LLM-Assisted Seed Generation for Coverage-Guided Greybox Fuzzing of Programs with Structured Textual Languages

Hamidreza Maneshti
Department of Computer Science
Université du Québec à Montréal
Montreal, Canada
maneshti.hamidreza@courrier.uqam.ca

Alireza Manashty
Research & Development
DAITA Cloud Inc.
Regina, Canada
a.r.manashty@gmail.com

Quentin Stiévenart
Department of Computer Science
Université du Québec à Montréal
Montreal, Canada
stievenart.quentin@uqam.ca

***Abstract*—Coverage-guided greybox fuzzing depends strongly on the initial seed corpus. For programs that consume structured textual languages such as parsers, interpreters, and query engines, constructing useful seeds is difficult: minimal inputs provide little guidance, curated tests require project-specific effort, and grammar-based generation may satisfy syntax without reaching semantically interesting behavior. This paper studies whether a large language model (LLM) can serve as an offline source of seed-corpus diversity for such targets. We compare a minimal seed baseline (S0), a curated test-suite baseline (S1), and three LLM-assisted strategies that share one generation pipeline but differ in prompt context: test suite examples (S2), manual pages (S3), or formal grammars (S4). Our results point to a practical but targeted role for LLM-generated seeds. They provide reliable coverage gains over a minimal corpus, but curated test suites remain the strongest baseline overall. When test suites are unavailable, documentation- and grammar-prompted seeds can provide useful bootstrapping value, although their gains are target-dependent. Adding LLM seeds to an existing test suite yields selective improvements, and offline LLM seed generation is therefore best viewed as a low-overhead supplement to conventional corpus construction, not as a replacement for curated tests.**

***Index Terms*—fuzzing, software testing, large language models, seed generation, code coverage, AFL++, Honggfuzz, software security**

I. Introduction

Coverage-guided greybox fuzzing has become the dominant technique for discovering software vulnerabilities at scale. Production deployments such as Google's OSS-Fuzz have used fuzzers like AFL++ [1] and Honggfuzz [2] to discover thousands of vulnerabilities and bugs across many open-source projects since 2016 [3], [4]. The technique is now standard practice in security engineering. Its effectiveness, however, has strong ties with an artifact almost orthogonal to the fuzzer itself: the *initial seed corpus* [5]. Because mutations are local byte-level transformations, seeds that already exercise complex program features give the mutation engine a head start that random exploration may never achieve. A poor seed corpus simply caps how deep a campaign can go.

In this paper, we focus on initial seeds for programs that consume structured textual input languages, such as parsers, interpreters, query engines, and other tools whose inputs encode executable or analyzable behavior. Constructing a strong seed corpus for these programs is harder than it looks, and the challenges decompose naturally into three concrete questions that this paper addresses jointly. *(i) The seed-quality gap.* Traditional seed-construction techniques may miss coverage that manually crafted seeds would have reached: minimal inputs require many mutation rounds to reach deep states; test suites encode correctness rather than vulnerability, lack edge-case diversity, and require per-program extraction effort [6]; grammar-based generators enforce syntax but not semantics, so for example SQL queries against nonexistent tables are rejected before reaching the optimizer [7]. *(ii) Prompt-design uncertainty.* Recent work has begun to leverage large language models (LLMs) for fuzzing inputs [8]–[11], but the literature tends to fix one prompting strategy and one fuzzing engine, leaving practitioners without principled guidance on which contextual source (test cases, documentation, or formal grammar) is most effective for which target, or what validity and diversity profiles to expect. *(iii) Cost and integration unknowns.* If LLM-based generation requires per-target prompt engineering effort, fuzzer modifications, or substantial dollar cost, it is not a practical option for the open-source maintainers and security teams who actually run fuzzing at scale.

Structured textual languages are a natural meeting point between fuzzing and LLMs. Many important fuzzing targets such as parsers, interpreters, compilers, query engines, and configuration processors consume text whose syntax and idioms are documented in examples, manuals, grammars, and tests. Since modern LLMs are trained to generate code and structured text, they may be able to produce seed inputs that are more meaningful than random bytes while still requiring no target-specific generator.

We propose three LLM-assisted seed-generation strategies for offline corpus construction. The goal, as depicted in Figure 1, is not to put an LLM in the fuzzing loop, where inference latency would compete with fuzzer throughput, but to use the model before fuzzing begins to produce a fixed set of plain-text seed files. The three strategies share the same

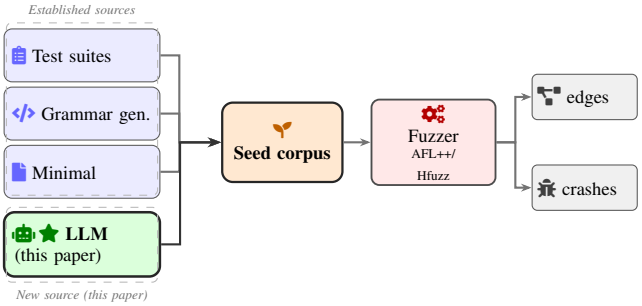

Fig. 1. Sources of initial seed corpora for coverage-guided fuzzing. This paper studies offline LLM-generated seeds as a fourth source alongside developer tests, grammar-based generation, and minimal fuzzer inputs.

generation pipeline and differ only in the contextual source supplied to the model: existing test files (strategy S2), the program's manual page (strategy S3), or a formal grammar (strategy S4). The resulting corpora can be passed directly to unmodified fuzzing campaigns.

Our goal is not to replace greybox fuzzing with an LLM-driven fuzzer. We use the LLM only as an offline corpus-construction tool and ask how much practical value this adds to fuzzing campaigns. The question is: when an LLM is used offline before fuzzing to prepare an initial corpus for an otherwise unmodified greybox fuzzer, which prompt sources provide useful coverage, how close do they come to developer test suites, and what is the practical cost of using them? Our results support a conservative conclusion: LLM-generated seeds are a useful bootstrap when no curated test suite is available, but curated test suites remain hard to beat.

This paper makes the following contributions:

- We design three offline LLM-assisted seed-generation strategies (test-suite-, manual-page-, and grammar-prompted) that share one generation pipeline and differ only in the contextual source supplied to the model, isolating the effect of prompt content on seed quality.
- We evaluate these strategies under a uniform fuzzing setup on four source-instrumented C targets and two production greybox fuzzers, with ten 12-hour runs per configuration. Statistical tests show that LLM+Tests (S2) significantly improves over the minimal baseline in all eight target-fuzzer combinations, but that curated test suites remain difficult to beat.
- We characterize validity, duplicate generation, initial corpus coverage, crash discovery, hybrid corpora, and API cost. The evidence positions offline LLM seed generation as a low-overhead supplement to corpus construction, not as a replacement for developer test suites.

The remainder of the paper is organized as follows. Section II reviews coverage-guided fuzzing, established seed-construction techniques, and existing LLM-for-fuzzing work. Section III presents the five seed strategies and the shared generation procedure. Section IV describes the experimental methodology, and Section V answers the four research questions (RQ1-RQ4) in turn. Section VI discusses threats to validity. Section VII concludes the paper.

II. Background and Related Work

Figure 2 situates the present paper in the literature. Three established research lines converge at the body of work on LLM-assisted seed generation: coverage-guided fuzzing, seed-corpus construction, and LLMs for software testing. The existing LLM-seed-generation studies tend to fix one prompt source and one fuzzing engine at a time. We extend this line of work along three axes simultaneously: three prompt sources held to a uniform generation procedure, two fuzzers across four targets, and hybrid corpora that combine the test suite with one or more LLM strategies. We also add a cost and integration analysis. The remainder of this section covers the main related work.

