2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

ReFuzzer: Feedback-Driven Approach to Enhance Validity of LLM-Generated Test Programs

1st Iti Shree
King's College London
London, United Kingdom
iti.shree@kcl.ac.uk

2nd Karine Even-Mendoza
King's College London
London, United Kingdom
karine.even_mendoza@kcl.ac.uk

3rd Tomasz Radzik
King's College London
London, United Kingdom
tomasz.radzik@kcl.ac.uk

***Abstract*—Existing LLM-based compiler fuzzers often produce syntactically or semantically invalid test programs, limiting their effectiveness in exercising compiler optimisations and backend components. We introduce `ReFuzzer`, a framework for refining LLM-generated test programs by systematically detecting and correcting compilation and runtime violations (e.g. division by zero or array out-of-bounds accesses). `ReFuzzer` employs a feedback loop with a local LLM to validate and filter erroneous programs before execution, improving fuzzing effectiveness beyond crash detection and enabling the generation of diverse yet valid test programs.**

We evaluated `ReFuzzer`'s effectiveness across black-, grey- and white-box fuzzing approaches targeting LLVM/Clang. `ReFuzzer` improved test programs' validity from 47.0–49.4% to 96.6–97.3%, with an average processing time of 2.9–3.5 s per test program on a dual-GPU machine. Further, refuzzing significantly increased code coverage in critical optimisation and IR generation components. For example, *vectorization* coverage had an absolute of 9.2%, 2.3%, and 7.1% improvement in black-, grey-, and white-box fuzzing, enhancing testing effectiveness.

***Index Terms*—Compiler Fuzzing, Large Language Models**

I. Introduction

Compiler fuzzing has uncovered thousands of bugs, leading to fixes and significantly contributing to compiler reliability over the past decade [1]–[4]. The emergence of *large language models* (LLMs) in code generation has made them valuable for compiler fuzzing [4], [5] due to their language-agnostic nature and ability to generate diverse programs. Such test programs are primarily used to detect compiler hangs and crashes (where the compiler fails to complete due to internal errors). It is because LLMs often struggle to produce valid code, reporting a 23.02%–49.05% static validity rate during fuzzing [4].

We follow the terminology of `GrayC` [3] and classify program validity as follows. A program is **statically valid** if it conforms to the language specification and is expected to compile without errors. A program is **dynamically valid** if it produces a well-defined, deterministic result at runtime, without triggering undefined, unspecified, or implementation-defined behaviours (e.g. division by zero or array out-of-bounds accesses). Language-defined failures like exceptions or return code of `-1` are allowed if they conform to the language semantics.

Testing for miscompilation (silent errors during compilation that result in an incorrectly compiled program execution) is impossible with *statically invalid* code, as such programs fail to compile entirely, producing no binary. Testing with *dynamically invalid* programs can lead to false positives, where the observed issue is in the test program itself (e.g. division by zero), rather than genuine compiler bugs.

In this tool paper, we propose a lightweight approach to enhance the static and dynamic validity test programs in fuzzing. Our new tool, `ReFuzzer`, seamlessly integrates into the LLM-based fuzzing workflow, demonstrated in the evaluation with black-, grey-, and white-box LLM-based fuzzing. The design of `ReFuzzer` utilises systematic feedback loop repair mechanisms, where compiler errors and sanitizer warnings guide iterative improvements of invalid test programs via LLM-suggested fixes of compiler and runtime issues, substantially increasing the proportion of valid test programs. `ReFuzzer` leverages locally deployed models as a safer alternative by preventing exposure of potentially sensitive code to external services, a crucial factor for industrial applications' developers.

We assess the validity rate of C programs and code coverage improvement on the LLVM/Clang compiler using black-, grey-, and white-box LLM-based fuzzers using various computer hardware. `ReFuzzer` significantly improved the static and dynamic validity rate and code coverage. Results show a strong dependency on hardware choice, with GPU-based configurations outperforming CPU-only setups. On GPU machines, `ReFuzzer` increased the validity rate from 47.0–49.4% to 96.8–97.3%, and improved code coverage in the LLVM/Clang compiler codebase by an absolute 0.3–21.2 percentage points across different compiler components.

