction requesting behavior-altering modifications; (b) the diff-patch format specification; and (c) the original generator code to be mutated. The full prompt is provided in the supplementary material.

## 3.3 Creating Generator Seeds

Like other evolutionary fuzzers, GenEvo requires an initial population to start the search. This initial population consists of input generators for the target format. To construct these seed generators, GenEvo uses the same generation and refinement procedure described in Algorithm 1. In our experiments, GenEvo is initialized with five seed generators.

## 3.4 LLM Request Management

The performance of LLM-based fuzzing is sensitive to inference latency and throughput. To prevent LLM latency from stalling the fuzzing loop and to ensure controlled resource usage, we design an asynchronous request pool to manage all interactions.

The request pool maintains two internal queues: (1) an ongoing queue for requests that have been dispatched to the LLM and are awaiting responses, and (2) a waiting queue for requests that have not yet been issued. A maximum capacity (set to 40 in our experiments) is imposed on the ongoing queue to bound the number of concurrent LLM queries and maintain a stable resource budget. When a new request is generated, it is immediately dispatched to the LLM only if the ongoing queue is not full and the waiting queue is empty. Otherwise, the request is placed in the waiting queue until execution capacity becomes available. During fuzzing, all newly generated requests (both random mutation and targeted solving) are submitted to the pool at the end of each round. At the beginning of subsequent rounds, the fuzzer retrieves completed responses and incorporates the resulting generator updates. The request pool also supports request cancellation. In particular, pending targeted solving requests are canceled as soon as their corresponding target branches have been covered, avoiding unnecessary computation and reducing overall solving overhead.

## 3.5 Fitness Evaluation

In line with conventional CGF, GenEvo adopts coverage feedback as its reward signal. Specifically, it uses a composite reward that incorporates two metrics: simple new branch coverage and a more fine-grained measure based on bucketing branch hit counts, as used by AFL [6]. The immediate reward for a generator $g$ in a given round $n$, denoted as $reward(g, n)$, is calculated as follows:

$$reward(g, n) = newBrCov(g, n) + \alpha \cdot newBrBucketCov(g, n) \quad (1)$$

Here, $newBrCov(g, n)$ is the number of inputs generated by $g$ in round $n$ that cover a new branch, and $newBrBucketCov(g, n)$ is the number of inputs generated by $g$ in round $n$ that cover a new branch-count bucket. The parameter $\alpha$ is used to balance the contribution of the two components, which is set to 0.2 in our experiments. This formula assigns a higher reward for discovering entirely new branches and a smaller reward for hitting new branch-count buckets for existing branches.

At the end of each round, an overall fitness score is computed for every generator to estimate its long-term potential in contributing to further exploration. This score is based on the intuition that a generator's future performance is positively correlated with the rewards it has gained in previous rounds, with more recent rewards being stronger indicators. Therefore, we use a discounted return [50], a concept commonly used in reinforcement learning, to calculate the fitness score for generator $g$ at the end of round $n$:

$$fitness(g, n) = \sum_{k=1}^{n} \gamma^{n-k} \cdot reward(g, k) \quad (2)$$

Here, $\gamma$ is a discount factor (set to 0.9 in our experiments), which progressively downweights older rewards in favor of more recent ones. This allows the scoring mechanism to dynamically emphasize a generator's current effectiveness while reducing the influence of outdated behavior that may no longer be relevant.

Following fitness computation, GenEvo retains only the top-$N$ generators (with $N$=100 in our experiments) for the next generation, discarding the rest. This selection strategy reduces the computational cost, concentrating the search effort on generators that demonstrate a higher capacity to uncover new program behaviors.

# 4 Evaluation

To evaluate the performance of GenEvo, we seek to answer the following research questions:

**RQ.1 (Code Coverage)** Can GenEvo improve code coverage in comparison to baseline approaches?
**RQ.2 (Bug Detection)** Is GenEvo effective in detecting bugs?
**RQ.3 (Ablation)** What is the impact of random mutation on the performance of GenEvo? How does the choice of the backend LLM affect GenEvo?

## 4.1 Evaluation Setup

*Subject Programs.* We constructed a comprehensive evaluation suite to evaluate GenEvo, which comprises 49 real-world subject programs sourced from 3 widely-used fuzzing benchmarks: FuzzBench [36], UniFuzz [30], and Magma [20]. Our suite includes all packages from these benchmarks, with the exception of four packages (i.e.,

`libxslt`, `openthread`, `systemd`, and `swftools`) because of compilation issues. These subject programs collectively cover 59 input file formats, ranging from highly-constrained (e.g., `sql`) and medium-constrained (e.g., `png`) to loosely-constrained (e.g., `elf`). The details of the subject programs are in the supplementary material.

