LegoFuzz: Interleaving Large Language Models for Compiler Testing

Yunbo Ni
Nanjing University
Nanjing, China
yunboni@smail.nju.edu.cn

Abstract

Using large language models (LLMs) to test compilers is promising but faces two major challenges: the generated programs are often too simple, and large-scale testing is costly. We present a new framework that splits the process into an offline phase—where LLMs generate small, diverse code pieces—and an online phase that assembles them into complex test programs.

Our tool, LegoFuzz, applies this method to test C compilers and has discovered 66 bugs in GCC and LLVM, including many serious miscompilation bugs that prior tools missed. This efficient design shows strong potential for broader AI-assisted software testing.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**.

Keywords: Compilers, Testing, Reliability

ACM Reference Format:
Yunbo Ni. 2025. LegoFuzz: Interleaving Large Language Models for Compiler Testing. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3758316.3763251

1 Introduction

Compilers are critical in today's software ecosystem, yet they remain vulnerable to bugs despite years of improvement [1]. To address this, researchers have developed various testing techniques, including random program generation [7–9, 15] and mutation-based methods [2–6, 12].

Recently, large language models (LLMs) have opened up new opportunities in compiler testing. Tools like Fuzz4All [13] and WhiteFox [14] use LLMs to generate test programs based on prompts, aiming to improve the diversity and effectiveness of compiler fuzzing. However, practical adoption of these tools still faces two major challenges:

➤ **Challenge 1: Low-quality test programs.** LLMs often produce overly simple or invalid code—syntactically incorrect or semantically unsound—making them ineffective for uncovering deep compiler bugs such as miscompilations. Existing LLM-based tools mostly report frontend crashes, with generated programs short and lacking complexity.

➤ **Challenge 2: High computational cost.** Embedding LLMs in the testing loop is expensive. Systems like Fuzz4All yield only tens of thousands of valid programs per day, far fewer than traditional fuzzers that generate millions efficiently, making large-scale LLM-based fuzzing impractical.

➤ **Our Idea and Approach.** To address the limitations of current LLM-based compiler testing, we decouple the process into two phases: an **offline phase** and an **online phase**.

In the *offline phase*, we employ LLMs to generate small yet diverse code snippets, guided by real-world templates and filtered for syntactic and semantic validity. Over 500,000 single-function samples from 146 open-source projects (e.g., Linux, Redis, Nginx) are collected into a large code database.

In the *online phase*, we compose these functions into complex test programs by inserting function calls and sharing global variables under type and semantic constraints, enabling coherent, large-scale, and cost-efficient fuzzing without invoking LLMs during testing.

2 Our Approach: LegoFuzz

We introduce LegoFuzz, whose core idea is to separate the testing process into offline and online phases. Figure 1 shows the high-level workflow of LegoFuzz. In this section, we describe the technical details of our LegoFuzz framework. Section 2.1 details the offline code collection with large language models, while Section 2.2 introduces the online iterative program synthesis.

2.1 Offline Code Database Construction

The goal of the offline phase is to construct a reusable database of validated, expressive C functions using large language models (LLMs). These functions serve as building blocks for test program synthesis in the online phase.

SPLASH Companion '25, Singapore, Singapore

ACM ISBN 979-8-4007-2141-0/25/10
https://doi.org/10.1145/3758316.3763251

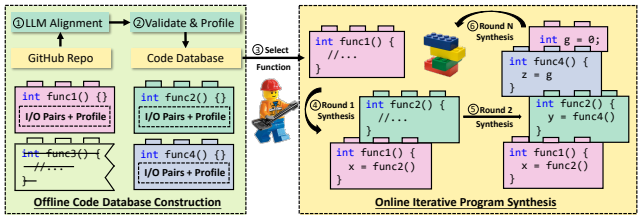

Figure 1. Design overview of LegoFuzz.

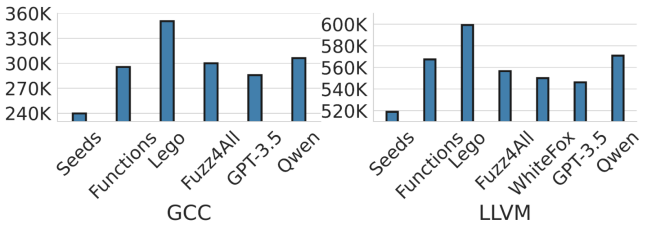

Figure 2. Line coverage of different variants

Each entry in the database $\mathcal{D}_F$ is a pair $E_i = \{F_i, \widehat{prof}_i\}$, where F_i is a self-contained function, and $\widehat{prof}_i$ is a runtime profile containing input/output values and expression states.