Tool Availability. A video demonstration showcasing `ReFuzzer` end-to-end is available at [6]. `ReFuzzer` with data and results, is freely accessible as a Zenodo record [7] and GitHub [8]. `ReFuzzer` can be installed via README or a pre-built Docker [9]. Greybox and whitebox fuzzing were extended to utilise open-source LLMs via PR [10], [11].

II. ReFuzzer Design

`ReFuzzer` extends LLM-based fuzzing tools, enhancing the validity of LLM-generated test programs. It aims to address the gap between generating interesting test programs for testing compilers and the low validity rate common in LLM-based fuzzing. We refer to a program as valid if it is statically and dynamically valid.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00364

Overview.

Large Language Models (LLMs) are trained on publicly available code from sources like GitHub and are therefore likely to generate similar code snippets. To improve validity rates, we designed a feedback-driven, error-fixing loop using local LLMs. Failed attempts are retained for crash testing. Importantly, `ReFuzzer` does not preserve the original program's semantics during fixes; this is inconsequential, as the seeds are inherently random, a general characteristic of fuzzing.

Figure 1 illustrates the systematic workflow of our approach: 1) `ReFuzzer` takes C programs from LLM-based fuzzers like `WhiteFox` or `Fuzz4All` and checks whether they compile and pass sanitizer analysis. 2) Once a test program is detected to be statically or dynamically invalid, `ReFuzzer` captures warnings and errors from the compiler or code sanitisers together with the C program and feeds it to a local LLM model using the following template:

> **Prompt template for fixing C test programs**
>
> "Given the following C program and its compilation error log with «`ARG1`» optimisation level, analyze and correct the program to resolve «`ARG2`».\n«`PROGRAM-TO-FIX`»"

`ARG1` is a compilation level (e.g.-O0) and `ARG2` is the error type ("compilation errors" or "sanitizer errors). Step 2 is repeated up to n times until it succeeds in fixing all issues or the attempt limit is reached. 3) `ReFuzzer`, if succeeds, outputs a C program that is statically and dynamically valid.

Feedback-Driven Error-Fixing Mechanism. `ReFuzzer` utilises a feedback-driven approach as shown in Figure 1 between the test program code, the errors and warnings log (in yellow) and a local LLM (in grey-purple). `ReFuzzer` identifies invalid LLM-generated test programs by analysing compilation errors, warnings, and sanitiser outputs. These, along with the source code, are fed into a refinement loop powered by a local LLM. The loop suggests fixes, which `ReFuzzer` applies and verifies to produce valid test programs. Unfixable cases are moved to a crash-only folder and excluded from the seed bank to maintain quality. We use compilation output for static validity checks and code sanitizers for dynamic validity [12]–[15].

Improving the validity rate of LLM-generated programs can lead to higher compiler code coverage, as valid programs are more likely to progress beyond the frontend and trigger middle- and backend optimisations, paths that invalid programs cannot reach. Example 1 demonstrates the result of refuzzing code.

Example 1: `ReFuzzer` receives an invalid program shown in Figure 2. The original program contains assembly syntax errors and unsafe memory operations. The ASAN error.log:

```
... ==2352008==ERROR: AddressSanitizer: stack-buffer-overflow
on address 0x7b4a700de02a at pc 0x5600114ff85b bp
0x7ffda0bb6f20 sp 0x7ffda0bb66d0 ...
```

`ReFuzzer` fixes it and outputs program shown in Figure 3, free of ASAN errors.

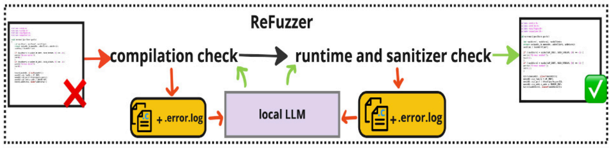

Fig. 1: Overview of `ReFuzzer`.