*Baselines.* For comparison, we selected ten coverage-guided fuzzers spanning five categories, and configured AFL++ with the four most widely used options. This resulted in a total of 13 baseline fuzzers. The details are shown below:

***• General-Purpose Fuzzers:***

- AFL [58]: The foundational and highly influential fuzzer.
- MOPT [33]: A greybox fuzzer that employs particle swarm optimization to dynamically adjust the selection probabilities of mutation operators.
- AFL++ [16]: An extension of AFL that incorporates numerous cutting-edge fuzzing techniques and optimizations. We evaluate AFL++ with four of its power schedulers: explore (balances energy across paths; the default), fast [8] (assigns energy proportional to selection frequency), mmopt (applies targeted optimizations to newly discovered paths), and quad (calculates energy using a quadratic function).

***• Hybrid Fuzzer:***

- SymCC [42]: A concolic execution engine designed for hybrid fuzzing that collects path constraints through instrumentation. The QSYM backend [57] is used.

***• Structure-Aware Fuzzers:***

- AFLSmart [41]: A fuzzer that performs structure-aware mutations on complex data formats by leveraging high-level structural representations.
- Gramatron [48]: A fuzzer that utilizes grammar automatons to generate structured inputs. We employ the implementation integrated within AFL++ [1].
- Weizz [15]: A binary-only fuzzer that infers input structure by analyzing comparison instructions, enabling structural mutations on chunk-based binary formats.

Due to the reliance on predefined structural information, AFLSmart and Gramatron cannot support all input formats within our benchmark suite. Specifically, AFLSmart only supports `elf`, `jpeg`, `bloaty_composite`, `mp3`, `mp4`, `pcap`, `pdf`, `png`, `wav`, and `zip`, while Gramatron only supports `javascript`, `php`, and `ruby`. During the evaluation, we found that Weizz failed to execute on two binaries in our test suite: `libxml2` and `php`.

***• Generator-Based Fuzzer:***

- Fuzztruction [7]: A fuzzer that mutates a generator through rule-based fault injection to produce valid-like inputs. Its application is limited to programs with known generators (the original paper provides generators for 8 formats). Within our test suite, Fuzztruction supports `elf`, `png`, `pdf`, and `zip`.
- G2Fuzz [60]: A fuzzer designed to enrich the seed corpus. To address the limitations of LLMs in generating non-textual inputs, G2Fuzz first synthesizes input generators with specific features and then executes them to produce seeds.
- ELFuzz [10]: A fuzzer that synthesizes input generators using three LLM-based structured mutation operators. It performs a long-running synthesis phase (typically over 20 hours), whose outputs are fed as seeds to AFL++ for mutation-based fuzzing. The current implementation of ELFuzz supports four targets in our benchmark, namely `libxml2`, `jsoncpp`, `sqlite3`, and `re2`.

For each baseline fuzzer, we run three instances in the AFL's parallel mode [18], which is a commonly used practice in fuzzing research [54, 57]. If a fuzzer is already configured in the parallel mode, we retain its original setup without modification. The resulting configurations are as follows. For AFL, MOPT, AFL++, AFLSmart, Weizz, and ELFuzz, the setup involves one master instance and two slave instances. For SymCC, the setup involves two fuzzing instances (one master, one slave) and one SymCC concolic executor instance. For Gramatron, Fuzztruction, and G2Fuzz, the setup involves two AFL++ instances (one master, one slave) and one instance of the respective fuzzer.

*Configuration of* GenEvo. We use two configurations:

- GenEvo(standalone): This configuration performs pure hybrid generator space fuzzing, running a single instance.
- GenEvo: It integrates GenEvo with conventional fuzzers, forming the same number of fuzzer instances as that of baselines: one GenEvo instance, one AFL instance, and one SymCC concolic executor. All instances operate in the AFL's parallel mode.

*LLM Setting.* We use Qwen3-32B [51], a medium-sized, open-source model, as the backend LLM for LLM-assisted fuzzers (i.e., G2Fuzz, ELFuzz, and GenEvo). Qwen3-32B is a hybrid reasoning model that supports two modes: a fast but less capable non-thinking mode, and a slower but more capable thinking mode. Compared to large-scale, closed-source, and costly models such as GPT-4, which are more powerful and commonly used in prior LLM-assisted fuzzing research, Qwen3-32B is a lighter and more cost-effective alternative, although with weaker capabilities.

To optimize throughput, we deploy an AWQ-quantized version of this model [3] locally using vLLM [26] on an Ubuntu 24.04 machine equipped with one NVIDIA H100 PCIe GPU and one NVIDIA A100 PCIe GPU. For controlled resource usage, we limit the size of the ongoing request queue to 40 per fuzzing instance. This ensures that all LLM-assisted fuzzers operate with identical model capabilities and experience comparable response throughput. In this setup, the LLM generates each random mutation mutant in approximately 2.2 seconds, and each targeted solving mutant in about 86.1 seconds on average for GenEvo.

*Fuzzing Setting.* All target programs are compiled with AddressSanitizer [46] to detect memory corruption vulnerabilities. For fuzzers that require seed corpus (i.e., all baseline fuzzers and GenEvo), we use the high-quality seed corpora and fuzzing dictionaries [2] provided by FuzzBench [36], UniFuzz [30], and Magma [20] to ensure optimal performance. Each experiment is run for 24 hours on a Ubuntu 24.04 machine with Intel(R) Xeon(R) Platinum 8260 CPU (96 cores in total).