➤ **Code Generation via Real-world Alignment.** Direct LLM prompting often fails to produce diverse and complex code patterns. To address this, we introduce *real-world code-aligned prompting*, which guides LLMs to transform real-world functions from open-source projects into numeric, single-function representations. This improves both expressiveness and controllability. Prompts are designed to enforce:

- **Syntax-level alignment:** The output must be a single function with numeric-type inputs and outputs, and no extra declarations.
- **Semantics-level alignment:** The transformed function must preserve the logic of the original, flattening complex types and inlining helpers as needed.

➤ **Validation and Profiling.** Generated functions undergo two-stage validation: compilation for syntactic correctness, followed by randomized execution with sanitizers [10, 11] to detect undefined behavior. Only passing functions are retained. For each, we record a runtime profile $\widehat{prof}_i$ capturing input/output values and per-line expression results.

As shown on the left of Figure 1, in Step ① we use LLMs to transform real-world code into functions. In Step ②, we validate each function via compilation and runtime checks. Valid ones (e.g., `func1`, `func2`, `func4`) are profiled and stored in the database, while invalid ones (e.g., `func3`) are discarded.

2.2 Online Iterative Program Synthesis

To effectively expose compiler bugs, test programs must feature complex control flows and inter-function interactions. Simply aggregating independent functions is inadequate. We thus adopt an iterative synthesis approach that combines functions from the offline database with semantic dependencies built through two mechanisms: (1) *function call insertion*, guided by runtime profiles to preserve behavior, and (2) *global variable sharing*, which establishes cross-function data dependencies.

➤ **Synthesis Procedure.** Given a code database $\mathcal{D}_F$ and an iteration count $\mathcal{N}$, the synthesis proceeds as follows:

Step 1. *Seed Selection:* Select a random function from $\mathcal{D}_F$ and generate a driver program to invoke it.

Step 2. *Expression Matching:* Identify replaceable expressions from the selected function based on runtime profiles.

Step 3. *Dependency Construction:* For each matched expression, synthesize a dependency by either inserting a function call or using a shared global variable.

This process is repeated for $\mathcal{N}$ iterations, gradually expanding the program while preserving its semantics. By leveraging runtime profiling, the synthesis ensures that all transformations maintain valid execution behavior.

As shown in the right part of Figure 1, the synthesis begins in Step ③ by selecting `func1` as the seed. In Step ④, `func1` is expanded by replacing an expression with a call to `func2()`, guided by profiling data. Then, in Step ⑤, additional dependencies are introduced: `func2` calls `func4`, and `func4` reads a shared global variable `g`. This process continues in Step ⑥, enabling further reuse of building blocks.

3 Evaluation

We evaluate LegoFuzz on bug-finding capability and coverage improvement.

➤ **Bug finding.** LegoFuzz discovered **66 compiler bugs**, including **30 miscompilations**, with **56** already fixed. Unlike Fuzz4All and WhiteFox, which mainly trigger frontend crashes, LegoFuzz exposes deep optimization bugs—some hidden for nearly **20 years**[1] .

➤ **Coverage improvement.** As shown in Figure 2, LegoFuzz increases coverage by **12.5%** in GCC and **4.9%** in LLVM over Seeds and Functions, and covers about **20K** more lines in GCC and **50K** more in LLVM than Fuzz4All and WhiteFox. Replacing GPT-4o with GPT-3.5 or Qwen still outperforms Seeds, confirming LegoFuzz 's **generalizability**.

4 Conclusion

We present LegoFuzz, a compiler testing framework that combines LLM-generated code snippets with iterative synthesis. It has found 66 bugs in GCC and LLVM, including 30 miscompilations—far more than prior LLM-based tools. LegoFuzz shows that simple, reusable building blocks can enable scalable and effective compiler testing. We hope it inspires future work in this area.

[1]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118915

Acknowledgement

This work is conducted under the supervision of Minxue Pan and Shaohua Li.

References

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (2020). doi:10.1145/3363562

[2] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 16)*. 85–99. doi:10.1145/2980983.2908095

[3] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. 1219–1231. doi:10.1145/3597926.3598130

[4] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. 38.

[5] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 14)*. 216–226. doi:10.1145/2666356.2594334

[6] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA' 15)*. 386–399. doi:10.1145/2858965.2814319

[7] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (2020). doi:10.1145/3428264

[8] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (2023). doi:10.1145/3591295

[9] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 187–196. doi:10.1145/2491956.2491967

[10] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. 157–157. doi:10.1109/SecDev.2016.043

[11] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code.

[12] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications (OOPSLA' 16)*. 849–863. doi:10.1145/3022671.2984038

[13] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. 1–13. doi:10.1145/3597503.3639121

[14] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (2024). doi:10.1145/3689736

[15] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 283–294. doi:10.1145/1993498.1993532

Received 2025-07-07; accepted 2025-08-12