On the Practicality of LLM-Based Compiler Fuzzing

Gabriel Guimarães dos Santos Ricardo
UFMG
Belo Horizonte, Brazil
gabriel.guimaraes@dcc.ufmg.br

Natanael dos Santos Junior
UFMG
Belo Horizonte, Brazil
natanael0junior@dcc.ufmg.br

Flavio Figueiredo
UFMG
Belo Horizonte, Brazil
flaviovdf@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

Abstract

Recently, Italiano and Cummins introduced an elegant methodology for uncovering performance bugs in compilers. Their approach involves using a pre-trained large language model (LLM) to generate a seed program, followed by successive mutations designed to provoke unexpected behavior, even in mainstream compilers. This methodology is particularly appealing due to its language-agnostic nature: it can be adapted to different programming languages without the need to develop a dedicated fuzzer for each one. Moreover, it has proven highly effective, uncovering previously unknown (zero-day) performance bugs in widely used compilers such as Clang, ICC, and GCC. In an effort to reproduce the results reported by Italiano and Cummins, we confirm that their technique outperforms general-purpose LLMs, such as open-source versions of LLaMA and DeepSeek, in identifying compiler performance bugs. However, we also observe that while the LLM-based approach is commendable, it lags behind tools like CSmith in terms of throughput (the number of bugs found over time) and latency (the time to discover the first bug). LLMs also require significantly greater computational resources. Although this outcome may seem discouraging, it is important to note that we are comparing novel LLMs with a mature language-specific fuzzer. Nevertheless, as technology evolves, we expect the performance of LLM-based fuzzing to improve, potentially surpassing traditional methods in the future.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Compilers**.

Keywords: Fuzzing, Large-Language-Model, Compiler

1 Introduction

In February 2025, Italiano and Cummins [6] introduced a new methodology for finding bugs in compilers. Their technique combines large language models (LLMs) with differential testing. Italiano and Cummins use LLMs not only to generate base test programs but also to produce mutations of those programs. The goal is to detect situations where compilers fail to optimize code as expected. Their approach is lightweight, adaptable to multiple languages (such as C/C++, Rust, and Swift), and has already uncovered confirmed bugs in production compilers, including `clang`, `gcc`, and `icc`.

The work of Italiano and Cummins has sparked considerable discussion. Unlike other techniques that apply LLMs to assist compiler development, their approach does not depend on the LLM producing fully correct or meaningful programs. Instead, as explained in Section 2, the LLM is used to uncover code-size regressions. It only needs to generate code that (i) compiles, passing syntactic and semantic checks, and (ii) causes a compilation mode (for example, `clang -O0`) to produce a binary larger than that produced by a size-optimized mode (for example, `clang -Oz`).

Carbon Questions. We hypothesize, however, that some of these results could be achieved with fewer resources using traditional compiler fuzzers. This paper investigates that hypothesis by addressing two practical questions:

- **Q1:** Can we find more code-size regressions using a traditional compiler fuzzer, such as CSmith [17] or YarpGen [8], instead of relying on a large language model to generate test cases?
- **Q2:** Is the gradual mutation strategy proposed by Italiano and Cummins more effective than a simpler single-shot approach, where test cases are generated independently, without reusing or evolving previous ones?

These questions aim to assess the practicality of Italiano and Cummins's methodology. Using LLMs to generate programs that expose compiler regressions is an elegant and promising idea, but it raises the question of whether similar outcomes could be achieved more efficiently using traditional techniques. Specifically, we ask whether comparable results can be obtained with lower consumption of human effort, computational time, and energy. This evaluation is particularly relevant because running large language models often incurs substantial computational costs, which may not always be justified if simpler alternatives provide similar effectiveness.