## 4.2 Branch Coverage (RQ.1)

Code coverage is widely used for evaluating the performance of fuzzers, and we used it as well. We report the total number of

**Table 1: Branch coverage in 24 hours across programs and fuzzers.**

| Program | AFL | MOPT | AFL++ (explore) | AFL++ (fast) | AFL++ (quad) | AFL++ (mmopt) | SymCC | AFLSmart | GT | Weizz | FT | ELFuzz | G2Fuzz | GenEvo(s) | *Imp(%)* | GenEvo | *Imp(%)* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bento4/mp42aac | 235 | 155 | 165 | 163 | 163 | 155 | 340 | 1362 | N/A | 248 | N/A | N/A | 928 | 2330 | 931.6 | 2407 | 965.7 |
| bento4/mp42hevc | 444 | 149 | 157 | 161 | 170 | 167 | 332 | 1106 | N/A | 268 | N/A | N/A | 1372 | 2120 | 786.0 | 2333 | 875.0 |
| bento4/mp42avc | 402 | 174 | 161 | 149 | 160 | 163 | 268 | 1143 | N/A | 262 | N/A | N/A | 174 | 2277 | 986.4 | 2468 | 1077.5 |
| binutils/size | 2782 | 3174 | 2782 | 2928 | 2965 | 2780 | 3164 | 2961 | N/A | 2997 | 2824 | N/A | 3478 | 3708 | 24.8 | 4646 | 56.3 |
| binutils/readelf | 9870 | 9785 | 7407 | 9375 | 8715 | 7890 | 10064 | 10975 | N/A | 4889 | 10432 | N/A | 6818 | 11679 | 40.6 | 13467 | 62.1 |
| binutils/nm-new | 2257 | 2520 | 2367 | 2500 | 2714 | 2189 | 2571 | 2260 | N/A | 2525 | 3188 | N/A | 2054 | 2921 | 19.9 | 3670 | 50.7 |
| binutils/objdump | 5694 | 5468 | 5045 | 5311 | 5395 | 5557 | 6180 | 5748 | N/A | 3474 | 5851 | N/A | 4384 | 6343 | 23.0 | 8624 | 67.2 |
| binutils/strip-new | 4869 | 5245 | 4176 | 4451 | 4276 | 4432 | 6331 | 4792 | N/A | 3538 | 5111 | N/A | 3981 | 6274 | 37.8 | 8410 | 84.7 |
| bloaty/fuzz_target | 3069 | 2923 | 2944 | 2810 | 2785 | 3007 | 3164 | 3134 | N/A | 2473 | N/A | N/A | 2876 | 3424 | 17.9 | 4043 | 39.2 |
| cflow/cflow | 1645 | 1666 | 1652 | 1672 | 1671 | 1671 | 1643 | N/A | N/A | 1648 | N/A | N/A | 1678 | 1709 | 2.9 | 1709 | 2.9 |
| curl/fuzzer_http | 1034 | 1032 | 1035 | 1034 | 1036 | 1035 | 1034 | N/A | N/A | 1015 | N/A | N/A | 1036 | 1005 | -2.6 | 1038 | 0.6 |
| exiv2/exiv2 | 2490 | 2845 | 3179 | 2607 | 2656 | 3098 | 1956 | 1821 | N/A | 3099 | N/A | N/A | 3222 | 5060 | 94.5 | 5688 | 118.6 |
| ffmpeg/ffmpeg | 11355 | 13298 | 11904 | 11935 | 11894 | 11732 | 11410 | N/A | N/A | 8967 | N/A | N/A | 23626 | 26543 | 118.8 | 27746 | 128.7 |
| flvmeta/flvmeta | 397 | 400 | 401 | 401 | 403 | 403 | 399 | N/A | N/A | 395 | N/A | N/A | 404 | 385 | -3.8 | 402 | 0.4 |
| freetype2/ftfuzzer | 7631 | 7924 | 7865 | 7909 | 7906 | 7861 | 8406 | N/A | N/A | 7418 | N/A | N/A | 7455 | 6919 | -11.4 | 10565 | 35.3 |
| harfbuzz/hb-shape-fuzzer | 13649 | 14506 | 14484 | 14182 | 14477 | 14636 | 13870 | N/A | N/A | 13006 | N/A | N/A | 14092 | 10902 | -22.6 | 14637 | 3.9 |
| jasper/imginfo | 2168 | 2281 | 2345 | 2296 | 2230 | 2306 | 2267 | N/A | N/A | 2168 | N/A | N/A | 2294 | 1888 | -16.5 | 2378 | 5.2 |
| jhead/jhead | 259 | 262 | 263 | 262 | 263 | 263 | 740 | 258 | N/A | 707 | N/A | N/A | 688 | 814 | 152.3 | 962 | 198.2 |
| jq/jq | 2146 | 2167 | 2161 | 2167 | 2166 | 2165 | 2141 | N/A | N/A | 2160 | N/A | N/A | 2174 | 2187 | 1.2 | 2187 | 1.2 |
| jsoncpp/jsoncpp_fuzzer | 664 | 667 | 667 | 667 | 666 | 666 | 670 | N/A | N/A | 663 | N/A | 671 | 667 | 677 | 1.5 | 677 | 1.5 |
| lame/lame | 3964 | 3643 | 3984 | 3775 | 3984 | 3984 | 3954 | 3948 | N/A | 2996 | N/A | N/A | 3379 | 4036 | 8.2 | 4097 | 9.8 |
| lcms/cms_transform_fuzzer | 681 | 1175 | 1403 | 1477 | 1360 | 1370 | 1561 | N/A | N/A | 1306 | N/A | N/A | 666 | 1750 | 57.2 | 2103 | 88.9 |
| libjpeg/turbo_fuzzer | 2028 | 2100 | 2143 | 2066 | 2152 | 2082 | 2333 | 2008 | N/A | 2064 | N/A | N/A | 2296 | 2412 | 13.6 | 2469 | 16.3 |
