EvolFuzz: Evolving AFL++ Custom Mutators via LLM-Guided Program Synthesis

Ron Yifeng Wang Xiaoyue Wang
Stanford University

Abstract

Coverage-guided greybox fuzzers like AFL++ rely on hand-tuned mutation heuristics that are general-purpose but not optimized for any particular target. We introduce **EvolFuzz**, a system that uses LLM-guided evolutionary program synthesis to automatically discover target-specific mutation strategies. Our system evolves C-language custom mutators within an AlphaEvolve-style MAP-Elites framework, using edge coverage as the fitness signal and AFL++'s plugin API to load evolved strategies at runtime. On five target applications spanning binary and text formats, evolved mutators achieve 1–32% coverage improvement over vanilla AFL++, with the largest gains on structured binary formats: 5–6% on libpng, libtiff, and libxml2, and 32% on OpenSSL's DER/ASN.1 parsing. On MAGMA's ground-truth bug-finding benchmark, the libpng-specialized mutator triggers one additional bug, and our trigger rate analysis shows that evolved mutators are 2–28× more likely to trigger vulnerabilities per code reach across all targets. To our knowledge, **EvolFuzz** is the first study to apply evolutionary program synthesis to automatically design fuzzer mutation strategies, demonstrating that LLM-guided evolution can complement decades of manual heuristic engineering.

1 System Description

We present **EvolFuzz**, a system that uses LLM-guided evolutionary program synthesis to automatically generate custom mutation strategies for coverage-guided greybox fuzzing (CGF). Rather than relying on hand-tuned mutation heuristics, our system uses a large language model to discover effective target-specific mutation strategies within an AlphaEvolve-style evolutionary framework. We implement our system with AFL++ [3], the state-of-the-art CGF tool. By leveraging its custom mutator plugin API, we are able to load evolved strategies at runtime. Below, we describe our method, target bug classes, and target applications.

Bug-Finding Method. Coverage-guided greybox fuzzing works by instrumenting target programs with lightweight edge coverage tracking, then iteratively mutating inputs and retaining those that discover new code paths. While standard CGF tools apply a fixed set of mutation operators, such as bit flips, arithmetic, block operations, and splicing with static probability distributions, our system uses a learned mutation strategy that is specialized for a specific target program.

Target Bug Classes. Our system targets memory safety vulnerabilities in C/C++ programs, the same class of bugs detected by existing greybox fuzzers. To capture this capability, We evaluate on the MAGMA benchmark [5], which instruments real-world programs with ground-truth bug canaries that track whether vulnerabilities are *reached* (code near the bug is executed) and *triggered* (the specific buggy condition is satisfied). Example bug types include buffer overflows, integer overflows, use-after-free, null pointer dereferences, and format-specific parsing errors.

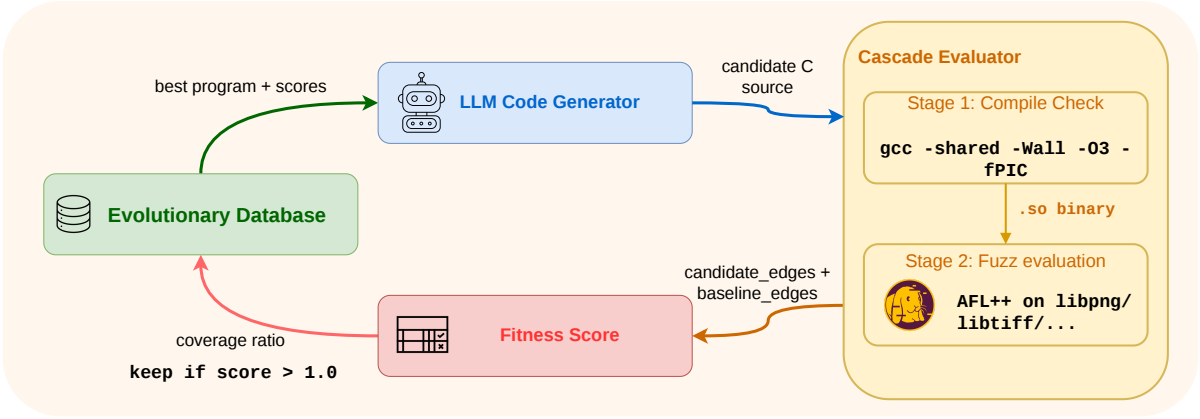


Figure 1 EvolFuzz system architecture. The evolutionary database provides parent programs to the LLM, which generates candidate C mutator source code. The cascade evaluator compiles the candidate and runs AFL++ for 30 seconds. The resulting coverage ratio determines whether the candidate enters the population.

Target Applications. We evaluate on five applications from the MAGMA benchmark, spanning a diverse range of input (binary vs. text), parsing complexity (image vs. cryptographic vs. structured), and bug counts (7–20). Table 1 presents the target application details.

Target	Version	Format	MAGMA Bugs	Harness Source
libpng	a37d4836	Binary (PNG image)	7	FuzzBench / Google
libtiff	c145a6c1	Binary (TIFF image)	14	OSS-Fuzz
sqlite3	8c432642	Text (SQL)	20	OSS-Fuzz
libxml2	ec6e3efb	Text (XML)	17	Chromium / OSS-Fuzz
OpenSSL	3bd5319b	Binary (DER/ASN.1)	20	OpenSSL built-in

Table 1 Target applications and MAGMA bug counts. Versions are pinned commits from the MAGMA benchmark suite. Harness source indicates the origin of the `LLVMFuzzerTestOneInput` wrapper that bridges between the fuzzer and the library under test; all harnesses are from production fuzzing infrastructure.

2 Technical Approach

Figure 1 illustrates the EvolFuzz architecture. The system operates as a closed loop between four components: an evolutionary database that maintains a population of candidate mutators, an LLM code generator that produces new candidates, a cascade evaluator that measures coverage improvement, and a fitness scorer that determines population membership.