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
  // Use -O1 flag to enable optimization level 1
  __asm__("echo %1 > /proc/sys/kernel/osrelease\n", argv[0]);
  // Use deadargelim flag to eliminate dead arguments
  int num_args = sizeof(argv) / sizeof(argv[0]);
  for (int i = 0; i < num_args; i++) {
    if (!strcmp(argv[i], "-dead")) {
      argv[i] = NULL;
    }
  }
  // Use Collection of optimization passes
  int ret = vsprintf_s(NULL, "Hello, world! (%d args)\n", num_args);
  if (ret < 0) {
    printf("vsprintf_s failed: %d\n", ret);
  } else {
    printf("%s\n", argv[0]);
  }
  return 0;
}
```

Fig. 2: Original program generated via a black-box fuzzing calling `Ollama llama3.2`.

```c
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  // Ensure we have valid arguments
  if (argc < 1 || argv == NULL || argv[0] == NULL) {
    return 1;
  }
  // Use -O1 flag to enable optimization level 1
  __asm__("echo %0 > /proc/sys/kernel/osrelease\n" : : "r" (argv[0]));
  // Use deadargelim flag to eliminate dead arguments
  int num_args = argc;
  for (int i = 0; i < num_args && argv[i] != NULL; i++) {
    if (!strcmp(argv[i], "-dead")) {
      argv[i] = NULL;
    }
  }
  // Use Collection of optimization passes
  char buffer[512]; // Increased buffer size for safety
  int ret = vsprintf(buffer, "Hello, world! (%d args)\n", num_args);
  if (ret < 0) {
    printf("vsprintf failed: %d\n", ret);
  } else {
    printf("%s\n", buffer); // Print formatted buffer instead of argv
      [0]
    if (argv[0] != NULL) {
      printf("Program name: %s\n", argv[0]);
    }
  }
  return 0;
}
```

Fig. 3: Program fixed with `ReFuzzer`.

`ReFuzzer` fixes errors in inline assembly instructions, replaces non-standard functions with standard equivalents, adds memory safety checks and corrects buffer allocations, and eliminates potential dynamic errors reported by sanitizers.

Utilising Local LLMs. We design a refinement loop using *local* LLMs to fix invalid programs. Based on suggestions from the refinement loop, `ReFuzzer` applies targeted edits via an LLM and generates a corrected test program, which is then re-verified using the prompt template along with code analysis outputs from the refuzzed program. In this design, `ReFuzzer` executes models locally instead of cloud-based APIs, and can be configured to use any model available through `Ollama` [16] and open-source platforms like Hugging Face [17], allowing the user to substitute any locally runnable language model of their choice.

This flexibility allows users to balance computational requirements, LLM versions, and data privacy with their specific testing needs and hardware capabilities. This design ensures reproducible evaluation, offers high reusability for users without access to paid platforms, and keeps all code, error logs, and corrections within the user's security perimeter–critical for testing compilers with proprietary or security-sensitive code.

Implementation. We implemented ReFuzzer [7], [8] in C++ and Python to automate multiple binary executions and generate a test suite. Our current implementation supports refuzzing of C and C++ programs. We utilised LLaMA 3.2 LLM due to its strong ability to interpret error messages and apply fixes in C and C++. We set the number of refuzzing attempts to be two per test program.

Nonetheless, as LLM-based approaches are programming language agnostic, our implementation supports test suites generated by fuzzers in any programming language. Yet, since ReFuzzer relies on dynamic analysis via code sanitizers, it requires appropriate dynamic analysis for the language of the test programs. Extending ReFuzzer to non-C/C++ compilers requires integrating equivalent dynamic analysis tools for those languages.

III. Evaluation