A. Coverage-guided greybox fuzzing

Coverage-guided greybox fuzzers use lightweight, compile-time instrumentation to track which control-flow edges have been executed and use that feedback to drive mutation toward unexplored regions [13], [23]–[25]. AFL [12] and its actively maintained fork AFL++ [1], established the dominant design: an evolutionary loop in which seeds drawn from a queue are mutated, executed under instrumentation, and retained whenever they trigger new edges. Honggfuzz [2] adopts a related design with multiple feedback mechanisms and multi-threaded workers. Edge coverage (the count of distinct basic-block transitions) has become the de facto measure of fuzzing progress because it captures control-flow decisions that block- or line-coverage miss [13], [26]. We perform our study using both AFL++ and Honggfuzz.

B. Seed corpus construction

Prior work shows that seed quality strongly affects fuzzing outcomes [5]. Established techniques span minimal seeds, test-suite extraction [6], grammar-based generation [7], and corpus distillation. Each has limitations: minimal seeds require many mutation rounds to reach deep states; test suites encode correctness rather than vulnerability and require per-program extraction effort [6]; grammars enforce syntax but not semantics [7]; distillation reduces an existing corpus but cannot introduce new behavior. Early learning-based approaches based on RNNs, GANs, or VAEs have faced training-data, validity, and generalization barriers [27], [28].

C. LLMs for fuzzing

LLMs pre-trained on code and documentation corpora can generate structured inputs that are syntactically valid and often semantically coherent. Most LLM-for-fuzzing work targets general-purpose input generation or keeps the model in the fuzzing loop. We focus instead on offline seed generation for programs that consume structured textual languages. This scope matches a core strength of LLMs: generating code-like and document-like text with recognizable syntax and idioms. It avoids a weaker setting for language models, namely inventing opaque binary blobs whose validity depends on layout constraints hard to express in natural-language context.

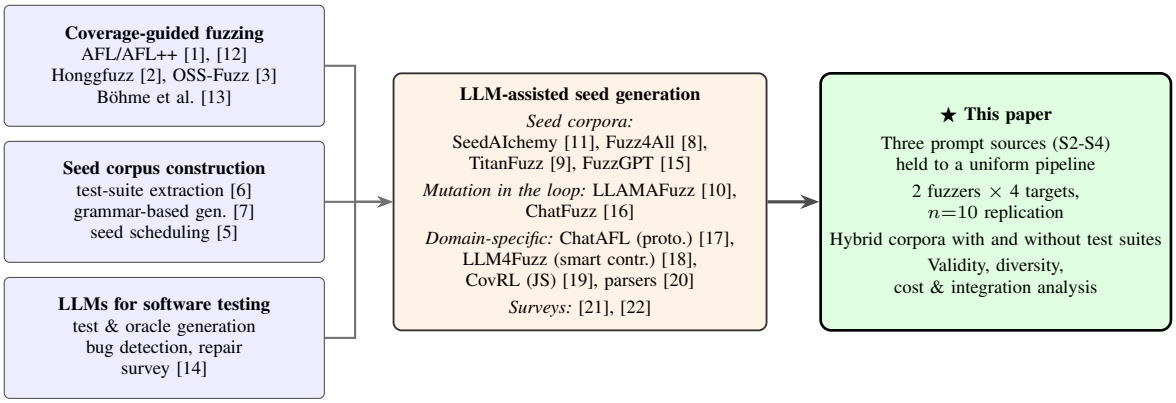

Fig. 2. Position of this paper in the LLM-fuzzing literature.

LLAMAFuzz [10] keeps the LLM in the mutation loop: it consults the model at each iteration to seed and guide mutations using coverage feedback. The LLM acts as a learned mutator that proposes targeted edits to the queue, in contrast to our approach, where the LLM is invoked only during offline corpus construction before fuzzing begins. Fuzz4All [8] fuzzes diverse language targets without per-target customization, but it keeps the LLM in the loop throughout the campaign, continuously generating and mutating inputs, whereas we use the LLM only to build a fixed seed corpus for an unmodified greybox fuzzer. SeedAIchemy [11] is the closest related work, but it addresses an orthogonal corpus-construction setting: it uses LLMs to guide retrieval and mining of existing public seed files, whereas we study direct offline generation of new seeds under controlled prompt sources.

TitanFuzz [9] and FuzzGPT [15] demonstrate zero-shot and edge-case generation. LLM-based fuzzing has also been applied to protocols [17], smart contracts [18], JavaScript engines [19], and parsers [20], with surveys synthesizing the area [21], [22]. Augmentation approaches such as ChatFuzz [16] integrate LLMs into existing pipelines for specific tasks rather than replacing the fuzzer. Despite these advances, systematic comparison of prompt sources, validation of generated seeds, and analysis of cost-benefit trade-offs across multiple fuzzers remain limited. We address those gaps.

III. LLM-Assisted Seed Generation

A. Design overview

The dominant question in seed corpus design is where the corpus' diversity comes from. Traditional answers locate diversity in human effort (developers writing tests), in syntactic enumeration (a grammar generator), or in fuzzer-internal mutation (a minimal seed). We propose to relocate it to a fourth source: the latent knowledge of a pre-trained LLM. Rather than embedding the LLM into the mutation loop, where its inference latency would compete directly with the fuzzer's throughput, we use the LLM only in an offline pre-processing phase before the campaign begins. The resulting corpus is plain text and feeds an unmodified fuzzer.

TABLE I
The five seed generation strategies.

Strategy	Knowledge source	LLM	Generation scope
S0 (Minimal)	–	no	1 minimal file
S1 (Test-suite)	Test cases	no	extracted
S2 (LLM+Tests)	Test cases	yes	per-file
S3 (LLM+Manual)	Manual pages	yes	batch
S4 (LLM+Grammar)	BNF/EBNF grammars	yes	single-case

Figure 3 shows the resulting design. The five strategies share the same fuzzers, campaign budget, instrumentation, and coverage/crash measurements. Strategy S0 contains a single minimal seed, S1 uses the available test-suite inputs limited at 26 unique tests, and S2-S4 target 26 unique LLM-generated seeds. Strategies S1-S4 differ only in the contextual source they consume and the path that source takes to the corpus. The two non-LLM strategies map a source to a corpus directly: the minimal baseline S0 corresponds to a single minimal file, and the test-suite baseline S1 to a deterministic extraction from the program's existing test repository. The three LLM-assisted strategies (S2, S3, S4) instead pass their source through a shared generation procedure that we describe in Section III-B. Holding corpus size, fuzzer, and measurements fixed makes it possible to attribute coverage differences cleanly to the contextual source rather than to corpus size, fuzzer choice, or campaign budget. Table I summarizes the five strategies.

We do not include a context-free LLM strategy: with only the target name, outputs for popular tools may reflect memorized examples, while outputs for obscure tools would lack enough input-language guidance. We focus on auditable sources practitioners can supply: tests, manuals, and grammars. We omit a pure grammar-generator baseline because it would answer a different question: whether grammar generators outperform LLM-based corpus construction. Our S4 condition instead isolates whether grammar text is useful as prompt context under the same LLM pipeline as S2 and S3.