2.1 Evolutionary Database

The evolutionary database (Figure 1, left) maintains a diverse population of mutator programs using MAP-Elites [4] with an island model (80 programs across 5 islands). MAP-Elites partitions the search space along feature dimensions (code complexity and behavioral diversity) and retains the best program per cell. The key benefit is that it preserves structural diversity: a short mutator that excels on one strategy can coexist with a longer one that uses a different approach. The database provides the LLM with diverse parent programs and their fitness scores, allowing it to reference multiple high-performing strategies when generating new candidates.

2.2 LLM Code Generator

The LLM (Figure 1, top) receives parent programs from the database and generates candidate C mutator source code. We use a diff-based approach where the LLM is only allowed to modify code within `EVOLVE-BLOCK` markers, while keeping the remaining AFL++ scaffolding (e.g., includes, struct definitions, buffer management) intact. This constrains the search space to mutation strategy and ensures the generated code always satisfies the API contract.

One key design choice we made is evolving mutators in C rather than Python. C mutators receive a pointer to AFL++'s internal state at initialization, providing direct access to runtime information such as auto-discovered dictionary tokens, corpus metadata, and execution statistics (see Appendix A for details). Python mutators lack this access entirely. Our C vs. Python ablation (Table 2(a)) confirms that this richer information, rather than execution throughput, drives the improvement.

2.3 Cascade Evaluator

The cascade evaluator (Figure 1, right) first compiles the candidate C source into a shared library ($<$1s), rejecting any program with syntax errors or type mismatches. Candidates that compile successfully are loaded into AFL++ as the sole mutation engine, replacing the built-in havoc stage, and AFL++ fuzzes the target for 30 seconds using a deterministic seed. The evaluator records the number of unique edges discovered; candidates scoring below $0.5\times$ baseline are rejected without entering the population.

2.4 Fitness Scoring

The fitness scorer (Figure 1, bottom) computes the ratio of edges discovered by the candidate mutator to edges discovered by vanilla AFL++ on the same target:

$$\text{coverage ratio} = \frac{\text{candidate\_edges}}{\text{baseline\_edges}}$$

Baseline edges are pre-computed from vanilla AFL++ runs (median of 3). A score greater than 1.0 indicates the evolved mutator discovers more code paths than vanilla.

A single evolution run (Docker build, baseline computation, and 30 iterations on one target) completes in under 1 hour. Setup details are in Appendix A.

3 Evaluation Methodology

3.1 Metrics

We measure two complementary metrics. The **coverage ratio** (candidate_edges/baseline_edges) measures how many edge transitions the evolved mutator discovers relative to vanilla AFL++ on the same target. A ratio $>$1.0 indicates the evolved mutator explores more code paths. We use this as the fitness signal during evolution (at 30s) and validate at longer durations.

For bug-finding, we use the **MAGMA benchmark**, which instruments real programs with canaries at known vulnerability sites. For each bug, MAGMA records whether it is *reached* (code near the bug was executed) and *triggered* (the specific buggy condition was satisfied). We additionally compute the *trigger rate* (triggered/reached), which measures mutation quality independent of exploration volume: a higher trigger rate means the mutator produces inputs that more efficiently satisfy vulnerability conditions.

3.2 Evaluation Infrastructure

In designing and implementing our evaluation infrastructure, one major challenge was measurement noise, a common issue in fuzzing research. We address this with three techniques. First, we utilize AFL++'s deterministic seed mode, which eliminates random variance and allows a single trial to reliably measure

Mutator	Score
Evolved C	**1.026**×
Evolved Python	0.996×

Mutator	MUTATOR_ONLY	Supplement
Seed C (hand-written)	1.003×	**1.029**×
Evolved C (Gemini)	**1.026**×	1.001×
Evolved Python	0.996×	1.000×

Table 2 **(a)** C vs. Python: access to AFL++ internal state drives the improvement, not throughput. **(b)** MUTATOR_ONLY vs. supplement: evolution helps only when the custom mutator is the sole mutation source.

Iter	Score	Key Change
0	1.000×	Seed mutator (baseline)
1	1.010×	Minor probability adjustments
2	1.044×	Insert-or-overwrite for dictionary tokens
8	1.053×	Endian-aware values + increased dictionary weight

Table 3 Evolution trajectory for libpng. The best mutator (1.053×) was found at iteration 8 (of 30). 83% of all candidates beat baseline.

coverage differences as small as 1%.[1] Second, we pre-compute baselines by running vanilla AFL++ once per target (median of 3 trials) and reusing the result across all evaluations. Finally, all experiments run on a single dedicated machine (GCP c4d instance with 32 vCPUs), eliminating hardware heterogeneity.[2] Together, these techniques ensure reproducibility while reducing each evolution iteration to ∼1 minute.

To ensure our results are meaningful and comparable to standard fuzzing benchmarks, we adopt the same toolchain used by FuzzBench and OSS-Fuzz: targets are compiled with `afl-clang-fast` and fuzz harnesses are sourced directly from these benchmarks (see Table 1). The main difference from FuzzBench is that we use fork-server execution rather than persistent mode, which reduces absolute throughput but does not affect relative comparisons, since both the evolved mutator and the vanilla baseline run in the same mode. A detailed comparison is provided in Appendix D.

3.3 Ablation Studies

Our system involves several design choices that each affect performance. In Section 4, we isolate three of these: C vs. Python mutators, MUTATOR_ONLY vs. supplement mode, and single-target vs. multi-target evolution. These ablations inform the configuration used for the full benchmark evaluation. In Section 5, we additionally perform trigger rate analysis to evaluate whether evolved mutators produce higher-quality mutations that more efficiently satisfy bug conditions, beyond simply counting triggered bugs.

4 Basic Evaluation

Before evaluating **EvolFuzz** at scale, we validate three fundamental design choices: (1) should we evolve C or Python mutators? (2) should the evolved mutator replace AFL++'s built-in mutations entirely, or supplement them? and (3) does specializing for a single target outperform evolving across multiple targets?

For each ablation, we use Gemini-3.1-Pro to evolve mutators for 30 iterations with 30s evaluations, then validate the best candidates at 300s with 5 deterministic seeds. We answer the first two questions using a multi-target setting, where the mutator is evolved using scores from libpng, zlib, and libjpeg; we then compare this multi-target approach against single-target specialization. These findings inform the configuration we use in the full benchmark evaluation in Section 5.