A. Methodology

We evaluate ReFuzzer's quality and efficiency across three configurations, black-, grey-, and white-box LLM-based fuzzing *with locally installed LLMs*, by asking:

RQ1: *To what extent does ReFuzzer improve the validity rate of LLM-generated test programs by automatically detecting and fixing invalid cases through refuzzing?*

RQ2: *What is the impact on validity rates when executed on CPU vs GPU architectures?*

RQ3: *How effectively does ReFuzzer increase compiler code coverage beyond the frontend, particularly in the middle and back ends, across the three different configurations?*

Fuzzing Configurations Approaches. We evaluate:

(1) BlackBox. Our LLM-based black-box fuzzer for generating C test programs using Ollama, leveraging general programming and optimisation keywords as in [18].

(2) Fuzz4All. A grey-box compiler fuzzer [4].

(3) WhiteFox. A white-box compiler fuzzer [5].

Hardware Configurations. LLMs are highly dependent on GPU acceleration: Using more powerful GPU clusters generally yields faster refinements, higher throughput, and better test generation quality at the cost of increased resource demands. To quantify this trade-off, we evaluated:

(i) CPU. Intel Xeon D-1548 (8 cores, 2.0 GHz, 64 GB RAM); no GPU acceleration.

(ii) GPU(x1). Intel Xeon Silver 4114 (20 cores, 2.2 GHz, 192 GB RAM); 1 NVIDIA Tesla P100 (12 GB).

(iii) GPU(x2). 2 AMD EPYC 7542 (64 cores, 2.9 GHz, 512 GB RAM); 2 NVIDIA Tesla V100S (32 GB each).

Experimental Procedure and Setup. We first generated a test suite of LLM-generated programs over 24 hours with BlackBox, Fuzz4All, and WhiteFox fuzzers. This results in 3 sets of test programs, one for each fuzzer. We stored them in persistent storage. We then refuzzed each set separately. ReFuzzer compiles each test program with Clang at -O0 to collect a compilation log, applies fixes if needed, and re-evaluates the corrected programs using sanitizers to ensure dynamic validity. We then assess the quality of the test suites with and without ReFuzzer intervention.

We fuzzed LLVM Clang 21.0.0 using GCOV 11.4.0 with LCOV 2.3.1 for coverage analysis. Each 24-hour fuzzing run used a 60 s timeout and 16 GB memory limit per refuzzing. We configured ReFuzzer to use Ollama [16] version 0.5.7 with LLaMA 3.2 across all 3 configurations.

B. Results

RQ1 & RQ2: Test Programs Validity Rate. For each fuzzer, BlackBox, Fuzz4All, and WhiteFox, we measured the throughput and percentage of valid test programs with and without ReFuzzer. We then compared the improvement in validity against the additional time it required.

Table I shows the number of valid test programs before ReFuzzer (B column) and after (A column) applying ReFuzzer, the total number of generated test programs (# Tests column), and the average processing time by ReFuzzer per test program (including up to one retry attempt when the initial fix fails; Time/Test column), across different hardware configurations (CPU, GPU(x1) and GPU(x2) column).

The results show a significant increase in valid test programs across all fuzzers after applying ReFuzzer, reaching a stable dynamic validity rate of 96.6-97.3% with GPU setups, and slightly lower rates of 55.0-80.7% on CPU-only configurations. On average, each test program took 14.2–15.1 s to be amended using a CPU-only setup, 4.6–5.2 s with GPU(x1), and 2.9–3.5 s with GPU(x2). While ReFuzzer significantly improves the validity rate, the time required for refuzzing is highly dependent on hardware. GPU-based setups offer a clear advantage, achieving results up to 4–5 times faster than CPU-only configurations. This emphasises the importance of hardware acceleration in LLM-based compiler fuzzing workflows.