B. Shared generation procedure

A naive use of an LLM as a seed generator (one prompt, one call, one corpus) would conflate two orthogonal design axes

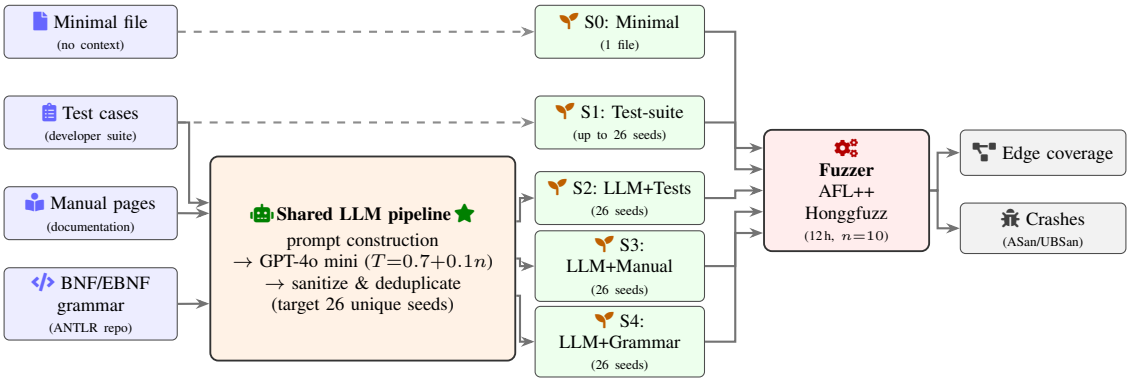

Fig. 3. System design of the proposed approach.

that this paper aims to disentangle: the contextual source given to the model and the engineering details of how the corpus is assembled. We therefore fix the corpus-assembly procedure and vary only the contextual source given to the model. The shared procedure used by all three LLM-assisted strategies (S2-S4) is the central pipeline depicted in Figure 3: a prompt is constructed from contextual material, the LLM is invoked, the output is sanitized to remove markdown fences, and uniqueness is checked against a persistent memory of previously accepted seeds. When the memory rejects a candidate, the next call uses a higher temperature set as $T = \min(0.7{+}0.1\,n, 1.0)$, where n is the retry index, together with an added instruction to generate seed content different from the previous attempts. The procedure repeats until the corpus contains 26 unique seeds.

Three details of this procedure are worth motivating explicitly. The fixed corpus size of 26 is dictated by the smallest available test suite (BusyBox bc has exactly 26 test cases): choosing this number lets BusyBox bc's S1 corpus be reported without artificial truncation and holds the three LLM strategies to the same target size. Corpus size is therefore controlled across S2-S4. S0 itself is intentionally minimal, and xmllint S1 contains only 23 usable standalone inputs after filtering. We account for this limitation when interpreting S0/S1 comparisons. We hold every LLM strategy to the same size, so that coverage differences cannot be attributed to corpus size. The temperature schedule starts at $T = 0.7$ to favor validity on the first attempt and rises only when uniqueness fails. This avoids the alternative of using high temperature uniformly, which we found in pilot runs to hurt syntactic validity. Each seed is attempted with up to six calls.

C. Strategy details

We now detail each of the five strategies.

a) S0: minimal seed: S0 consists of a single minimal file. It establishes a lower bound: the coverage achievable by the fuzzer's own mutation engine without any seed guidance.

b) S1: test-suite seeds: S1 consists of inputs extracted from each program's official test repository. For BusyBox bc, the test suite contains exactly 26 cases (all included). For lua and sqlite, 26 cases are randomly selected. The same 26 selected cases are used as in-context examples for S2, so any difference between S1 and S2 attributes to LLM-generated variation rather than to a different example set. xmllint is one exception: after removing duplicates, DTD fragments, and shell-driver wrappers from the libxml2 test suite, only 23 standalone, well-formed XML inputs remain. We report S1 over those 23 rather than padding the corpus with synthetic inputs that would distort the baseline.

c) S2: LLM + tests: For each test-suite file, the LLM is prompted with the file's content, an instruction to produce a unique test of the same structure, and extension-specific validity guidance. Generation is per file, and uniqueness is checked within the retry loop for that source file.

d) S3: LLM + manual page: The complete user-facing manual page for each target is embedded in a single prompt, and the LLM is asked for a batch of inputs that exercise diverse documented features. Output is split on double newlines and deduplicated.

e) S4: LLM + grammar: We embed verbatim in the prompt a formal grammar specification sourced from the ANTLR `grammars-v4` repository [29]. The LLM is instructed to produce a single concrete derivation per call, with an additional check ensuring the output does not trivially echo the grammar text. A global memory set prevents repeats across calls.

D. Prompt design and constraints

All prompts emphasize three properties: (i) *validity*, by asking for complete, syntactically valid inputs, or grammar-matching inputs for S4; (ii) *format hygiene*, prohibiting markdown, code fences, and explanatory text so that outputs can be written directly as seed files; and (iii) *diversity*, by requiring outputs to differ from previous seeds and, for S3, to exercise different documented features or edge cases.

E. No validity-based pre-filtering

We deliberately do *not* discard seeds that exit non-zero on a dry run before fuzzing. Both AFL++ and Honggfuzz perform their own corpus calibration on startup, deciding which seeds enter the queue based on coverage signal rather than on exit code. Pre-filtering would shadow that calibration and

could remove inputs the fuzzer would have used productively. Furthermore, real-world test suites deliberately contain error-triggering inputs (e.g., `SELECT`s against non-existent tables, expressions that exercise bc's error path), and removing these would amount to evaluating a degraded baseline. The validity rates reported in Section V-B therefore describe what each strategy *produces*, not a pre-filter applied to the experiments.

IV. Experimental Methodology

The goal of the evaluation is to attribute coverage and crash differences cleanly to seed-generation strategies, holding everything else fixed. This section justifies our choices for each axis that the evaluation touches: the targets, the fuzzers, the run design, and the budget.

A. Research Questions

Our research questions are the following.

- **RQ1.** *Do LLM-generated seeds improve final coverage and vulnerability discovery over a minimal baseline and over a curated test-suite baseline?* This asks whether LLMs bring any usable diversity to the corpus at all, and whether what they bring is competitive with a hand-curated test suite.
- **RQ2.** *How does the choice of prompt source affect seed validity and functional diversity?* This question asks whether different prompt sources produce seeds that are valid, distinct, and structurally diverse, since those are the properties a fuzzer's mutation engine can subsequently exploit.
- **RQ3.** *What does LLM-based seed generation cost in dollars, time, and integration effort?* Whether LLM seeds help is moot if the cost is comparable to the fuzzing budget itself. This question quantifies the overhead so that practitioners can decide whether to adopt the approach.
- **RQ4.** *Are LLM-generated seeds complementary to a curated test suite when combined into a hybrid corpus?* If LLM seeds alone do not beat the test suite, this does not mean that such seeds are not valuable. This question tests whether mixing LLM seeds with test suites produces a corpus that is stronger than either alone.

B. Targets

A meaningful comparison of seed-generation strategies needs targets that span distinct input formats but admit a uniform fuzzing setup. Our four targets satisfy four constraints simultaneously: they are widely-deployed open-source C programs (so the fuzzer toolchain applies), they accept structured input that can be described by a formal grammar (so the grammar-based strategy S4 has something to consume), they read input from a file or standard input (so the AFL++/Honggfuzz execution model applies), and together they span a wide range of input complexity (so cross-target patterns are observable). The four targets are **BusyBox bc** version 1.38.0 (a POSIX arbitrary-precision calculator with small input language and a well-defined grammar), **xmllint** from libxml2 version 2.15.3 (an XML parser/validator with namespaces, entities, DTD, and XPath, representing high structural complexity), **lua** version 5.4.8 (the Lua interpreter, with imperative and functional constructs and a substantial standard library), and **sqlite** version 3.53.1 (an in-process SQL database with lexer, parser, optimizer, and execution stages, run in in-memory mode for fuzzing). We deliberately did not include targets without a public test suite, since one of our experimental conditions (S1) requires one. We used the most recent version of each tool at the time of our experiments (May 2026).