[1] We found that with deterministic seeds, repeated runs on the same target produce identical edge counts (CV $< 0.1\%$ at 30s in our experiments). Without this, AFL++'s internal randomness introduces 1–3% variance, which obfuscates the improvements we aim to measure.

[2] Our initial experiments used Modal, a serverless cloud platform where each evaluation runs on a different physical machine. Hardware differences across the fleet introduced 3–4% measurement bias. Switching to a single dedicated VM eliminated this noise entirely.

Seed (before)

```
/* Interesting values: native
   byte order only */
*(int16_t *)(out + pos) =
  my_interesting_16[
    fast_rand_below(NUM_INTERESTING_16)
  ];

/* Splice from add_buf:
   always overwrites */
size_t dst_pos = min_pos +
  fast_rand_below(
    cur_len - splice_len - min_pos + 1);
memcpy(out + dst_pos,
       add_buf + src_pos, splice_len);
```

Evolved (after)

```
/* Interesting values: 50% chance
   of byte-swap (big-endian) */
uint16_t val = my_interesting_16[
    fast_rand_below(NUM_INTERESTING_16)];
if (fast_rand_below(2))
  val = __builtin_bswap16(val);
*(uint16_t *)(out + pos) = val;

/* Splice: 50% insert (preserves
   existing bytes), 50% overwrite */
if (fast_rand_below(2) &&
    cur_len + splice_len <= max_size) {
  memmove(out + dst + splice_len,
    out + dst, cur_len - dst);
  memcpy(out + dst,
    add_buf + src_pos, splice_len);
  out_size += splice_len;
} else { /* overwrite as before */ }
```

Figure 2 Key code changes made by the LLM. **(a)** Seed mutator. **(b)** Evolved mutator. Top: endian-aware interesting values. Bottom: insert-or-overwrite splicing (inserting preserves existing structure rather than destroying it).

4.1 C vs. Python Mutators

Our first question is whether to evolve C or Python mutators. Both are supported by AFL++'s custom mutator API, but C mutators receive a pointer to AFL++'s internal state (dictionary tokens, corpus metadata) while Python mutators do not. Table 2(a) shows the C mutator outperforms Python (1.026× vs. 0.996×) despite having comparable throughput (∼5500 exec/s). This shows having access to AFL++ internal state is beneficial to mutator improvement.

4.2 MUTATOR_ONLY vs. Supplement Mode

Next, we study whether the evolved mutator should fully replace AFL++'s built-in havoc stage (`MUTATOR_ONLY`) or run alongside it (supplement mode). In supplement mode, AFL++ alternates between its own mutations and the custom mutator. Table 2(b) shows that the evolved C mutator achieves 1.026× in MUTATOR_ONLY mode, a clear improvement over vanilla, but only 1.001× in supplement mode. The reason is that in supplement mode, AFL++'s built-in havoc is already running and provides strong coverage; the custom mutator's additional mutations offer little marginal benefit, making the improvement signal perhaps too weak for evolution to optimize against. In MUTATOR_ONLY mode, the evolved mutator is solely responsible for all mutations, giving evolution a strong and direct signal to optimize.

4.3 Single-Target vs. Multi-Target Ablation

We also investigate whether target specialization matters. We compare the multi-target mutator (evolved across libpng+zlib+libjpeg) against the single-target mutator (libpng only) and evaluate both on MAGMA's libpng benchmark. The multi-target mutator achieves no coverage improvement on libpng (1.000× at 300s), while the single-target mutator (evolved for libpng only) achieves 1.053×. We think the reason is that fitness signal becomes diluted when optimizing across targets with conflicting format requirements.

4.4 Single-Target Walkthrough: libpng

We now demonstrate single-target evolution on libpng, using C mutators in MUTATOR_ONLY mode as our configuration. Table 3 shows the progression of the running best program.

4.4.1 What the Evolved Mutator Changed

Figure 2 compares the seed and evolved mutators side-by-side to show the two most impactful changes. In total, the LLM made four categories of modifications:

Target	Format	Baseline	Best	Gain	Best At
libpng	Binary (PNG)	827	871	1.053×	Iter 8
libtiff	Binary (TIFF)	1,687	1,779	1.055×	Iter 2
sqlite3	Text (SQL)	4,430	4,492	1.014×	Iter 6
libxml2	Text (XML)	3,299	3,482	1.056×	Iter 26
openssl	Binary (DER)	815	1,076	1.320×	Iter 27

Table 4 Single-target evolution coverage results. Edges are AFL++ edge bitmap counts at 30s with deterministic seed.

1. **Endian-aware multi-byte values.** The seed mutator writes interesting values (e.g., 0, 256, 32767) in native little-endian byte order only. On an x86 machine, the value 256 is stored as `0x00 0x01`. But PNG uses big-endian (network byte order), where 256 is `0x01 0x00`. The evolved mutator randomly byte-swaps 16- and 32-bit values with 50% probability, so interesting values are written in both endiannesses. This doubles the chance of producing a meaningful value in PNG header fields (chunk lengths, image dimensions, CRC values). The same endian awareness is applied to arithmetic mutations.
2. **Insert-or-overwrite for splicing and dictionary tokens.** The seed mutator always *overwrites* existing bytes when copying data from another corpus entry or inserting a dictionary token. This destroys the structure existed at the write position. The evolved mutator has a 50% chance of *inserting* instead, where it shifts existing bytes right via `memmove` to make room. Inserting preserves the bytes after the insertion point, so subsequent PNG chunks remain intact while new content is added before them.
3. **Increased dictionary probability.** The seed allocates 11.7% of its mutation budget to dictionary operations. The evolved mutator raises this to 15.6%. Dictionary tokens are byte sequences that AFL++ has already discovered to trigger new coverage during the campaign, making them the highest-value mutation material available. Increasing their usage probability means more mutations leverage known-useful byte patterns rather than random bytes.
4. **Unified dictionary selection.** The seed mutator checks auto-discovered tokens first and only falls back to user-provided entries if none exist. The evolved mutator samples from both sources simultaneously when both are available (70% auto-discovered, 30% user-provided), increasing the diversity of dictionary-based mutations.