RQ1 Answer. Yes, ReFuzzer improved static & dynamic validity rates to 96.6–97.3%, with a stable processing time of 2.9–3.5 s per test on GPU(x2) across all three fuzzers, demonstrating that ReFuzzer operates efficiently with local LLMs.

RQ2 Answer. GPU(x2) and GPU(x1) achieved the same validity rate (96.6–97.3%), with GPU(x1) requiring nearly twice as long. CPU-only execution was significantly longer with lower validity rates (55.0–80.7%).

RQ3: Code Coverage. We analysed ReFuzzer's impact on deep code coverage, focusing on execution paths that extend beyond the parser and frontend, which are likely unreachable when test programs contain compilation failures or are too simplistic. The results reported in RQ3 are based on the GPU(x2) configuration. Results for other hardware setups are included in the artifact and show similar trends, but are omitted here due to page limit [8].

TABLE I: Validity rates, number of tests, and ReFuzzing time per test across configurations.

Fuzzer	CPU				GPU(x1)				GPU(x2)			
	B Valid	A Valid	# Tests	Time/Test	B Valid	A Valid	# Tests	Time/Test	B Valid	A Valid	# Tests	Time/Test
BlackBox	(46) 24.0%	(155) 80.7%	192	14.2 s	(3821) 47.0%	(7892) 96.8%	8150	4.9 s	(4103) 47.0%	(8452) 96.8%	8732	3.1 s
Fuzz4All	(85) 47.9%	(198) 55.9%	285	15.1 s	(4042) 48.5%	(8041) 96.6%	8321	4.6 s	(4320) 48.5%	(8611) 96.6%	8911	2.9 s
WhiteFox	(25) 12.3%	(112) 68.3%	204	14.5 s	(3847) 49.4%	(7582) 97.3%	7791	5.2 s	(4117) 49.4%	(8116) 97.3%	8341	3.5 s

TABLE II: Function Coverage (%) via lcov analysis. DCE: Dead Code Elimination. Analysis on GPU(x2) set, see full results in [7]. (B) Raw coverage, (A) Enhanced coverage with ReFuzzer, Δ: Absolute percentage improvement.

Component	BlackBox			Fuzz4All			WhiteFox		
	(B)	(A)	Δ	(B)	(A)	Δ	(B)	(A)	Δ
Frontend (Parser)	60.0	62.0	+2.0	57.0	58.0	+1.0	12.7	13.5	+0.8
AST & Semantics	10.7	12.4	+1.7	28.3	29.0	+0.7	9.2	9.5	+0.3
IR Generation	33.0	43.2	+10.2	9.2	9.6	+0.4	7.2	7.6	+0.4
Opt. Passes	12.4	13.8	+1.4	0.2	0.4	+0.2	10.3	11.2	+0.9
• Loop Opt.	8.3	21.0	+12.7	1.1	3.4	+2.3	6.7	17.5	+10.8
• Vectorization	5.8	15.0	+9.2	0.6	2.9	+2.3	4.6	11.7	+7.1
• Inlining	11.8	33.0	+21.2	0.3	6.0	+5.7	9.2	26.0	+16.8
• DCE	17.0	34.0	+17.0	0.2	3.8	+3.6	13.4	26.5	+13.1
Backend Code Gen.	6.3	6.8	+0.5	7.0	7.3	+0.3	2.8	3.9	+1.1

Table II presents *function coverage* measured via lcov and gcov instrumentation on the LLVM/Clang codebase. We evaluated function coverage percentage for LLM-generated test programs executed with the BlackBox, Fuzz4All, and WhiteFox fuzzers, before (B column) and after (A column) refuzzing, with the coverage improvement presented in absolute percentage improvements (Δ column). The components in the table highlight improvements in the frontend (parsing and semantic analysis), intermediate representation (IR) generation, optimisation passes (e.g., loop optimisations, inlining, and dead code elimination) and backend code generation.

Our results show that ReFuzzer enhances function coverage, particularly in optimisations such as loop optimisation, inlining, and dead-code elimination (DCE). The highest improvement was observed with BlackBox, achieving gains in inlining (+21.2%), DCE (+17.0%), and loop optimisation (+12.7%). Followed by WhiteFox refuzzing, achieving significant improvements, particularly in inlining (+16.8%) and DCE (+13.1%). ReFuzzer had a lower yet meaningful impact on Fuzz4All, notably in inlining (+5.7%) and DCE (+3.6%). ven for Fuzz4All, the overall *Opt. Passes* coverage increased from 0.2% to 0.4% (doubling the coverage). This demonstrates that ReFuzzer's improvements in test program validity enhance coverage beyond frontend components.

RQ3 Answer. Yes, ReFuzzer effectively increases compiler code coverage beyond the frontend, most significantly in compiler optimisations, with improvements of up to +21.2% in inlining, +17.0% in dead code elimination, and +12.7% in loop optimisation across the evaluated fuzzers.