C. Fuzzers

We use AFL++ and Honggfuzz because they are mature and widely used production fuzzers with different implementations, reducing the risk that our conclusions are specific to one fuzzing engine. For **AFL++** (version 4.40c) [1], targets were compiled with the fuzzer's `afl-clang-fast` compiler wrapper, with AddressSanitizer and UndefinedBehaviorSanitizer enabled. AFL++ was invoked with default mutators, no memory limit (`-m none`), a 1000 ms per-input timeout (`-t 1000`), and a 12-hour campaign budget. Each AFL++ campaign runs as a single `afl-fuzz` process consuming approximately one CPU core. For **Honggfuzz** (version 2.6) [2], targets were compiled with the fuzzer's `hfuzz-cc` compiler wrapper and SanitizerCoverage instrumentation, the same sanitizers, and the campaign duration set to 12 hours. Honggfuzz's default `--threads` setting was kept: this rule auto-detects the number of available logical CPUs and spawns one fuzzing worker thread per logical CPU to make use of the available cores (2 workers per campaign in our evaluation setup). The two engines consume different amounts of CPU per campaign by design, and their reported edge coverage measurement is influenced by fuzzer-specific details (e.g., how the instrumentation is performed). We therefore do not compare coverage across fuzzers and rely only on *within-fuzzer* comparisons.

D. Orchestrator and hardware

The experiments were run through a lightweight orchestration layer that prepares the instrumented target builds, launches fuzzing runs, and collects coverage and crash statistics. The coverage measurements are extracted from each fuzzer's runtime statistics (AFL++ from `plot_data`'s `edges_found`, and Honggfuzz from its statistics CSV).

The experiments were run on an HPC cluster, where each fuzzing run was allocated 2 CPU cores and 16GB of RAM on AMD EPYC CPU nodes, clocked between 2.40GHz and 2.65GHz. Runs were executed on node-local temporary storage. Each fuzzing campaign was run for 12 hours, repeated 10 times for statistical stability.

E. Run design

Coverage-guided fuzzing is inherently stochastic, so any single campaign is a noisy measurement [26], [30], [31]. We repeat every cell of the experimental matrix 10 times. The single-strategy matrix is therefore 4 targets $\times$ 2 fuzzers $\times$

5 strategies $\times$ 10 runs $=$ 400 12-hour campaigns. For RQ4, we evaluate hybrid corpora that pair the test suite with one or more LLM strategies, as well as one hybrid combining S3 and S4. That hybrid matrix adds a further 4 targets $\times$ 2 fuzzers $\times$ 5 hybrid strategies $\times$ 10 runs $=$ 400 campaigns. Each cell is reported as the arithmetic mean of its ten runs and the standard deviation is reported alongside as a descriptive indicator.

For the main statistical comparisons, we compare pairs of seed strategies within each target-fuzzer cell. For example, S1 vs. S2 on bc/AFL++ compares the ten final-coverage values from S1 against the ten final-coverage values from S2. Because runs are independent, we use two-sided, unpaired Mann–Whitney U tests [32]. We correct for multiple comparisons with Holm correction [33], applied separately within RQ1 and RQ4, using $\alpha = 0.05$. We also report Cliff's delta [34] as a non-parametric effect size. We call a result statistically supported only when the Holm-corrected p-value is below 0.05. Otherwise, we describe it as having "no clear difference," not as evidence of equivalence.

F. Coverage stabilization

We use the coverage growth to justify the 12-hour budget. Across all 400 completed non-hybrid single-strategy runs (used for RQ1-RQ3), 311 runs (77.8%) reach at least 80 % of their final coverage within the first hour. The growth is largely saturated by the final quarter of the campaign: 385 runs (96.3%) gain at most 5% additional final coverage between hours 9 and 12. The main exception is sqlite, whose configurations reach 80 % substantially later (median 104.6 minutes), especially under AFL++, indicating that this target benefits more from the longer campaign budget.

V. Empirical Results

We now present the results for each of our four research questions.

A. RQ1: Coverage and Vulnerability Discovery

1) Final edge coverage: Table II reports the final edge coverage of each (strategy, fuzzer, target) compared to the S0 baseline strategy.

The statistical tests confirm that S2 is the strongest LLM strategy, but not a replacement for S1. S1 obtains the highest final coverage in 7 of 8 target-fuzzer cells. Against S1, S2 is not clearly different after Holm correction in four cells, significantly worse in three cells (bc/Honggfuzz and Lua with both fuzzers, adjusted $p \leq 0.008$), and significantly better in one cell (xmllint/Honggfuzz, adjusted $p = 0.036$). Against the minimal baseline S0, S2 significantly improves coverage in all eight cells (all adjusted $p = 0.008$), S3 in seven (adjusted $p \leq 0.041$), and S4 in five (adjusted $p \leq 0.024$). Thus, LLM-generated seeds provide reliable improvement over a minimal corpus when prompted with test-suite examples, while documentation- and grammar-prompted seeds provide target-dependent bootstrapping benefits. The complete RQ1 statistics, including U values, adjusted p-values, and Cliff's deltas, are included in the replication package.

2) Improvement over the minimal baseline: Table II expresses every cell as relative improvement over S0. Descriptively, S1, S2, and S4 have positive mean deltas in all eight cells, while S3 is positive in seven and slightly negative on sqlite/AFL++. The formal tests show a similar but more conservative pattern: S2 significantly improves over S0 in all eight target-fuzzer cells (all adjusted $p = 0.008$), S3 in seven (adjusted $p \leq 0.041$), and S4 in five (adjusted $p \leq 0.024$). What separates the strategies is therefore not only whether they beat S0, but how much of S1's advantage they recover. S2 recovers most of it outside Lua, S3 recovers roughly 75-80% of S1's improvement on bc and xmllint but little on Lua and sqlite, and S4 recovers only a small fraction on most targets. The one consistent exception is Lua, where S1 leads the best LLM strategy by 26-31 pp and no LLM strategy comes close.

3) Bootstrapping without a test suite: A practical subquestion is what LLM seeds offer when no curated test suite exists, since this scenario rules out both S1 and S2 (the latter prompts the LLM with test-suite files, so it is unavailable when no test suite exists). The remaining bootstrapping options are therefore S3 (LLM + manual page) and S4 (LLM + grammar), which require only documentation or a formal grammar as input. Compared to the alternative of starting from a minimal seed, the better of S3 and S4 improves over the S0 mean in all eight (target, fuzzer) cells, by margins ranging from +0.7% (sqlite/AFL++) up to +76.7% (xmllint/Honggfuzz). How much of the test suite's advantage this recovers is strongly target-dependent: on bc and xmllint the better of S3/S4 recovers 75-82% of S1's improvement over S0, making it a genuinely useful bootstrap, whereas on lua and sqlite it recovers only 4-24%, leaving a large gap to S1. S3 or S4 are therefore a practical bootstrapping option for projects that lack an established test suite when the input language is comparatively simple (bc) or well captured by its documentation (xmllint), but offer little over a minimal corpus on targets such as sqlite, whose coverage depends heavily on error-path and feature-specific inputs that documentation- and grammar-prompted seeds do not reproduce.