To summarize, the LLM arrived at format-aware heuristics through evolutionary search using only the provided fitness signal, without any explicit knowledge of the PNG format.

5 Microbenchmarks

We now evaluate **EvolFuzz** on five MAGMA benchmark targets to answer the following research questions:

- **RQ1:** Does target-specialized evolution improve coverage over vanilla AFL++?
- **RQ2:** Does the coverage improvement translate to finding real bugs?
- **RQ3:** How does the evolutionary search behave across different targets?

For each target, we evolve a specialized C mutator (30 iterations, Gemini 3.1 Pro, MUTATOR_ONLY mode, 30s evaluation per candidate) and validate on MAGMA with 30-minute campaigns. We additionally present a case study on OpenSSL (Section 5.4).

5.1 Coverage Improvement

For each of the five targets, we compute a baseline (vanilla AFL++, 3 trials, median edge count) and evolve a specialized C mutator for 30 iterations. Each candidate is compiled into a shared library and evaluated by running AFL++ for 30 seconds with a deterministic seed. The coverage ratio (candidate edges / baseline edges) serves as the fitness score.

Figure 3 and Table 4 show that all five evolved mutators outperform vanilla AFL++. For three binary-format targets, libpng, libtiff, and libxml2, there is consistent 5–6% improvement. OpenSSL (DER/ASN.1) coverage improved significantly by 32%, whereas sqlite3, the only text-based target, only increased slightly by 1.4%.

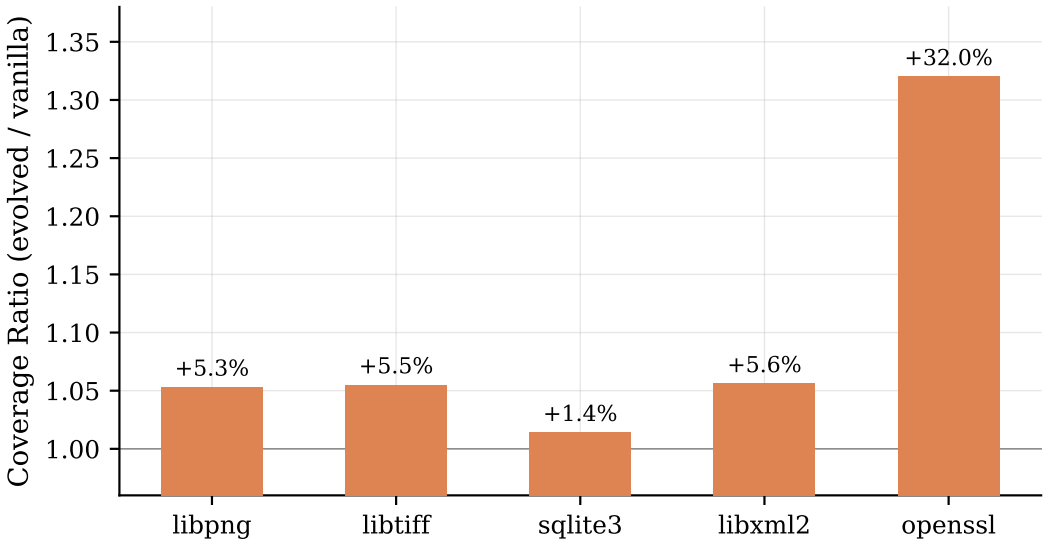


Figure 3 Coverage ratio (evolved / vanilla) across five MAGMA targets (30s, deterministic seed).

Structured binary formats benefit most from evolved mutators. Formats like PNG, TIFF, and DER/ASN.1 have strict byte-level structure (magic bytes, length fields, checksums) that penalizes random mutations, as most random byte flips produce immediately-rejected inputs. Evolved mutators learn to work *with* this structure (e.g., endian-aware values, structure-preserving insertion; see Section 4), therefore producing more inputs that pass initial validation and reach deeper code paths. In contrast, text-based formats like SQL are harder to improve with byte-level mutations, as meaningful changes require token-level understanding.

Insight. The largest gains occur on structured binary formats, where vanilla AFL++'s format-agnostic mutations are least effective. Evolved mutators discover format-aware heuristics through coverage feedback alone, without explicit format knowledge.

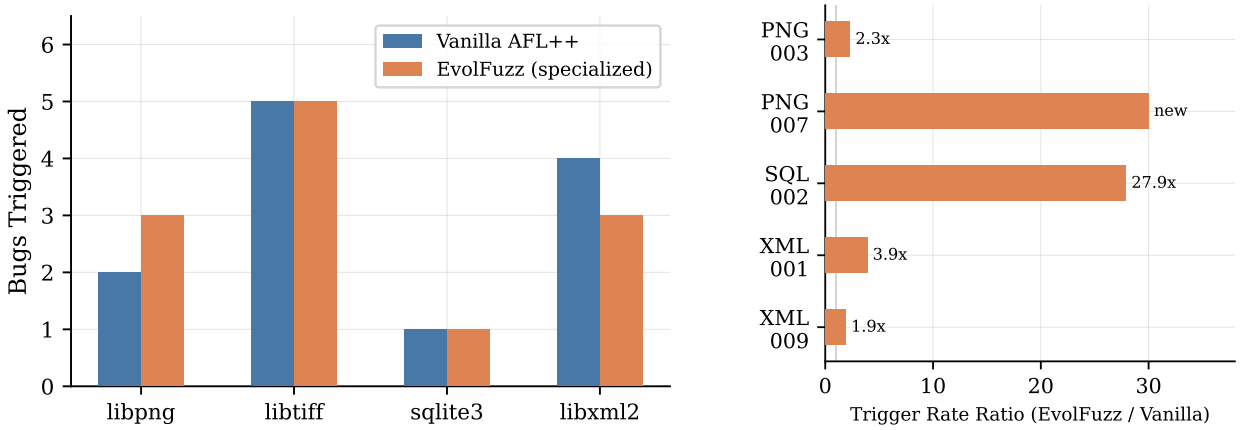
5.2 Bug-Finding on MAGMA

We run each evolved mutator against vanilla AFL++ on the corresponding MAGMA target for 30 minutes. MAGMA's canary instrumentation records how many times each bug is reached and triggered.