| libpcap/fuzz_both | 1366 | 2716 | 2897 | 2915 | 2904 | 2933 | 1520 | 1471 | N/A | 2030 | N/A | N/A | 3126 | 3740 | 72.9 | 3780 | 74.8 |
| libpng/read_fuzzer | 1822 | 1750 | 1858 | 1865 | 1826 | 1868 | 1845 | 1832 | N/A | 1779 | 1866 | N/A | 1896 | 1986 | 8.2 | 2009 | 9.4 |
| libsndfile/sndfile_fuzzer | 3122 | 3264 | 3186 | 3142 | 3187 | 3194 | 3138 | N/A | N/A | 3565 | N/A | N/A | 3219 | 3323 | 3.2 | 3982 | 23.7 |
| libtiff/tiffsplit | 3643 | 3828 | 3829 | 3905 | 3852 | 3859 | 3649 | N/A | N/A | 3932 | N/A | N/A | 4058 | 4256 | 11.0 | 4281 | 11.6 |
| libtiff/tiff2ps | 4396 | 4569 | 4647 | 4700 | 4665 | 4673 | 4334 | N/A | N/A | 4379 | N/A | N/A | 4782 | 5080 | 11.2 | 5195 | 13.8 |
| libxml2/xml | 13062 | 13526 | 13586 | 13486 | 13595 | 13628 | 13064 | N/A | N/A | N/A | N/A | 13756 | 13189 | 10612 | -21.0 | 13837 | 3.0 |
| lua/lua | 4549 | 5026 | 4666 | 4530 | 4598 | 4583 | 4719 | N/A | N/A | 4307 | N/A | N/A | 4717 | 6034 | 30.4 | 6105 | 32.0 |
| mbedtls/fuzz_dtlsclient | 2508 | 2535 | 2593 | 2570 | 2563 | 2579 | 2670 | N/A | N/A | 2309 | N/A | N/A | 2602 | 1523 | -40.1 | 2514 | -1.2 |
| mp3gain/mp3gain | 486 | 490 | 484 | 484 | 488 | 487 | 518 | 426 | N/A | 458 | N/A | N/A | 503 | 513 | 6.6 | 552 | 14.7 |
| mruby/mruby_fuzzer | 8629 | 8912 | 9103 | 9464 | 8911 | 8681 | 8900 | N/A | 8991 | 6933 | N/A | N/A | 9229 | 11183 | 28.3 | 11589 | 32.9 |
| mujs/mujs | 3187 | 3188 | 3305 | 3307 | 3294 | 3254 | 3231 | N/A | 3291 | 3272 | N/A | N/A | 3530 | 5212 | 58.7 | 5252 | 60.0 |
| ncurses/infotocap | 1919 | 2393 | 2157 | 1912 | 2134 | 2184 | 1896 | N/A | N/A | 1898 | N/A | N/A | 2400 | 2566 | 23.3 | 2641 | 26.9 |
| openh264/decoder_fuzzer | 8333 | 8560 | 7846 | 7497 | 8015 | 7999 | 8234 | N/A | N/A | 7234 | N/A | N/A | 8726 | 9369 | 16.8 | 9581 | 19.4 |
| openssl/x509 | 8070 | 8070 | 8075 | 8058 | 8001 | 8043 | 8091 | N/A | N/A | 8035 | N/A | N/A | 8080 | 6488 | -19.5 | 8131 | 0.9 |
| php/php-fuzz-parser | 16672 | 16377 | 16621 | 16562 | 16602 | 16451 | 16341 | N/A | 16614 | N/A | N/A | N/A | 16523 | 12969 | -21.5 | 17938 | 8.5 |
| pixbuf/pixdata | 332 | 332 | 332 | 332 | 332 | 332 | 332 | N/A | N/A | 332 | N/A | N/A | 389 | 494 | 46.4 | 497 | 47.3 |
| poppler/pdf_fuzzer | 3357 | 3349 | 3221 | 3373 | 3140 | 3135 | 3596 | 4217 | N/A | 3377 | 6472 | N/A | 5936 | 10692 | 189.2 | 11302 | 205.7 |
| proj4/proj_crs_to_crs_fuzzer | 6467 | 6507 | 7167 | 6934 | 6820 | 6799 | 6554 | N/A | N/A | 6196 | N/A | N/A | 7237 | 7690 | 14.3 | 10417 | 54.9 |
| re2/re2_fuzzer | 3328 | 3335 | 3369 | 3346 | 3349 | 3372 | 3352 | N/A | N/A | 3346 | N/A | 3387 | 3391 | 3385 | 0.8 | 3400 | 1.3 |
| sqlite3/ossfuzz | 12402 | 12287 | 14684 | 13792 | 13324 | 13572 | 12381 | N/A | N/A | 10729 | N/A | 18954 | 14862 | 20122 | 50.0 | 20303 | 51.3 |
| stb/stbi_read_fuzzer | 1326 | 1441 | 1482 | 1505 | 1478 | 1473 | 1351 | N/A | N/A | 1466 | N/A | N/A | 1897 | 2265 | 53.3 | 2286 | 54.7 |
| tcpdump/tcpdump | 10701 | 12292 | 12937 | 11599 | 11914 | 12877 | 13305 | 11434 | N/A | 12046 | N/A | N/A | 13518 | 9966 | -18.3 | 15651 | 28.3 |
| vorbis/decode_fuzzer | 1327 | 1353 | 1322 | 1371 | 1356 | 1354 | 1298 | N/A | N/A | 1299 | N/A | N/A | 1378 | 536 | -60.0 | 1315 | -1.8 |
| woff2/convert_woff2ttf_fuzzer | 1293 | 1299 | 1333 | 1315 | 1302 | 1337 | 1382 | N/A | N/A | 1222 | N/A | N/A | 1246 | 1551 | 19.2 | 1610 | 23.7 |
| xpdf/pdftotext | 3769 | 3989 | 3450 | 3341 | 3371 | 3389 | 3763 | 4103 | N/A | 3279 | 5015 | N/A | 4312 | 7422 | 98.5 | 8471 | 126.6 |
| zlib/uncompress_fuzzer | 420 | 432 | 435 | 436 | 435 | 437 | 420 | 421 | N/A | 429 | 432 | N/A | 448 | 460 | 6.7 | 460 | 6.7 |
| Avg Coverage Imp | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 78.2 | - | 98.3 |