4) Vulnerability discovery: Table III reports, for each cell, the number of crashing inputs that reproduce under AddressSanitizer/UndefinedBehaviorSanitizer, the number of *unique sanitizer signatures* they collapse to (distinct crash sites/types), as well as the number of *unique bugs*, since many distinct inputs frequently trigger the same underlying bug, sometimes with different sanitizer signatures. We distinguish sanitizer signatures using the sanitizer output, including crash type and reported source locations when available. We group unique bugs by identifying the root cause of the bug and fixing it to confirm that the crashing input does not fail after the fix. It should be noted that the number of unique bugs detected is low since we test the latest version of the tools, and these tools are common fuzzing targets. Crashes were observed only on BusyBox bc and sqlite while xmllint and lua produced none under any strategy, consistent with the maturity of those codebases. Two patterns stand out. First, crashing-input counts are a poor proxy for distinct bugs: on bc the strategies span

TABLE II

MEAN FINAL EDGE-COVERAGE IMPROVEMENT OVER S0 AFTER 12 HOURS. EACH CELL REPORTS $(\overline{C}_{Si} - \overline{C}_{S0})/\overline{C}_{S0}$, WHERE $\overline{C}_{Si}$ IS THE MEAN COVERAGE OF STRATEGY Si. THE PARENTHESES REPORT THE RUN-TO-RUN STANDARD DEVIATION IN PERCENTAGE POINTS AFTER NORMALIZATION. COLORS MARK BEST, SECOND-BEST, AND WORST VALUES IN EACH COLUMN. THE S0 ROW IS 0 BY DEFINITION AND IS SHOWN AS THE REFERENCE BASELINE.

	BusyBox bc		**xmllint**		**lua**		**sqlite**	
Strategy	AFL++	Hfuzz	AFL++	Hfuzz	AFL++	Hfuzz	AFL++	Hfuzz
S0 (Minimal)	0.0% (±5.9 pp)	0.0% (±1.2 pp)	0.0% (±2.1 pp)	0.0% (±2.7 pp)	0.0% (±1.5 pp)	0.0% (±1.2 pp)	0.0% (±2.1 pp)	0.0% (±4.1 pp)
S1 (Test-suite)	+31.5% (±0.9 pp)	+25.4% (±0.3 pp)	+90.2% (±0.5 pp)	+96.3% (±1.0 pp)	+82.7% (±1.0 pp)	+73.2% (±3.3 pp)	+17.9% (±3.0 pp)	+26.9% (±2.0 pp)
S2 (LLM+Tests)	+30.8% (±0.7 pp)	+24.2% (±0.3 pp)	+89.7% (±0.8 pp)	+98.5% (±1.6 pp)	+56.8% (±2.1 pp)	+42.0% (±4.8 pp)	+16.6% (±2.7 pp)	+26.4% (±2.0 pp)
S3 (LLM+Manual)	+25.7% (±1.7 pp)	+20.9% (±1.1 pp)	+67.8% (±0.6 pp)	+76.7% (±1.6 pp)	+19.8% (±2.0 pp)	+6.1% (±2.0 pp)	-0.2% (±4.3 pp)	+5.1% (±2.6 pp)
S4 (LLM+Grammar)	+7.5% (±3.1 pp)	+5.7% (±1.5 pp)	+3.4% (±2.1 pp)	+7.3% (±6.4 pp)	+13.8% (±1.2 pp)	+3.9% (±1.7 pp)	+0.7% (±8.8 pp)	+3.2% (±3.0 pp)

TABLE III

CRASHING INPUTS, UNIQUE CRASH SIGNATURES, AND MANUALLY IDENTIFIED SOURCE-LEVEL BUGS FROM THE NON-HYBRID 12-HOUR CAMPAIGNS. EACH CELL REPORTS *replayed crashing inputs / unique sanitizer signatures / unique bugs*.

	bc		**sqlite**	
Strategy	AFL++	Hfuzz	AFL++	Hfuzz
S0 (Minimal)	329 / 13 / 2	0 / 0 / 0	83 / 5 / 3	35 / 12 / 7
S1 (Test-suite)	32 / 2 / 2	9 / 2 / 2	62 / 5 / 3	19 / 8 / 6
S2 (LLM+Tests)	140 / 2 / 2	16 / 2 / 2	27 / 5 / 3	21 / 6 / 5
S3 (LLM+Manual)	200 / 8 / 2	5 / 3 / 3	85 / 5 / 3	29 / 6 / 5
S4 (LLM+Grammar)	11 / 3 / 3	1 / 1 / 1	39 / 5 / 3	17 / 6 / 5

up to 329 crashing inputs but collapse to at most 3 unique bugs, and on sqlite all five strategies converge to at most 7 unique bugs despite input counts spanning up to 85. Second, no seeding strategy holds a consistent crash-discovery advantage, and the minimal baseline is surprisingly competitive: S0 yields the most distinct signatures on bc/AFL++ (13) and on sqlite/Honggfuzz (12), indicating that the crashes reachable in these targets sit on shallow paths the fuzzer finds even from a minimal seed. The only cases where a seeded strategy surfaces more unique bugs than S1 are S4 (LLM+Grammar) on bc/AFL++ and S3 (LLM+Manual) on bc/Honggfuzz, each with 3 bugs against S1's 2. Since the number of unique bugs found is low, we do not claim a vulnerability-discovery advantage for LLM seeds. We are preparing upstream reports with minimized inputs and root-cause analyses.

Answer to RQ1

LLM-generated seeds improve final coverage over a minimal baseline, but do not replace curated test suites when they are available. S2 (LLM+Tests) significantly improves over S0 in all eight target-fuzzer cells (all adjusted $p = 0.008$), while S3 and S4 do so in seven (adjusted $p \leq 0.041$) and five cells (adjusted $p \leq 0.024$), respectively. Against S1, S2 is not clearly different after Holm correction in four cells, significantly worse in three, and significantly better in one. The largest gap remains Lua, where the developer test suite exercises behavior that generated scripts do not reproduce. Without a test suite, S3 and S4 are useful bootstrapping options on some targets, especially bc and xmllint, but their gains are target-dependent. For vulnerability discovery, unique sanitizer-signature and bug counts are small and largely strategy-independent.

TABLE IV

DIRECT VALIDITY RATES: CLEAN-PARSE SEEDS OVER TOTAL SEEDS.

Strategy	**bc**	**xmllint**	**lua**	**sqlite**
S1 (Test-suite)	18/26	23/23	25/26	12/26
S2 (LLM+Tests)	10/26	17/26	26/26	13/26
S3 (LLM+Manual)	22/26	24/26	26/26	26/26
S4 (LLM+Grammar)	15/26	26/26	26/26	26/26

B. RQ2: Validity and Diversity

1) Direct validity: We re-run each generated seed once through the corresponding target binary with a 5-second timeout and record the exit code. A seed is *valid* iff it parses cleanly (exit 0). The dry run does not invoke either fuzzer. Table IV reports the results. The LLM strategies that use richer context (S3 and S4) reach 100 % validity on three of four targets and meet or exceed the test-suite baseline. S2 is the weakest on bc (10/26) and sqlite (13/26), suggesting that in-context test-suite examples can lead the LLM to over-generalize into syntactically incorrect territory.

The test-suite baseline is itself *not* at 100% validity (12/26 on sqlite, 18/26 on bc): real test suites deliberately include error-triggering inputs (`SELECT`s against non-existent tables, expressions that exercise bc's error path) to exercise parser error handling. The validity gap between S1 and the LLM strategies on these targets therefore reflects a difference of *intent*, not LLM superiority: the developers' test suite tests both happy paths and error paths, while default LLM prompts produce happy-path inputs only. This partly explains S1's coverage advantage on sqlite, where parser error paths are a substantial fraction of the program logic.