Summary of Findings: Using the setup described in Section 3, we reproduced the research environment of Italiano and Cummins and were able to replicate several of their results, including multiple code-size regressions, one of which is detailed in Section 2. This confirms part of the findings

from the work that motivated this study. However, we also observed that traditional compiler fuzzers can be substantially more effective than the LLM-based approach. Our main observations, described in Section 4, are now summarized:

1. A version of CSmith running on a commodity CPU produces nearly two orders of magnitude more test programs than local instances of state-of-the-art LLMs, even when those models are served on an A100 GPU (see Fig. 2).
2. Not every program generated by an LLM is syntactically or semantically valid; some fail to compile. In contrast, every program generated by CSmith compiles successfully. Interestingly, models such as CodeLlaMa and DeepSeek produce a higher percentage of compilable programs than the fuzzer YarpGen, which occasionally generates invalid code due to its less restrictive design (see Fig. 3).
3. The most effective LLM-based fuzzer in our experiments was LlaMa3.1. It achieved a slightly higher *regression density*, meaning the number of regressions divided by the total number of generated programs, compared to CSmith (see Fig. 4).
4. Despite this higher regression density, the absolute number of regressions found within a fixed time budget was nearly 20 times higher when using CSmith than when using LlaMa3.1. This difference is a direct consequence of the much higher throughput of CSmith (see Fig. 5).
5. The gradual mutation approach proposed by Italiano and Cummins proved significantly more effective than the single-shot approach when applied to LlaMa3.1. However, this advantage did not generalize to other LLMs; in those cases, gradual mutation failed to produce any regressions, whereas the single-shot approach succeeded in finding a few, albeit at a much lower rate compared to LlaMa3.1 (see Fig. 6).
6. Contrary to expectations, the LLM-based fuzzers that use models specialized for code did not demonstrate superior performance compared to general-purpose models. The general-purpose models that we have evaluated, namely Llama3.1 and DeepSeek R1 Distill Llama3.1, successfully detected regressions, whereas the specialized models did not. This observation suggests that code specialization alone may be insufficient for effective bug-detection tasks (this result is also available in Figure 6).

Although our results show that a traditional fuzzer can find more regressions than an LLM-based fuzzer within the same time budget, this does not mean that the LLM-based approach should be disregarded. It offers an important advantage, as Italiano and Cummins point out: the method is easily portable across programming languages. This portability stems from the fact that the approach relies on natural language prompts to guide the LLM rather than language-specific generators. Adapting the method to a new language only requires changing the seed program to match the target syntax and updating the compiler invocation. Italiano and Cummins support this claim by successfully applying the method to Rust and Swift, with minimal modifications, and discovering bugs in both. This stands in contrast to traditional fuzzers like CSmith, which demand substantial language-specific engineering effort.

2 Code-Size Regressions

In the context of a compiler, a "*Regression*" refers to any situation where a newer version of the compiler behaves worse than a previous version—in correctness, performance, or other quality metrics—for the same input program. The goal of Italiano and Cummins's work is to detect "*Code-Size Regressions*", a notion that Definition 2.1 formalizes.

Definition 2.1. A code-size regression is a situation where a newer version of a compiler or a supposedly more optimizing configuration (e.g., `-Os`, `-Oz`) generates a larger binary for a given program than an older version or a less optimizing configuration (e.g., `-O0`, `-O3`).

Identifying code-size regressions is important because they often reflect a missed opportunity for optimization, or even a degradation caused by a recent compiler change. Example 2.2 illustrates such a situation. The program in Figure 1 is an edited version of a very similar program produced using Italiano and Cummins's fuzzer.

Example 2.2. Figure 1 shows a code-size regression observed in `gcc 9.4.0`. In this case, `gcc -O2` produces a smaller binary than `gcc -Os`. Since `-Os` is explicitly designed to optimize for binary size—potentially at the expense of runtime performance—it is unexpected for `-O2`, which prioritizes performance over size, to yield a smaller result. This suggests that `-Os` may be missing an optimization opportunity. In this particular example, the smaller binary produced by `-O2` is due to *inlining*—an optimization that replaces a function call with the body of the function itself. The decision to apply inlining is governed by a cost model, which makes it more aggressive at `-O2` than at `-Os`, where inlining is more conservative to avoid code-size bloat. However, in Figure 1, the function in question is small and invoked only once. Inlining it avoids the overhead associated with a function call and leads to a smaller binary, highlighting a missed opportunity in the `-Os` pipeline.