Figure 4a shows that coverage improvement does not uniformly translate to bug-finding. On libpng, **EvolFuzz** triggers 3/6 bugs vs. vanilla's 2/6 (+1), including PNG007 which vanilla reaches but never triggers. On libtiff and sqlite3, both fuzzers trigger the same bugs. On libxml2, vanilla triggers one more (4/8 vs. 3/8). On OpenSSL, neither fuzzer triggers any bugs on the x509 harness in 30 minutes (see Section 5.4).

Raw bug counts, however, tell an incomplete story. We therefore also measure the trigger rate (triggered / reached): the probability of triggering a bug given that the vulnerable code was reached. Table 5 shows that **EvolFuzz** achieves 2–28× higher trigger rates on shared bugs. For sqlite3/SQL002, despite fewer total reaches (4.1M vs. 5.4M), **EvolFuzz** triggers the bug 21× more often (84 vs. 4 triggers). Although our system is evolved on coverage rather than bug-triggering, we hypothesize that coverage optimization indirectly improves trigger rates: the heavier use of dictionary tokens may produce inputs that more closely resemble valid SQL, increasing the chance of satisfying specific bug conditions.

Insight. Even when bug counts are tied, evolved mutators produce higher-quality mutations that are 2–28× more likely to trigger a vulnerability per code reach.



(a) Number of bugs triggered.

(b) Trigger rate ratio per bug.

Figure 4 MAGMA bug-finding results. **(a)** EvolFuzz triggers one additional bug on libpng; other targets are tied or slightly worse. **(b)** On bugs where both trigger, EvolFuzz achieves 2–28× higher trigger rates.

Bug	Vanilla Rate	EvolFuzz Rate	Ratio
libpng / PNG003	1.2×10^{-1}	2.7×10^{-1}	2.3×
libpng / PNG007	0	9.4×10^{-6}	new
sqlite3 / SQL002	7.4×10^{-7}	2.1×10^{-5}	27.9×
libxml2 / XML001	3.4×10^{-2}	1.3×10^{-1}	3.9×
libxml2 / XML009	7.5×10^{-5}	1.5×10^{-4}	1.9×

Table 5 Trigger rate comparison on bugs triggered by at least one fuzzer. Full per-bug reach/trigger counts are in Appendix E.

5.3 Evolution Dynamics

We analyze the evolution trace for each target to understand search dynamics. Figure 5 reveals qualitatively different patterns. libpng is the easiest target: 83% of candidates beat baseline, with steady incremental progress. libtiff finds its best early (iteration 2) but has high variance, as 23% of candidates score below 0.95×. libxml2 shows gradual optimization with 7 successive improvements over 26 iterations.

OpenSSL has 25 iterations hover near 1.00×, then the LLM discovers a different strategy at iterations 26–27, increasing coverage to 1.32×. This suggests that for highly structured formats, most incremental modifications are ineffective, but the search space might contain rare high-value solutions that require sustained exploration.

Across all targets, Gemini 3.1 Pro achieves 100% compile rate and 29–83% of candidates beat the baseline depending on target difficulty (see Appendix G).

Insight. Key breakthroughs can happen late during the run. Thus, running evolution for longer iterations could lead to much better results than early stopping.

5.4 Case Study: OpenSSL

OpenSSL provides insight into the relationship between coverage and bug-finding. OpenSSL uses Distinguished Encoding Rules (DER), a strict binary tag-length-value format for encoding cryptographic structures. We evolved the mutator against the `x509` harness, which parses DER-encoded certificates, while MAGMA evaluates OpenSSL across 6 harnesses.[3]

[3]MAGMA's default configuration required adjustment for compatibility with our evolved mutator. See Appendix F for details.

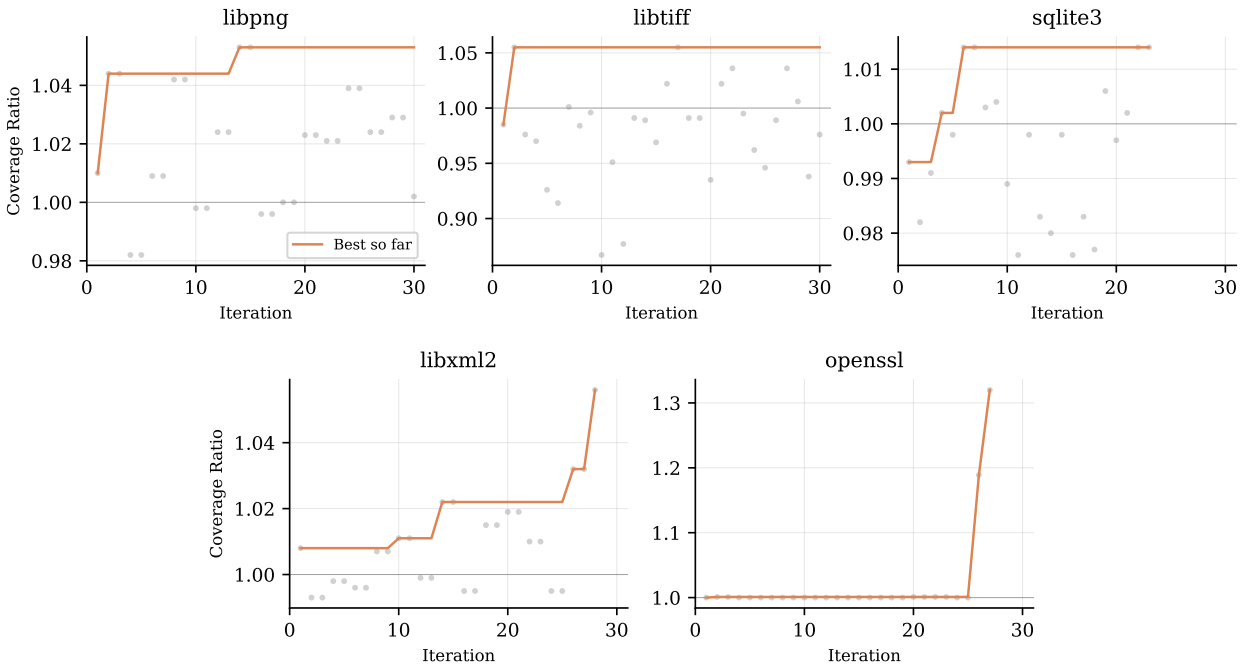


Figure 5 Evolution trajectories. Points show individual candidate scores; the line tracks the running best.