2) Duplicate rejection: We report the fraction of LLM calls whose output was rejected as a duplicate during generation. S2 achieved zero duplicates on all four targets (per-seed memory of growing examples steers the model toward unique outputs). S3 produced two duplicates in total, both on lua. S4 was the most repetitive: a 13% duplicate rate on bc and 16% on lua, while xmllint and sqlite reached zero duplicates. This suggests that the constrained output space of a grammar specification yields more repetition on the simpler input languages, even

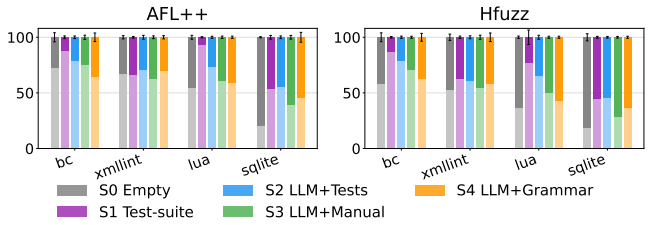

Fig. 4. Edge coverage (in percentage of final coverage) of each seed corpus before mutation (bottom part of the bars) and after the full 12h campaign (entire bar).

though previously generated seeds are supplied to the model as negative examples. Every strategy ultimately produced the target 26 unique seeds per target.

3) Initial seed coverage as functional diversity: String-level uniqueness does not capture whether seeds exercise *different parts* of the program. We use the initial edge coverage, i.e., the count of edges covered by the seed corpus before any mutation, as a measure of functional diversity. Figure 4 reports the result. S2 achieves the highest initial coverage on three of four targets (bc, xmllint, sqlite), exceeding S1 in some cases. S1 dominates on lua, again reflecting the comprehensiveness of the Lua test suite. S4 has the lowest initial coverage among the non-minimal strategies on three of four targets, consistent with its higher duplicate rate. On sqlite/AFL++, S2 starts from the highest initial coverage (8,911 edges, marginally above S1's 8,844). At the twelve-hour budget this early diversity is largely realized: S2 finishes as the second-highest strategy in final coverage on that cell, within run-to-run noise of S1 (Table II). More generally, on the three targets where S2 starts broadest (bc, xmllint, and sqlite under AFL++) it also finishes second only to S1, so initial breadth is a reasonable predictor of final coverage, yet S1 retains a small, consistent edge despite a comparable or narrower start, pointing to seed *mutability* (how readily small byte-level changes cross new branch boundaries) as a second factor beyond raw initial breadth.

Answer to RQ2

Different prompting strategies produce seeds with measurably different validity and diversity profiles. Context-rich strategies (S3, S4) reach 100% validity on three of four targets and meet or exceed the test-suite baseline. S2 overgeneralizes from in-context examples on bc and sqlite. S2 often provides the broadest initial coverage and has zero duplicates. S4 is the most repetitive on simple targets. Crucially, validity alone does not determine effectiveness (S2 has low validity on bc and sqlite yet the highest initial coverage), and at the twelve-hour budget initial diversity broadly tracks final coverage, though S1's small, consistent final-coverage edge over the broader-starting S2 shows that seed *mutability* still matters.

TABLE V
SEED GENERATION COST. API IS NUMBER OF LLM API CALLS, AND DUP. IS THE NUMBER OF DUPLICATES REJECTED BY DEDUPLICATION.

Strategy	Target	Seeds	API	Dup.	In tok	Out tok	Cost	Time
S2	bc	26	26	0	21,665	12,600	$0.011	4.3 min
	xmllint	26	26	0	39,039	55,059	$0.039	17.6 min
	lua	26	26	0	131,292	60,182	$0.056	17.9 min
	sqlite	26	26	0	5,729	3,416	$0.003	1.2 min
S3	bc	26	3	0	20,649	727	$0.004	0.7 min
	xmllint	26	3	0	5,937	1,728	$0.002	0.6 min
	lua	26	2	2	24,646	829	$0.004	0.6 min
	sqlite	26	3	0	136,254	577	$0.021	0.3 min
S4	bc	26	30	4	46,403	1,204	$0.008	0.6 min
	xmllint	26	26	0	53,246	3,078	$0.010	1.5 min
	lua	26	31	5	76,969	2,394	$0.013	0.8 min
	sqlite	26	26	0	214,445	1,491	$0.033	1.9 min

C. RQ3: Cost and Integration

We instrumented the generation procedure to record API calls, token consumption, duplicate rejections, wall-clock time, and dollar cost per pair of strategy and target. All runs used GPT-4o mini ($0.15 per 1M input tokens, $0.60 per 1M output tokens). Table V reports per-target results. From this data, we can make the following observations.

S3 is the most cost-efficient strategy: it needs only 2-3 API calls per target because it generates seeds in batches, totaling $0.030 and less than one minute per target. S2 yields zero duplicates but is slowest because it generates one seed per call with large in-context test files, taking up to 17.9 min on lua and costing $0.108 overall. S4 has moderate cost: it needs 26-31 calls per target, with at most five duplicate rejections, and sqlite dominates its token use because the grammar is large. Across all LLM strategies and targets, generation costs about $0.20 and 48 minutes. This excludes human time to locate/clean contextual material and build harnesses, so RQ3 shows that marginal generation cost is low once the setup exists.

Finally, integration is light: the generator is around 300 lines of Python and generated seeds are plain text files dropped into the fuzzer's input directory. No fuzzer modification is required. The main effort is one-time prompt engineering per target (selecting an appropriate test file or feeding the manual page or grammar).

Answer to RQ3

The measured API cost and generation time are small relative to a 12-hour fuzzing campaign: approximately $0.20 and 48 minutes for the full configuration set. S3 is the cheapest and fastest strategy. S2 is slower but provides the strongest LLM coverage results. Integration requires no fuzzer modification because the generated seeds are plain text files. These measurements support using LLM-generated seeds as a low-overhead supplement when suitable contextual material and harnesses are already available.

D. RQ4: Hybrid Corpora

RQ4 examines whether LLM-generated seeds can be *complementary*, either to a curated test suite when one is present, or together in the absence of a test suite. Because the cost of running an additional LLM-assisted strategy on top of a test suite is low (RQ3), even a modest complementary effect would make LLM seeds a useful optional supplement in practice. We evaluate multiple hybrid corpora: S1+Si for each LLM strategy, S1+S2+S3+S4 for evaluating the complementarity of all of our strategies together with S1, and S3+S4 for identifying the potential of combining the strategies that do not rely on an existing test suite. The hybrid configurations are run in the same setup as for RQ1: 10 runs of 12 hours. Table VI reports on the resulting coverage improvements. Based on these results, we can make the following observations.

1) Hybridization produces only small changes relative to S1: Across the four S1-based hybrids and eight cells, the change relative to S1 ranges from -1.0% (S1+S2 on sqlite/AFL++) to +4.6% (S1+S2+S3+S4 on xmllint/Honggfuzz). The statistical tests show that significant gains are concentrated on bc and xmllint: S1+S3 and S1+S2+S3+S4 significantly improve over S1 for both fuzzers on those two targets. The adjusted p-values are small in all eight supported cases (adjusted $p = 0.027$ and 0.017 for bc/AFL++ with S1+S3 and S1+S2+S3+S4, respectively, and adjusted $p \leq 0.007$ for the other six), with large effects ($|d| \geq 0.89$). S1+S2 and S1+S4 show no clear improvement over S1 in any target-fuzzer cell after Holm correction, and no hybrid involving S1 significantly degrades coverage relative to S1. The complete RQ4 statistics are included in the replication package.