3 Methodology

The methodology proposed by Italiano and Cummins [6] combines large language models (LLMs) with differential testing to uncover missed code-size optimizations in compilers. The key idea is to use an LLM to iteratively mutate

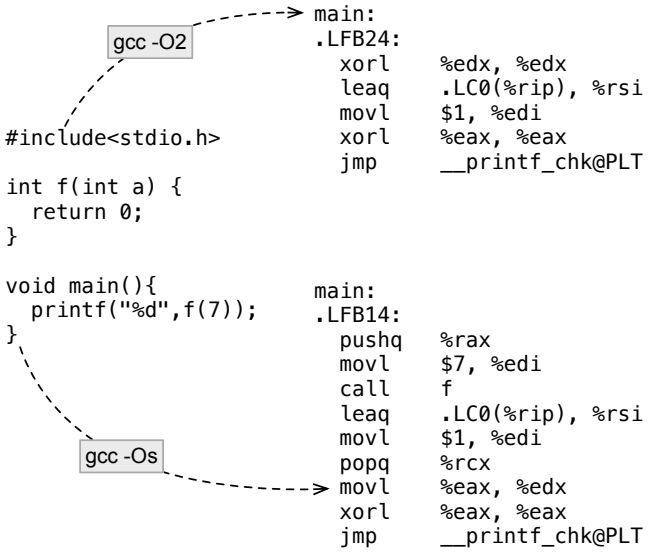

Figure 1. Example of a program where `gcc -Os` produces code that is larger than the binary produced by `gcc -O2` due to inlining.

small seed programs, progressively increasing their complexity, and then applying differential testing strategies to detect anomalies in compiler output, specifically code-size regressions. The approach consists of two main components:

1. **Code Mutation Using LLMs.** The process starts with a simple seed program, typically a trivial function that compiles in the target language. The LLM is prompted to mutate this code iteratively by sampling from a predefined set of mutation instructions. These instructions include transformations such as:
   - Adding control flow structures (e.g., loops, conditionals).
   - Modifying existing conditions to make them more complex.
   - Introducing pointers, arrays, structs, or unions.
   - Injecting dead code or nested constructs.

   Each mutation step produces a new version of the program, which is then passed to the testing framework.
2. **Differential Testing for Missed Optimizations.** After each mutation, the generated code is compiled under different configurations or compiler versions. The system applies four types of differential testing strategies to identify code-size regressions:
   - *Dead Code Differential Testing:* Adds dead code and checks whether the compiler eliminates it correctly.
   - *Optimization Pipeline Differential Testing:* Compares binaries generated with different optimization flags (e.g., `-Oz` vs. `-O2`).
   - *Single-Compiler Differential Testing:* Compares binaries produced by different versions of the same compiler to detect regressions over time.
   - *Multi-Compiler Differential Testing:* Compares output from different compilers to identify missed optimizations in one relative to another.

The process continues iteratively until a stopping criterion is met, such as reaching a maximum number of mutations, a compilation failure, or the discovery of a potential regression. When a code-size regression is detected, the framework applies heuristics to filter out false positives, including checks for compilation success, semantic correctness, and monotonic increases in program complexity.

Our Adaptations: We followed the methodology proposed by Italiano and Cummins, implementing it according to the description provided in the paper. In addition, we extended their setup by incorporating three additional LLMs (as described in Section 4) and two traditional compiler fuzzers, CSmith and YarpGen.

4 Evaluation

This section evaluates the following research questions:

RQ1: What is the "fuzzing throughput"; that is, the number of programs that are produced by the different fuzzers (which can be compiled or not) within a fixed amount of time.