Across all 6 harnesses (30 min each), both fuzzers trigger the same 2 out of 10 reachable bugs: SSL002 (via client/server) and SSL003 (via asn1). However, the evolved mutator triggers them more reliably: SSL002 2.1× more often (1,477 vs. 696 triggers) and SSL003 1.2× more often (83,591 vs. 69,382). It also reaches SSL009 8.8× more intensively (34,642 vs. 3,919), suggesting it explores a qualitatively different region of the DER input space. Full per-harness data is in Appendix F.

Insight. The 32% improvement in coverage leads to higher trigger counts (2.1× on SSL002) and deeper exploration (8.8× on SSL009), but not additional unique bugs in 30 minutes. The evolved mutator may need longer campaigns to convert its deeper exploration into new bug triggers.

6 Discussion

6.1 Summary of Findings

Our evaluation demonstrates that LLM-guided evolution can produce AFL++ custom mutators that improve coverage (1.01–1.32×) and bug-finding (higher trigger rates, additional bugs on libpng). We highlight three key results.

First, target specialization is essential for evolving effective mutation strategies. Indeed, single-target evolution produces roughly double the coverage gains of multi-target evolution and reverses a negative bug-finding result into a positive one on libpng. The fitness signal is diluted when optimizing across targets with conflicting format requirements.

Second, structured binary formats benefit most. OpenSSL (DER/ASN.1) shows a 32% coverage improvement, an order of magnitude larger than the 1–6% gains on other targets. Formats with strict encoding rules create the largest opportunity for learned strategies, because vanilla AFL++'s format-agnostic mutations waste most attempts on structurally invalid inputs.

Third, coverage improvement does not reliably predict bug-finding. libxml2 shows 5.6% more coverage but one fewer triggered bug on MAGMA. However, trigger rate analysis reveals that evolved mutators

consistently produce higher-quality mutations (2–28× more likely to trigger a bug per code reach). We suspect longer fuzzing campaigns could lead to additional bug counts.

6.2 Limitations

Compute constraints. Fuzzing is inherently time-consuming, and compute was our primary limiting factor. Standard fuzzing evaluation [6, 9] requires multiple trials with statistical significance testing (e.g., Mann-Whitney U tests). Ideally, we would run (a) 100+ evolution iterations per target to explore the search space more thoroughly, (b) 10+ independent MAGMA trials per configuration for statistical significance testing, and (c) 24-hour campaigns matching the FuzzBench standard [8]. At our current single-machine throughput, this would require approximately 500 hours of compute (vs. the 20 hours our pipeline consumed). Unfortunately, this means our results have weaker statistical power, and some additional effects might be underexplored.

Single LLM and evolution framework. We evaluated only one LLM (Gemini 3.1 Pro) and one evolutionary framework (OpenEvolve with MAP-Elites). Our proposal identified several alternative synthesis approaches with different search paradigms: K-Search [2], which uses a co-evolving world model to plan high-level optimization intents before generating code; ShinkaEvolve [7], which adds novelty-based rejection sampling for sample efficiency; and GEPA [1], which uses natural language reflection instead of population-based evolution. Different LLMs may also produce qualitatively different mutation strategies.

Generalization. Each evolved mutator is specialized for one target. We did not evaluate whether mutators transfer across targets within the same format family (e.g., does a PNG mutator help on other image formats?) or across format families. The multi-target evolution results suggest that generalization is difficult, but a systematic study of transfer would be beneficial.

6.3 Future Work

Scaling evaluation via FuzzBench. An AlphaEvolve-style system, like ours, could benefit from decoupling evolution and validation. We envision a future system that evolves mutators locally and then submits the best candidates to Google's hosted FuzzBench service [8] for rigorous 24-hour, 20-trial evaluation across dozens of targets. This would provide statistically robust results without requiring us to maintain expensive compute infrastructure, and would make our results directly comparable to the fuzzing research community's standard benchmark.

Bug-triggered fitness. Our current fitness function optimizes coverage, which is an imperfect proxy for bug-finding. An alternative is to use MAGMA's canary feedback directly as the fitness signal. This will reward mutators that trigger bugs, not just reach code. The downside is this would require much longer per-iteration evaluations and could potentially lead to overfitting on existing bug benchmarks.

Per-operator attribution. In our analysis, we were able to show what the evolved mutator changed but not which of these changes contributed most to the improvement. Instrumenting the evolved mutator to log which operator produced each new edge would enable Driller-style [10] component attribution, providing a deeper understanding of why evolved strategies outperform vanilla.

7 Conclusion

We presented **EvolFuzz**, a system that uses LLM-guided evolutionary program synthesis to automatically discover custom mutation strategies for coverage-guided greybox fuzzing. By evolving C-language mutators specialized for individual targets, we achieve 1–32% coverage improvement over vanilla AFL++ across five MAGMA benchmark targets, with the largest gains on structured binary formats like DER/ASN.1. On MAGMA's ground-truth bug-finding benchmark, evolved mutators find an additional bug on libpng and achieve 2–28× higher trigger rates across targets, demonstrating that the improvement extends beyond coverage to mutation quality. Our results show that LLM-guided evolution is a promising direction for automated fuzzer design.

References

[1] Aviral Agrawal et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

[3] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of WOOT*, 2020.