2) The most consistent complements are S1+S3 and the full stack: S1+S3 and S1+S2+S3+S4 are the only hybrids involving S1 with significant improvements over S1 after correction, and those improvements occur on bc and xmllint for both fuzzers. The result suggests targeted complementarity rather than a general benefit from adding LLM seeds to an already strong test suite.

3) No substantial queue dilution is visible: Descriptively, S1+S2+S3+S4 (104 seeds) matches or exceeds S1+S2 (52 seeds) in 7 of 8 cells. This is not a size-controlled ablation: it shows the practical behavior of the larger hybrid under our twelve-hour budget, not a general proof that larger corpora are harmless. In a pilot variant, we minimized the input corpus with `afl-cmin` before fuzzing and did not observe material changes in the outcome.

4) Combining the test-suite-free strategies adds little: In the absence of a test suite, S3+S4 performs no better than the stronger of S3 or S4 alone in seven of eight cells (the deltas against the best single S3/S4 mean are within noise), the exception is sqlite/AFL++ (+6.7 pp). Running both the documentation- and grammar-prompted generators rather than the better one is therefore rarely worth the extra generation cost.

The practical takeaway is that adding LLM-generated seeds to a curated test suite usually leaves coverage close to S1, with selective gains on targets where the added seeds expose behavior not already covered by the test suite. The evidence supports targeted supplementation rather than routine improvement over a strong curated corpus.

Answer to RQ4

LLM-generated seeds are selectively complementary to a curated test suite. S1+S3 and S1+S2+S3+S4 significantly improve over S1 on bc and xmllint for both fuzzers (adjusted $p \leq 0.027$), while S1+S2 and S1+S4 do not significantly improve over S1 in any cell after Holm correction. No hybrid involving S1 significantly degrades coverage. Without a test suite, S3+S4 does not significantly beat the better of S3 or S4 alone in any target-fuzzer cell. Hybrid corpora therefore provide targeted rather than universal gains.

VI. Threats to Validity

A. Run-to-run variance

Every cell of our results matrices is reported as the mean of ten independent runs over a twelve-hour budget. Coverage-guided fuzzing is inherently stochastic, so even repeated runs of one configuration yield somewhat different final coverage. We therefore report run-to-run standard deviations as descriptive context and use unpaired Mann-Whitney U tests with Holm correction for the RQ1 and RQ4 comparisons. The standard deviations show the practical scale of the variation between runs, while the corrected tests determine which differences we treat as statistically supported. Since each cell has ten runs, small effects may remain undetected after correction. We therefore treat non-significant small deltas as inconclusive, not as evidence that two strategies are equivalent.

B. Concurrent campaigns and cross-fuzzer thread asymmetry

The orchestrator scheduled the campaigns concurrently on nodes of an HPC cluster, with two cores available per node. AFL++ runs as a single `afl-fuzz` process (one core) while Honggfuzz uses default multi-threaded workers (1 thread per core). Cross-fuzzer numerical comparisons therefore reflect both seed quality and engine-internal CPU asymmetry, and we do not consider them in our evaluation. Within-fuzzer rankings may be affected by node-to-node performance variation and queuing effects on the cluster. We mitigate this by using the same orchestration policy for every strategy and by repeating each 12h campaign ten times. We treat the low observed variance as supporting evidence for stability.

C. Sensitivity to model and prompt

Results characterize a single model (GPT-4o mini) with a fixed prompt template per strategy. Larger models (GPT-4o, Claude), open-source models (Llama, DeepSeek-Coder), alternative prompting techniques (chain-of-thought, few-shot, coverage-feedback loops), and other temperature regimes were not explored. Because each strategy is evaluated with one fixed prompt template and one fixed model, we cannot claim that the observed rankings are invariant across models or prompt

TABLE VI
HYBRID FINAL EDGE-COVERAGE DELTAS AFTER 12 HOURS. PARENTHESES REPORT NORMALIZED RUN-TO-RUN STANDARD DEVIATION IN PERCENTAGE POINTS. COLORS RANK ONLY THE S0-COMPARISON ROWS.

		BusyBox bc		xmllint		lua		sqlite	
Corpus	**Baseline**	AFL++	Hfuzz	AFL++	Hfuzz	AFL++	Hfuzz	AFL++	Hfuzz
S1+S2	vs S0	+31.2% (±0.3 pp)	+25.9% (±0.7 pp)	+91.1% (±0.6 pp)	+99.3% (±2.6 pp)	+85.0% (±3.0 pp)	+74.1% (±1.4 pp)	+16.8% (±3.1 pp)	+27.4% (±1.9 pp)
	vs S1	-0.3% (±0.2 pp)	+0.3% (±0.6 pp)	+0.5% (±0.3 pp)	+1.5% (±1.3 pp)	+1.2% (±1.7 pp)	+0.5% (±0.8 pp)	-1.0% (±2.6 pp)	+0.4% (±1.5 pp)
S1+S3	vs S0	+33.4% (±0.5 pp)	+27.2% (±0.3 pp)	+96.5% (±0.6 pp)	+102.4% (±1.3 pp)	+83.9% (±1.9 pp)	+74.0% (±1.1 pp)	+17.3% (±3.2 pp)	+29.1% (±1.5 pp)
	vs S1	+1.4% (±0.3 pp)	+1.4% (±0.2 pp)	+3.3% (±0.3 pp)	+3.1% (±0.7 pp)	+0.6% (±1.1 pp)	+0.5% (±0.6 pp)	-0.6% (±2.7 pp)	+1.7% (±1.2 pp)
S1+S4	vs S0	+31.5% (±0.6 pp)	+25.3% (±0.5 pp)	+89.9% (±0.4 pp)	+96.6% (±1.6 pp)	+84.0% (±2.9 pp)	+72.0% (±3.9 pp)	+19.3% (±4.1 pp)	+29.2% (±2.2 pp)
	vs S1	0.0% (±0.4 pp)	-0.1% (±0.4 pp)	-0.2% (±0.2 pp)	+0.1% (±0.8 pp)	+0.7% (±1.6 pp)	-0.7% (±2.2 pp)	+1.1% (±3.5 pp)	+1.8% (±1.7 pp)
S1+S2+S3+S4	vs S0	+33.6% (±0.4 pp)	+27.2% (±0.3 pp)	+97.2% (±0.9 pp)	+105.3% (±1.5 pp)	+85.2% (±2.9 pp)	+73.2% (±1.3 pp)	+18.9% (±2.5 pp)	+28.0% (±3.7 pp)
	vs S1	+1.5% (±0.3 pp)	+1.4% (±0.3 pp)	+3.7% (±0.5 pp)	+4.6% (±0.8 pp)	+1.3% (±1.6 pp)	0.0% (±0.8 pp)	+0.8% (±2.2 pp)	+0.9% (±2.9 pp)
S3+S4	vs S0	+24.9% (±1.2 pp)	+20.9% (±1.0 pp)	+67.8% (±0.5 pp)	+77.7% (±2.2 pp)	+21.5% (±1.2 pp)	+6.1% (±1.2 pp)	+7.5% (±1.8 pp)	+7.4% (±3.9 pp)
	vs best S3/S4	-0.6% (±0.9 pp)	0.0% (±0.8 pp)	0.0% (±0.3 pp)	+0.5% (±1.2 pp)	+1.4% (±1.0 pp)	0.0% (±1.1 pp)	+6.7% (±1.8 pp)	+2.2% (±3.7 pp)

designs. The comparison nevertheless isolates the practical behavior of three readily available contextual sources under a shared generation procedure, which is the deployment setting we target.