RQ2: What is the absolute and relative rate of "compilable" programs that is produced by the LLM-based fuzzers and the traditional fuzzers.

RQ3: What is the percentage of regressions and absolute count of it that we observe on a fixed time span considering LLM-based fuzzers and traditional fuzzers.

RQ4: Considering LLM-based fuzzers only, how does the gradual mutation approach of Italiano and Cummins [6] compares with single-shot testing?

Experimental Setup: The compiling experiments discussed in this section run on an Intel Core i5 8250U 8-Core Processor 1.8 GHz, with 8 GB of RAM, running Linux Ubuntu v24.04.1. The LLM-based code generation runs on Google Colab's GPU Nvidia A100 / Nvidia L4 according to model requirements. The compiler under test is gcc v14.2.0. Italiano and Cummins's fuzzer uses an out-of-the-shelf implementation of LlaMa3.1, we ran the LlaMa3.1 Instructed 8B version. For comparison, we include five other models, DeepSeek R1 Distill LlaMa3.1 8B ,DeepSeekCoder.v2 16B,DeepSeekCoder.v1 7B, CodeLlaMa 13B and CodeLlaMa 7B. We use latest version of CSmith (v2.3.0) from June of 2017.

Code-Size Regressions: In this section, we constrain Definition 2.1 as follows: we consider a code-size regression a program that causes gcc v14.2.0 to produce a larger program at the `-O0` optimization level when compared to the `-Os` level. As we further explain in Section 6, a code-size regression is not necessarily an optimization bug, although it is a strong indication of it.

4.1 RQ1: Throughput

This section evaluates the difference between traditional fuzzing and LLM-based fuzzing by comparing the throughput of these methods. We define *throughput* as the number of programs that are both *generated* and *compiled* within a fixed amount of time. This measurement includes compilation time, since compiling the generated programs is an essential step for detecting code-size regressions. However, it is important to note that this section does not measure the number of code-size regressions themselves; that analysis is presented in Section 4.4. The goal here is to quantify throughput as a proxy for energy consumption, with carbon footprint considerations in mind, and to compare the efficiency of LLM-based fuzzers to that of traditional fuzzers.

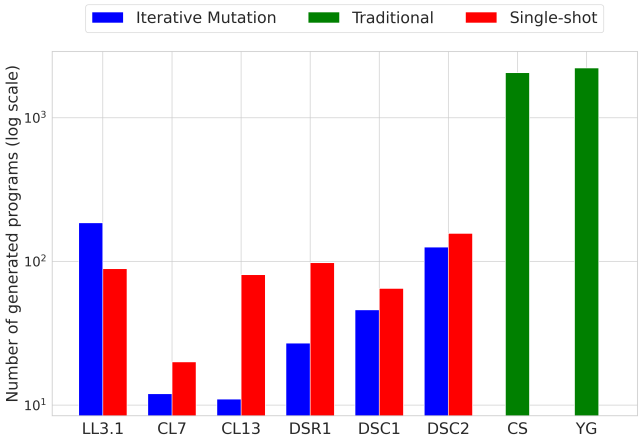

Figure 2. Number of programs generated within 3 hours using iterative mutation only. We use the following accronyms in this figure, and in every other figure in this section: `LL3.1` = LLaMA3.1 8B INSTRUCTED; `CL7` = CODELLAMA 7B INSTRUCTED; `CL13` = CODELLAMA 13B INSTRUCTED; `DSR1` = DEEPSEEK R1 DISTILL LLAMA3.1 8B; `DSC2` = DEEPSEEK-CODER V2 16B INSTRUCTED;`DSC1` = DEEPSEEKCODER V1 7B INSTRUCTED; `CS` = CSMITH; `YG` = YARPGEN;