[4] Google DeepMind. Alphaevolve: A coding agent for scientific and algorithmic discovery. Google DeepMind Blog, 2025.

[5] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[6] George Klees, Philipp Rümmer, Markus Rüthing, and László Szekeres. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138. ACM, 2018. doi: 10.1145/3243734.3243804.

[7] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv preprint arXiv:2509.19349.

[8] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1393–1403. ACM, 2021. doi: 10.1145/3468264.3473932.

[9] Moritz Schloegel, Marcel Böhme, et al. Sok: The state of fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024.

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

Appendix

A Setup Steps

Our artifact is a self-contained directory containing the complete **EvolFuzz** package, pre-built evolved mutators, and MAGMA integration files. The source code, evaluation infrastructure, and reproduction instructions are available at `https://github.com/evolfuzz/evolfuzz`.

Prerequisites. Use Linux (Ubuntu 22.04/24.04) with Docker installed and running. Ensure Python 3.10+ is available. You will also need a Gemini API key (`GEMINI_API_KEY`).

Step 1: Install dependencies.

```
cd artifact

# AFL++ v4.35c (must be this exact version)
git clone --branch v4.35c \
    https://github.com/AFLplusplus/AFLplusplus.git \
    third_party/AFLplusplus

# OpenEvolve (evolutionary framework)
git clone https://github.com/codelion/openevolve.git \
    third_party/openevolve
cd third_party/openevolve && pip install -e . && cd ../..

# Python dependencies
pip install -r evolve_fuzz/requirements.txt
```

Important: AFL++ must remain pinned to version `v4.35c`. Later versions cause segmentation faults with Python-based custom mutators.

Step 2: Build the Docker image (∼5 minutes).

```
python -m evolve_fuzz.run docker-build
```

This builds an Ubuntu 24.04-based image that compiles AFL++ with custom mutator support, builds all instrumented targets (libpng, libtiff, sqlite3, libxml2, openssl, zlib, libjpeg), and generates seed corpora.

Step 3: Configure API key and compute baselines.

```
export GEMINI_API_KEY="your-key-here"
python -m evolve_fuzz.run baseline \
    --duration 30 --trials 3 \
    --targets libpng libtiff sqlite3 libxml2 openssl
```

Baselines are stored in `evolve_fuzz/evaluation/baselines.json` and must be recomputed on new hardware.

Step 4: Run single-target evolution (∼45 min per target).

```
EVOLVE_AFL_SEED_START=0 \
EVOLVE_STAGE3_TARGETS=libpng \
python3 -m evolve_fuzz.run openevolve \
    --iterations 30 \
    --stage2-duration 30 --stage3-duration 30 \
    --stage3-trials 1 --stage3-targets libpng \
    --initial-program evolve_fuzz/mutator/initial_mutator.c \
    --evaluator evolve_fuzz/evaluation/evaluator_c.py \
    --config evolve_fuzz/methods/openevolve_c_config.yaml
```

Replace `libpng` with any target name (`libtiff`, `sqlite3`, `libxml2`, `openssl`). The best evolved mutator is saved to:

```
evolve_fuzz/mutator/openevolve_output_oe_<timestamp>/best/best_program.c
```

Using pre-built mutators. The `evolved_mutators/` directory contains the best mutators from our experiments, ready to use with any AFL++ installation:

```
gcc -shared -Wall -O3 -fPIC \
    -I /path/to/AFLplusplus/include \
    evolved_mutators/libpng_best.c -o mutator.so

AFL_CUSTOM_MUTATOR_LIBRARY=./mutator.so \
AFL_CUSTOM_MUTATOR_ONLY=1 \
    afl-fuzz -i seeds -o output -- ./target @@
```

Notes. Run single-target jobs sequentially to avoid CPU contention. Setting `EVOLVE_AFL_SEED_START=0` enables deterministic seeds for reproducible results. Scores >1.0 indicate improvement over vanilla AFL++.

B AFL++ Custom Mutator API

Evolved mutators implement three functions:

```
void *afl_custom_init(afl_state_t *afl, unsigned int seed);
size_t afl_custom_fuzz(void *data, uint8_t *buf, size_t buf_size,
                       uint8_t **out_buf, uint8_t *add_buf,
                       size_t add_buf_size, size_t max_size);
void afl_custom_deinit(void *data);
```

`afl_custom_fuzz` is called $\sim$5000 times/sec. It receives an input buffer (`buf`), a second corpus entry for splicing (`add_buf`), and a size limit (`max_size`). It must write the mutated input to `*out_buf` and return the new length. The `afl_state_t` pointer provides access to:

- `a_extras` / `a_extras_cnt`: auto-discovered dictionary tokens
- `extras` / `extras_cnt`: user-provided dictionary entries
- `queue_cur`, `queued_items`: corpus metadata
- `fsrv.total_execs`: total executions so far

Mutators are compiled with `gcc -shared -Wall -O3 -fPIC -I /AFLplusplus/include` and loaded via `AFL_CUSTOM_MUTATOR_LIBRARY`.

C OpenEvolve Configuration

- **LLM**: Gemini 3.1 Pro, temperature 0.9, top-p 0.95, max tokens 16384, timeout 300s
- **Population**: 80 programs, 5 islands, migration every 15 iterations (15% rate)
- **Selection**: 70% exploitation, 30% exploration
- **Feature dimensions**: complexity (code length/nesting) and diversity (behavioral distance)
- **Evolution mode**: diff-based (LLM modifies `EVOLVE-BLOCK` only), no full rewrites

D Evaluation Infrastructure

- **Hardware**: GCP c4d VM, 32 vCPUs (AMD EPYC 9B45), single-machine
- **AFL++ version**: v4.35c (pinned via git submodule)
- **Docker base**: Ubuntu 24.04, `afl-clang-fast` instrumentation
- **Harness interface**: `LLVMFuzzerTestOneInput` (standard FuzzBench/OSS-Fuzz)
- **Execution mode**: Fork-server with file-based input (not persistent mode)
- **MAGMA validation**: 30-minute campaigns, POLL=5s monitor snapshots

Our evaluation uses the same toolchain as FuzzBench (`afl-clang-fast` instrumentation, `LLVMFuzzerTestOneInput` harness interface) and harnesses sourced directly from FuzzBench/OSS-Fuzz. The key difference is execution mode: we use fork-server mode while FuzzBench uses persistent mode via `libAFLDriver.a`. This affects absolute throughput but not relative comparisons, since both candidate and baseline use the same mode.