GT=Gramatron, FT=Fuzztruction, GenEvo(s)=GenEvo(standalone). The best coverage is in red, the second best is in blue.

branches covered by each fuzzer in 24 hours in Table 1. In addition, we report the average percentage improvement of GenEvo(standalone) and GenEvo over the baselines (i.e., *Imp(%)*).

In comparison to the baselines, GenEvo(standalone) and GenEvo covered an average of 78.2% and 98.3% more code branches across the evaluated subjects, respectively. Specifically, GenEvo, even in its standalone mode, outperformed all baselines in 33 out of 49 subject programs. It is worth noting that GenEvo(standalone) was executed with only one fuzzer instance, thus producing far fewer test inputs than the baselines running with three fuzzer instances. When integrating GenEvo with conventional fuzzers and executing with the same number of fuzzer instances as that of baselines, the fuzzing performance was further improved substantially, where it performs the best in 46 out of the 49 subject programs.

When comparing GenEvo(standalone) with GenEvo, we had some interesting findings. Although it is generally assumed that using more fuzzer instances yields better performance, this does not always hold true in our scenario. For the programs that accept highly-structured input formats (e.g., `lua` and `sql`), a standalone GenEvo already performs remarkably well, and GenEvo with two additional instances has little performance gains. In contrast, for programs with loosely structured formats (e.g., `elf`), GenEvo significantly outperforms GenEvo(standalone). This reason is intuitive: conventional fuzzers tend to break the input format, which