Discussion: Figure 2 shows the results of this experiment. This figure, like the others that follow, includes three types of fuzzers: traditional fuzzers (CSMITH and YARPGEN), single-shot LLM-based fuzzers, and iterative LLM-based fuzzers—the latter being the approach proposed by Italiano and Cummins. The results in Figure 2 show that traditional fuzzers are able to generate substantially more programs than LLM-based fuzzers. Given a time budget of three hours, CSMITH produced 2063 programs, while YARPGEN generated 2222 programs.These numbers are significantly higher than those achieved by the LLM-based fuzzers.The best-performing LLM-based fuzzer was LLAMA3.1 in iterative mode, which produced 186 programs within three hours. When running in single-shot mode, its output dropped to 89 programs. The

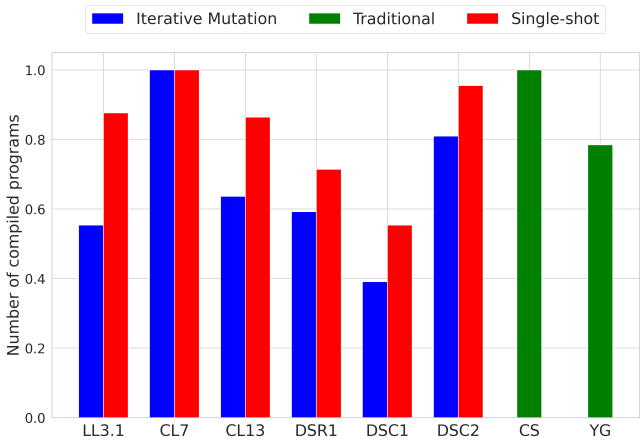

Figure 3. Percentual of compilable programs.

worst performance was observed with CODELLAMA7, which generated only 20 programs within the same time frame. Interestingly, the only LLM-based fuzzer that produced more programs in iterative mode than in single-shot mode was LLAMA3.1.For all other LLMs, the single-shot mode yielded a higher number of programs.This difference was particularly noticeable with CODELLAMA13, which generated 81 programs in single-shot mode compared to only 11 in iterative mode. We do not have a definitive explanation for this behavior, but it is worth noting that LLAMA3.1 is the same model used in Italiano and Cummins's original study.

4.2 RQ2: Percentage of Successful Compilation

We define the *Compilation Rate* of a model as the number of programs that could be successfully compiled using `gcc` v14.2.0 divided by the total number of programs produced by that model. The methodology reuses the experimental setup from Section 4.1: for each approach, we generate as many programs as possible within a fixed time budget of three hours, and check, for each program, if it compiles or not. The time to compile the programs is taken into consideration in this experiment, e.g., programs are compiled within the same budget of three hours.

Discussion: Figure 4 demonstrates that LLM-based fuzzers exhibit lower compilation rates under both code generation strategies compared to CSMITH (which achieves a 100% compilation rate, as expected for a traditional fuzzer). However, they outperform YARPGEN in some cases, with compilation rates approximating 78.49%. This result highlights the potential of LLM-based fuzzers, indicating their ability to generate syntactically valid code that passes compilation tests. It also makes it clear that traditional fuzzers are not guaranteed to generate always valid programs. As an example, CHIGEN [14], a Verilog fuzzer, produces about 60% of compilable code — an even lower rate than YARPGEN.

As hypothesized, iterative mutation yields lower compilation rates (56% for LLaMA3.1) than single-shot generation (87.64%). This trend is visible in the other models in figure 4 as well. This discrepancy arises from the inherent challenge of applying sequential mutations n-times while maintaining syntactic correctness, a limitation less prevalent in single-shot generation. Notably, compilation rates alone do not determine regression detection efficacy. As an example, LLaMA3.1's iterative mutation achieved superior regression rate despite its moderate compilation rate, a phenomenon we analyze in Section 4.4. In addition, models like CodeLLaMA 7B (100% compilation rate) and CodeLLaMA 13B (63.63% iterative, 86.42% single-shot) generated few programs (12 and 20 for iterative/single-shot respectively in 7B; 11 and 81 in 13B) and demonstrated poor performance in other metrics. Finally, we highlight the performance of DeepSeekCoder V2, which achieved notably high compilation rates among LLM-based fuzzers. The model generated 157 programs in single-shot mode (compilation rate: 95.54%) and 126 in iterative mutation (compilation rate: 80.95%), demonstrating robust performance across both strategies. These results position DeepSeekCoder V2 as one of the most effective LLM-based approaches in our evaluation.