IV. Related Work & Conclusion

Code LLMs such as StarCoder [19] excel at code generation. Recent fuzzing approaches like WhiteFox [5] and Fuzz4All [4] leverage LLMs but depend on costly, closed-source models like GPT-4 [20] and often produce low-validity programs, as discussed in §I. Notably, RoCode [21] has explored improving code quality with a focus on static validity in general programming tasks, while our focus is on enhancing compiler code coverage in the context of fuzzing via both static and dynamic validity.

In future work, we plan to explore corpus and test program minimisation techniques aimed at maximising compiler code coverage while reducing refuzzing costs, an increasingly relevant objective in the context of LLM scalability and green computing. Further, we will explore the adoption of ReFuzzer to other programming languages, LLM-based tools, and fuzzing approaches. Building on the findings of RQ2, we will experiment with ChatGPT-style platforms on distributed GPU clusters to assess how backend architecture influences test validity and repair effectiveness compared to local LLM deployments.

References

[1] X. Yang, Y. Chen *et al.*, "Finding and understanding bugs in c compilers," in *PLDI 2011*. ACM, 2011, pp. 283–294.

[2] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, p. 216–226, Jun. 2014.

[3] K. Even-Mendoza, A. Sharma *et al.*, "GrayC: Greybox fuzzing of compilers and analysers for C," in *ISSTA 2023*. ACM, July 17-21 2023, pp. 1219–1231.

[4] C. S. Xia, M. Paltenghi *et al.*, "Fuzz4all: Universal fuzzing with large language models," in *ICSE 2024*. ACM, 2024.

[5] C. Yang, Y. Deng *et al.*, "Whitefox: White-box compiler fuzzing empowered by large language models," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024.

[6] "ReFuzzer demonstration video," https://tinyurl.com/cb3cm527, 2025, video demonstration.

[7] I. Shree, K. Even-Mendoza, and T. Radzik, "Arrtifact of ReFuzzer in zenodo," Mar. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16385323

[8] "ReFuzzer: Reproducibility artifact and docker image," https://github.com/nmdis1999/ReFuzzer/, 2025, gitHub repository.

[9] "ReFuzzer: Pre-built docker image," https://hub.docker.com/repository/docker/shreei/refuzzer_ase/general, 2025, docker image.

[10] "Adding support for ollama models," https://github.com/fuzz4all/fuzz4all/pull/11, July 2025, pull Request #11 to fuzz4all/fuzz4all.

[11] "Adding support for ollama models," https://github.com/ise-uiuc/WhiteFox/pull/15, July 2025, pull Request #15 to ise-uiuc/WhiteFox.

[12] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *WBIA '09*, ser. WBIA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 62–71.

[13] K. Serebryany, D. Bruening *et al.*, "Addresssanitizer: a fast address sanity checker," in *USENIX ATC'12*, ser. USENIX ATC'12. USA: USENIX assoc., 2012, p. 28.

[14] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *CGO '15*, ser. CGO '15. USA: IEEE CS, 2015, p. 46–55.

[15] LLVM Project, "Undefined Behavior Sanitizer (UBSan)," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017.

[16] "Ollama," https://ollama.com/.

[17] H. Face, https://huggingface.co, 2024.

[18] A. Dakhama, K. Even-Mendoza *et al.*, "Searchgem5: Towards reliable gem5 with search based software testing and large language models," in *SSBSE 2023*. Springer, 2023.

[19] R. Li, L. B. Allal *et al.*, "Starcoder: may the source be with you!" 2023.

[20] J. Achiam, S. Adler *et al.*, "Gpt-4 technical report," 2023.

[21] X. Jiang, Y. Dong *et al.*, "ROCODE: Integrating Backtracking Mechanism and Program Analysis in Large Language Models for Code Generation ," in *ICSE 2025*. IEEE CS, May 2025, pp. 334–346.