**Table 2: Bugs detected during the experiment.**

| Package | Bug Type | File | Num | Status |
|---|---|---|---|---|
| bloaty | invalid memory read | debug_info.h | 1 | Known |
| bento4 | assertion fail | Ap4Atom.cpp | 1 | Reported |
| | division by zero | Ap4Atom.cpp | 1 | Fixed |
| | heap-buffer-overflow | Ap4ByteStream.cpp | 1 | Fixed |
| | heap-buffer-overflow | Ap4StdCFilByteStr.cpp | 1 | Fixed |
| | heap-buffer-overflow | Mp42Hevc.cpp | 1 | Reported |
| | memory leak | Ap4File.cpp | 1 | Reported |
| | memory leak | Ap4List.h | 1 | Reported |
| | memory leak | Ap4Movie.cpp | 1 | Reported |
| | memory leak | Ap4Track.cpp | 1 | Reported |
| | exception std::bad_alloc | AP4DataBuffer.cpp | 1 | Reported |
| mp3gain | heap-buffer-overflow | apetag.c | 5 | Reported |
| | stack-buffer-overflow | apetag.c | 1 | Reported |
| mujs | invalid memory read | jsarray.c | 1 | Reported |
| | heap-buffer-overflow | jsarray.c | 2 | Reported |
| | heap-use-after-free | jsvalue.c | 1 | Reported |
| mruby | invalid memory read | string.c | 1 | Known |
| xpdf | heap-buffer-overflow | GString.h | 1 | Fixed |
| lua | heap-buffer-overflow | lutf8lib.c | 1 | Comfirmed, CVE |
| ncurses | global-buffer-overflow | lib_tparm.c | 1 | Reported |
| | heap-buffer-overflow | captoinfo.c | 1 | Reported |
| pixbuf | invalid memory write | io-jpeg.c | 1 | Reported |
| | allocation-size-too-big | gdk-pixbuf.c | 1 | Reported |

**Table 3: Number of bugs detected by evaluated fuzzers.**

| Package | A | M | A(e) | A(f) | A(q) | A(m) | SC | AS | GT | WZ | FT | EL | G2 | GE(s) | GE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bloaty | **1** | **1** | **1** | **1** | **1** | **1** | **1** | 0 | N/A | 0 | N/A | N/A | **1** | 0 | **1** |
| bento4 | 0 | 4 | 4 | 4 | 4 | 4 | 6 | 4 | N/A | 0 | N/A | N/A | 6 | 7 | **10** |
| mp3gain | 3 | 2 | 3 | 3 | 3 | 2 | 3 | 3 | N/A | 0 | N/A | N/A | 2 | 0 | **5** |
| mujs | 1 | 0 | 2 | 1 | 2 | 1 | 1 | N/A | 1 | 0 | N/A | N/A | 1 | 3 | **4** |
| mruby | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A | 0 | 0 | N/A | N/A | 0 | **1** | **1** |
| xpdf | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | N/A | 0 | 0 | N/A | **1** | **1** | **1** |
| lua | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A | N/A | 0 | N/A | N/A | 0 | **1** | **1** |
| ncurses | 0 | 0 | 1 | 1 | 0 | 1 | 0 | N/A | N/A | 0 | N/A | N/A | 0 | 1 | **2** |
| pixbuf | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N/A | N/A | 0 | N/A | N/A | 0 | 0 | **2** |
| Total | 6 | 8 | 12 | 11 | 11 | 10 | 12 | 8 | 1 | 0 | 0 | 0 | 11 | 14 | **27** |

A=AFL, M=MOPT, A(e)=AFL++(explore), A(f)=AFL++(fast), A(q)=AFL++(quad)
A(m)=AFL++(mmopt), SC=SymCC, AS=AFLSmart, GT=Gramatron, WZ=Weizz
FT=Fuzztruction, EL=ELFuzz, G2=G2Fuzz, GE(s)=GenEvo(standalone), GE(h)=GenEvo.

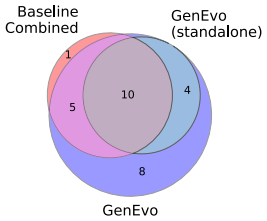



**Figure 7: Venn diagram of detected bugs by different fuzzers.**

thus have little additional benefit to GenEvo, even with additional fuzzer instances. These results suggest that when fuzzing loosely-structured inputs, GenEvo combined with conventional fuzzers can show superior performance, while GenEvo alone is already effective for fuzzing highly-structured inputs.

Although GenEvo has shown strong potential, we also noted that in some of the programs (e.g., `freetype2`, `harfbuzz`, `jasper`, and `libxml2`), GenEvo in its standalone configuration underperforms compared to the baseline tools. Our manual analysis reveals the following reasons: (1) In some programs (e.g., `php`, `libxml2`), the provided seed corpora are exceptionally high-quality, containing numerous manually crafted inputs that cover diverse semantic cases (specifically, the corpora contain 5071 input seeds for `php` and 203 for `libxml2`). As a pure generator-based fuzzer, GenEvo(standalone) cannot leverage these existing seeds, placing it at an initial disadvantage compared to mutation-based fuzzers. However, even in these scenarios, GenEvo explores distinct branches not covered by conventional methods. (2) For programs with loosely structured inputs (e.g., `jasper` and `tcpdump`), LLM-based generator solving suffers from the high latency of LLM calls, which thus limits its performance.

## 4.3 Bug Detection (RQ.2)

Bug detection capability is another key indicator of a fuzzer's effectiveness, and we thus evaluated it in this section. To identify unique crashes, AFL uses edge-coverage hashes [6], which do not accurately correspond to distinct bugs. To enable more precise classification, we use the AddressSanitizer-reported trigger locations to group unique crashes into unique bugs. For memory leaks, we classify based on the allocation site of the leaked object.