D. LLM training-data contamination

The four targets are widely deployed open-source projects whose source, official test suites, and manual pages were almost certainly part of GPT-4o mini's training corpus. Our prompts steer the model toward the explicit context (the supplied test file, manual page, or grammar), but we cannot guarantee that pre-training memory does not contribute. This threat does not invalidate the comparison among strategies, since all LLM-assisted strategies use the same model and are evaluated under the same generation procedure, but it limits claims about how much of the generated seed quality comes from prompt context rather than pre-training exposure.

E. External validity

Four C programs that process structured textual inputs do not exhaust the fuzzing landscape. Binary formats, network protocols, programs in managed languages, and projects without test suites remain open extensions. We focused on programs whose inputs are textual and grammar-like because this setting is especially plausible for LLM generation. The fixed target size of 26 seeds controls S2-S4 but not all comparisons: S0 is intentionally minimal, xmllint S1 contains 23 usable inputs, and hybrid corpora contain 52 to 104 seeds. The hybrid results therefore show the practical effect of adding seeds under a twelve-hour budget, not a size-controlled proof of complementarity. A controlled corpus-size ablation remains future work.

VII. CONCLUSION AND FUTURE WORK

We evaluated three LLM-assisted seed-generation strategies, based on test-suite examples (S2), manual pages (S3), and formal grammars (S4), against a minimal baseline (S0) and a test-suite baseline (S1) on four C programs that consume structured textual inputs and two established greybox fuzzers. Across ten 12-hour runs per configuration, the strongest LLM-generated corpus improves over S0 in every case, showing that LLM seeds can provide useful initial diversity when starting from a minimal corpus.

However, LLM-generated seeds do not replace curated test suites. S1 obtains the highest final coverage in 7 of 8 target-fuzzer cells. Formal statistical tests show that S2 is not clearly different from S1 in four cells, significantly worse in three, and significantly better in only one. The largest deficits occur on Lua, where the developer test suite exercises language-internal behavior that generated scripts do not reproduce. When no test suite is available, S3 and S4 are the relevant strategies: S3 significantly improves over S0 in seven cells and S4 in five, but their gains are target-dependent and recover much less of S1's advantage on Lua and sqlite. Crash-signature and unique-bug counts are low and do not show a consistent vulnerability-discovery advantage for LLM-generated seeds.

Hybrid corpora provide targeted rather than universal gains over S1. S1+S3 and S1+S2+S3+S4 significantly improve over S1 on bc and xmllint for both fuzzers. No hybrid involving S1 significantly degrades coverage. In the absence of a test suite, S3+S4 does not significantly beat the better of S3 or S4 alone in any cell. The measured API cost of seed generation is low: about $0.20 across all LLM configurations in this study, and the resulting corpora require no fuzzer modification. Overall, offline LLM-assisted seed generation is best viewed as a practical low-cost supplement to conventional corpus construction, particularly when curated tests are unavailable, rather than as a replacement for strong developer test suites.

To support reproducibility, we provide a replication package containing the prompt templates, generated corpora, compact per-run results, statistical-analysis scripts, and scripts for rebuilding the targets and reproducing the campaign[1].

ACKNOWLEDGMENT

This work was carried out as part of the first author's master's thesis research at Université du Québec à Montréal under the supervision of the third author. The first author thanks the second author for valuable feedback during the writing of this paper. Computations were made on the supercomputer Narval, managed by Calcul Québec and the Digital Research Alliance of Canada (Alliance). The operation of this supercomputer is funded by the Alliance, le ministère de l'Économie, de l'Innovation et de l'Énergie du Québec (MEIE) and le Fonds de recherche du Québec (FRQ).

[1]https://doi.org/10.5281/zenodo.21763875

REFERENCES

[1] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020.

[2] R. Swiecki, "Honggfuzz: Security oriented software fuzzer," https://github.com/google/honggfuzz, 2020.

[3] Google, "OSS-Fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2024, accessed: 2024.

[4] M. Rash, "afl-cve: A collection of vulnerabilities discovered by the AFL fuzzer," https://github.com/mrash/afl-cve, 2026, accessed: 2026.

[5] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 230–243.

[6] Y. L. Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," in *ISSRE 2007, The 18th IEEE International Symposium on Software Reliability, Trollhättan, Sweden, 5-9 November 2007*. IEEE Computer Society, 2007, pp. 93–102.

[7] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 206–215.

[8] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 126:1–126:13.

[9] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 423–435.

[10] H. Zhang, Y. Rong, Y. He, and H. Chen, "LLAMAFUZZ: large language model enhanced greybox fuzzing," *CoRR*, vol. abs/2406.07714, 2024.

[11] A. Wen, N. A. Alzahrani, J. Jiang, A. Joe, K. Shieh, A. Zhang, B. Alomair, and D. A. Wagner, "Seedaichemy: Llm-driven seed corpus generation for fuzzing," in *IEEE International Conference on Data Mining, ICDM 2025 - Workshops, Washington, DC, USA, November 12-15, 2025*. IEEE, 2025, pp. 1258–1267.

[12] M. Zalewski, "American fuzzy lop: A security-oriented fuzzer," https://lcamtuf.coredump.cx/afl/, 2014, technical whitepaper available at https://lcamtuf.coredump.cx/afl/technical_details.txt.

[13] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," vol. 45, no. 5, 2019, pp. 489–506.

[14] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Software Eng.*, vol. 50, no. 4, pp. 911–936, 2024.

[15] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *CoRR*, vol. abs/2304.02014, 2023.

[16] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative AI," *CoRR*, vol. abs/2306.06782, 2023.

[17] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[18] C. Shou, J. Liu, D. Lu, and K. Sen, "Llm4fuzz: Guided fuzzing of smart contracts with large language models," vol. abs/2401.11108, 2024.

[19] J. Eom, S. Jeong, and T. Kwon, "Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation," *CoRR*, vol. abs/2402.12222, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.12222

[20] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, Seattle, WA, USA, 17 July 2023*, M. Böhme, Y. Noller, B. Ray, and L. Szekeres, Eds. ACM, 2023, pp. 31–38.

[21] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," *CoRR*, vol. abs/2402.00350, 2024.

[22] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li, and Q. Zhang, "When fuzzing meets llms: Challenges and opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 492–496.

[23] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 230:1–230:36, 2022.

[24] S. Mallissery and Y. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 71:1–71:38, 2024.

[25] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecur.*, vol. 1, no. 1, p. 6, 2018.

[26] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2123–2138.

[27] Y. Wang, P. Jia, L. Liu, and J. Liu, "A systematic review of fuzzing based on machine learning techniques," *CoRR*, vol. abs/1908.01262, 2019. [Online]. Available: http://arxiv.org/abs/1908.01262

[28] P. Jha, J. Scott, J. S. Ganeshna, M. Singh, and V. Ganesh, "Bertrlfuzzer: A BERT and reinforcement learning based fuzzer (student abstract)," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 23 521–23 522.

[29] T. Parr, S. Harwell, K. Domino, and contributors, "grammars-v4: A collection of formal grammars written for ANTLR v4," https://github.com/antlr/grammars-v4, 2024, accessed: 2025.

[30] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. A. Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 1974–1993.

[31] M. Eceiza, J. L. Flores, and M. Iturbe, "Improving fuzzing assessment methods through the analysis of metrics and experimental conditions," *Comput. Secur.*, vol. 124, p. 102946, 2023.

[32] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[33] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[34] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.