E Detailed Reach and Trigger Counts

Target	Bug	V. Reached	V. Trig.	E. Reached	E. Trig.
libpng	PNG001	530,861	0	1,030,556	0
	PNG003	1,026,882	123,426	1,241,943	338,952
	PNG004	941,276	0	889,709	0
	PNG005	496,542	0	947,947	0
	PNG006	17,962,323	9,216	22,931,640	1,786
	PNG007	146,881	0	318,735	3
libtiff	TIF003	30,387,181	0	58,738,260	0
	TIF005	50,781	44,517	25,025	25,025
	TIF006	5,503	5,503	3,585	3,585
	TIF007	197,877	20,226	162,510	19,648
	TIF012	960,337	39	2,423,051	90
	TIF014	177,651	13	142,862	16
sqlite3	SQL002	5,395,725	4	4,056,977	84
	SQL007–019	reached only		reached only	
libxml2	XML001	46,719	1,589	22,296	2,950
	XML003	280,273	25,193	340,008	0
	XML009	388,999	29	525,260	76
	XML017	2,185,797	1,371,366	3,415,535	2,158,043

Table 6 Full MAGMA reach and trigger counts. V = Vanilla AFL++, E = EvolFuzz (target-specialized). All campaigns: 30 minutes, single trial.

F OpenSSL MAGMA Details

CmpLog incompatibility. MAGMA's default AFL++ configuration enables CmpLog, a dual-binary instrumentation mode that runs a second copy of the target to solve comparison checks. This mode was not used during our evolution. When the evolved mutator was loaded with CmpLog enabled, it segfaulted within seconds on all harnesses. After disabling CmpLog for both fuzzers, the evolved mutator completed full 30-minute campaigns without issues. This was a configuration mismatch, not a code defect.

Per-harness results. Table 7 shows per-harness bug triggering. Both fuzzers trigger the same bugs on the same harnesses: SSL003 via asn1, SSL002 via client and server.

Harness	Vanilla Triggered	EvolFuzz Triggered
asn1	SSL003	SSL003
asn1parse	—	—
bignum	—	—
client	SSL002	SSL002
server	SSL002	SSL002
x509	—	—

Table 7 OpenSSL per-harness bug triggering (both without CmpLog, 30 min).

Aggregate reach and trigger counts.

Bug	V. Reached	V. Trig.	E. Reached	E. Trig.
SSL001	4,703	0	4,767	0
SSL002	860,088	696	858,740	**1,477**
SSL003	57,149,246	69,382	34,130,717	**83,591**
SSL005	57,242	0	13,615	0
SSL008	6,918	0	8,929	0
SSL009	3,919	0	**34,642**	0
SSL010	8,904,703	0	3,591,840	0
SSL016	1,220,843	0	1,336,624	0
SSL019	1,878,077	0	1,880,691	0
SSL020	21,433	0	15,115	0

Table 8 OpenSSL aggregate MAGMA results (best across all 6 harnesses, both without CmpLog, 30 min). **EvolFuzz** triggers SSL002 2.1× more and SSL003 1.2× more. SSL009 reach is 8.8× higher.

G Search Efficiency

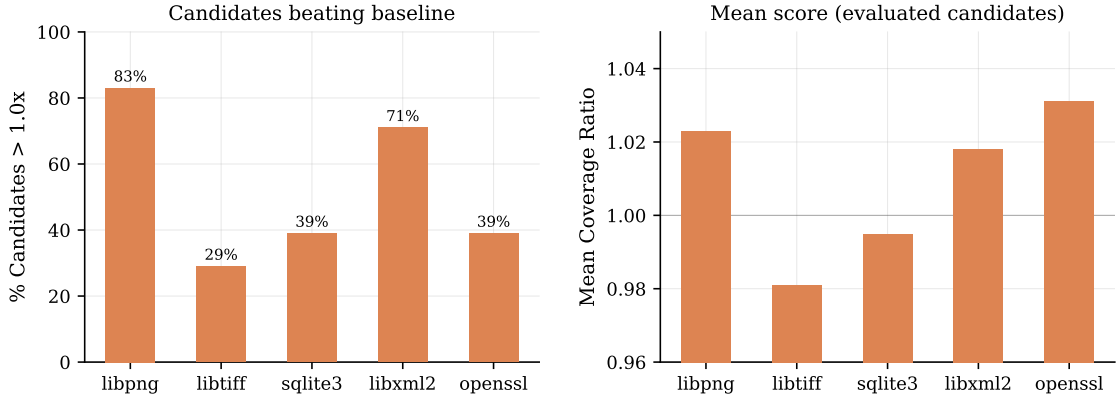


Figure 6 Left: percentage of LLM-generated candidates that beat vanilla baseline. Right: mean coverage ratio of all evaluated candidates. Gemini 3.1 Pro achieves 100% compile rate across all targets.

H AI Usage Statement

This project made use of a Large Language Model (LLM) to assist in several stages of development. Specifically, the LLM was used in the following ways:

1. **Fuzzer Design and Implementation**
   The LLM was consulted to help brainstorm and refine the overall implementation strategy for the fuzzer. This included possible architectures, mutation strategies, and approaches to input generation.
2. **Report Writing and Clarity Improvement**
   The LLM was used to improve the clarity, structure, and readability of the written report. It assisted in rephrasing sentences, organizing content, and ensuring the explanations were clear and concise.
3. **Environment Setup and Debugging**
   The LLM provided guidance for setting up the development environment, including Docker configuration and dependency management. It was also used to help diagnose and debug runtime errors encountered during implementation.

All final design decisions and evaluations were performed and verified by the project authors.