Table 2 lists all bugs detected during our experiments. Among the 49 subject programs, the evaluated fuzzers discovered a total of 28 bugs in 12 subject programs (from 9 packages), covering a wide range of bug categories, including memory corruption (e.g., buffer overflows and use after free) and assertion failures.

Table 3 summarizes the number of unique bugs detected by each fuzzer across the subject programs. Weizz did not detect any of these bugs because it is a binary-only fuzzer that is not compatible with AddressSanitizer, which is required to detect most of these bugs. Figure 7 presents a Venn diagram that illustrates the overlap in bugs detected among different approaches. Of the 28 bugs, 16 were detected by at least one baseline fuzzer, with a maximum of 12 bugs detected by a single baseline. In contrast, GenEvo(standalone) discovered 14 bugs, while GenEvo detected 27 bugs, covering nearly all identified ones. GenEvo(standalone) with a single instance found more bugs than a single baseline, and it also uncovered different ones. When combining it with conventional fuzzers, GenEvo significantly outperformed the standalone GenEvo in bug detection. The main factor for this improvement is that GenEvo in the hybrid configuration running with three fuzzer instances produces much more test inputs than the standalone GenEvo with a single instance. Another factor is that the hybrid configuration leverages carefully designed input-level mutation strategies optimized for bug discovery. GenEvo missed one bug out of all detected bugs, which we attribute to the randomness of fuzzing. A case study of discovered bugs is provided in the supplementary material.

## 4.4 Ablation (RQ.3)

*4.4.1 Random Mutation.* To assess the impact of the random mutation strategy, we conduct an ablation study with the following configurations, both using the same GPU resources:

- GenEvo(solving): Targeted generator solving only.
- GenEvo(random): Random mutations only.

We conduct experiments on three programs: `sqlite3`, `libpng`, and `objdump`, which represent high, medium, and low levels of input structural complexity, respectively. The branch coverage results

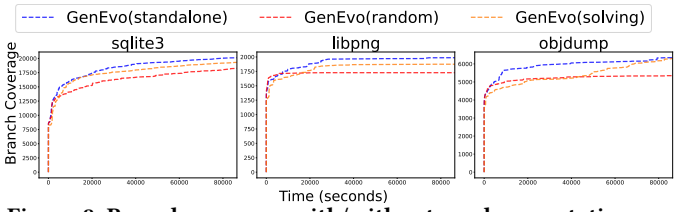



**Figure 8: Branch coverage with/without random mutation.**

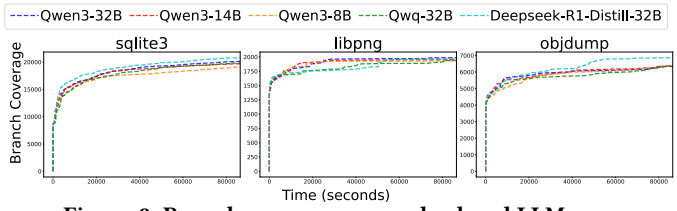



**Figure 9: Branch coverage across backend LLMs.**

are shown in Figure 8. GenEvo(random) demonstrates rapid initial coverage growth, highlighting the efficiency of random mutations in exploring program behaviors without costly analysis. However, it quickly plateaus as it lacks awareness of the program's internal logic. In contrast, GenEvo(solving) steadily increases coverage over time but at a slower pace because of computational overhead. These results confirm that random mutation is an effective mechanism for fast exploration, complementing targeted solving to improve overall fuzzing performance.

*4.4.2 Impact of LLM Choice.* To assess the impact of the backend LLM on fuzzing performance, we evaluate GenEvo with four alternative models: two smaller variants from the Qwen3 series (Qwen3-14B and Qwen3-8B), Qwq-32B [4], and Deepseek-R1-Distill-32B [19]. This evaluation is conducted on the same set of programs used in Section 4.4.1. To eliminate the effect of differing response latencies across model sizes, we adjust the size of the ongoing request queue for each model to ensure comparable response times. The branch coverage shown in Figure 9 indicates that while more capable models tend to perform better, the gap is relatively small, suggesting reduced reliance on model capability.

## 5 Conclusion

In this paper, we introduced hybrid generator space fuzzing, a new fuzzing method that combines constraint-guided exploration from hybrid fuzzing with the structural guarantees of generator-based fuzzing. We implemented this idea in GenEvo, where an LLM is used to perform constraint-guided generator solving, complemented by lightweight mutation for efficient exploration. To make this process effective in practice, we further developed a prompting strategy that integrates specification-grounded exemplars and retrieved program context, improving both the correctness and applicability of generated generators. Our evaluation on a diverse set of real-world programs demonstrates that GenEvo consistently improves code coverage and vulnerability discovery over a wide range of baselines. These results suggest that operating in the space of input-generation logic, combined with LLM-driven solving, is a promising direction for advancing fuzzing on complex, structured inputs.

## 6 Data Availability Statement

The code of GenEvo is available[2].

## Acknowledgments

We thank anonymous reviewers for their insights on improving this work. This work is supported in part by the National Natural Science Foundation of China (NSFC) (Grant No. 62232016).

## References

[1] [n. d.]. AFLplusplus/custom_mutators/gramatron. https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/gramatron
[2] [n. d.]. Dictionaries. https://github.com/google/AFL/tree/master/dictionaries
[3] [n. d.]. Qwen/Qwen3-32B-AWQ. https://huggingface.co/Qwen/Qwen3-32B-AWQ
[4] [n. d.]. QwQ-32B: Embracing the Power of Reinforcement Learning. https://qwenlm.github.io/blog/qwq-32b/
[5] [n. d.]. Source Level Debugging with LLVM. https://llvm.org/docs/SourceLevelDebugging.html
[6] [n. d.]. technical_details. https://lcamtuf.coredump.cx/afl/technical_details.txt
[7] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. 2023. Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge. In *Proceeding of the 32nd USENIX Security Symposium (USENIX Security)*. 1847–1864.
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043.
[9] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*. 725–741.
[10] Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. 2025. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. arXiv:2506.10323 [cs.CR] https://arxiv.org/abs/2506.10323
[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266
[12] Zhen Yu Ding and Claire Le Goues. 2021. An empirical study of oss-fuzz bugs. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142.
[13] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *Proceedings of the 12nd International Symposium on Search Based Software Engineering (SSBSE)*. 105–120.
[14] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 6491–6501.
[15] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2019. WEIZZ: automatic grey-box fuzzing for structured binary formats. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2019).
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX workshop on offensive technologies (WOOT)*.
[17] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2 (2023), 1.
[18] Google. [n. d.]. AFL/docs/parallel_fuzzing.txt. https://github.com/google/AFL/blob/master/docs/parallel_fuzzing.txt
[19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the 2020 ACM on Measurement and Analysis of Computing Systems (POMACS)* 4, 3 (2020), 1–29.
[21] Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. 2023. Co-dependence aware fuzzing for dataflow-based big data analytics. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1050–1061.
[22] Soha Hussein, Stephen McCamant, and Mike Whalen. 2026. Generator-Based Fuzzers with Type-Based Targeted Mutation. arXiv:2406.02034 [cs.SE] https://arxiv.org/abs/2406.02034

[23] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and improving hybrid fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 410–422.

[24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ArXiv* abs/2310.06770 (2023).

[25] Roman Kraus, Hoang Lam Nguyen, and Martin A. Schneider. 2024. Generator-based Fuzzing with Input Features. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing* (Lisbon, Portugal) *(SBFT '24)*. Association for Computing Machinery, New York, NY, USA, 13–20. doi:10.1145/3643659.3643925

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. 75–86.

[28] Ao Li, Madonna Huang, Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2025. The Havoc Paradox in Generator-Based Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 35, 1, Article 29 (Dec. 2025), 26 pages. doi:10.1145/3742894

[29] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1 (2018), 1–13.

[30] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 2777–2794.

[31] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: facilitating seed input generation in fuzzing via string constraint solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 391–403.

[32] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. [n. d.]. Agentic Concolic Execution.

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

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

[35] Ruijie Meng, Gregory J Duck, and Abhik Roychoudhury. 2026. Large language model assisted hybrid fuzzing. *IEEE Transactions on Software Engineering (TSE)* (2026).

[36] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1393–1403.

[37] Charlie Miller and Zachary N. J. Peterson. 2007. Analysis of Mutation and Generation-Based Fuzzing.

[38] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin T. Vechev. 2024. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[39] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

[40] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University* (2012).

[41] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2019), 1980–1997.

[42] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 181–198.

[43] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[44] Fayozbek Rustamov, Juhwan Kim, Jihyeon Yu, and Joobeom Yun. 2021. Exploratory review of hybrid fuzzing for automated vulnerability detection. *IEEE Access* 9 (2021), 131166–131190.

[45] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 2795–2809.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX annual technical conference (USENIX ATC)*. 309–318.

[47] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. 2023. {AIFORE}: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. 4967–4984.

[48] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 244–256.

[49] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16.

[50] Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[51] Qwen Team. 2025. Qwen3. https://qwenlm.github.io/blog/qwen3/

[52] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. 2026. Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3509–3527. doi:10.1109/SP63933.2026.00110

[53] Siwei Wei and Yan Cai. 2026. Generator Solving for Symbolic Execution. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*.

[54] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 1634–1645.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[56] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. 2025. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 6299–6318. https://www.usenix.org/conference/usenixsecurity25/presentation/yang-yupeng

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

[58] M. Zalewski. [n. d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl

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

[60] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. *Proceedings of the 34th USENIX Security Symposium (USENIX Security)* (2025).

[61] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 722–733.

[62] Tao Zhang, Yu Jiang, Runsheng Guo, Xiaoran Zheng, and Hui Lu. 2020. A survey of hybrid fuzzing based on symbolic execution. In *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies (CIAT)*. 192–196.

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