4.3 RQ3: Regressions

This section discusses the regression rate of each LLM-based model in comparison with traditional fuzzers. The rate is calculated based on the number of compiled programs, relative to the total number of programs generated over a 3-hour period of extensive generation. It is worth noticing that not all models found regressions in `GCC 14.3.0` within the 3-hour window. However, this does not necessarily imply that they are incapable of doing so. The LLM-based models were not specifically trained for regression-triggering code generation. Therefore, the results reflect the raw capabilities of the models, which could potentially be improved with task-specific fine-tuning.

Discussion: Figure 4 presents the fraction of compiled programs that led to regressions. LLAMA 3.1 in iterative mutation mode demonstrates the best performance, surpassing other LLM-based models and competing with traditional fuzzers. This model yielded a regression rate of 4.854%; hence, producing a regression once for each twenty compiled programs. All evaluated LLM-based fuzzers exceed Yarpgen regression rate of 0.0573% (which produced only one regression). This advantage, however, must be contextualized: CSmith achieved a regression rate of 4.65%. Although similar to Llama 3.1's regression rate, in absolute terms, CSmith does much better, producing 96 total regressions in a three-hour fuzzing campaign. The underperformance of Yarpgen relative to CSmith (Figure 5) indicates variability among traditional fuzzers, with some being outperformed by current LLM approaches.

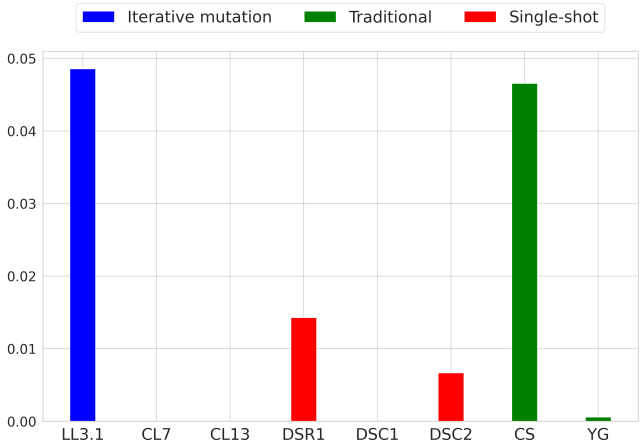

Figure 4. Proportion of regressions among compiled programs.

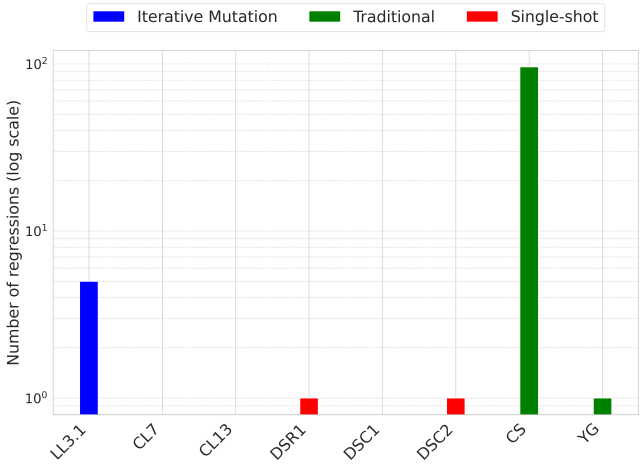

Figure 5. Model total regression comparison.

However, since the total number of programs generated by LLMs is significantly smaller than that of traditional fuzzers, it is important to also consider the proportion of compiled programs when interpreting these results, which suggest a better perspective for LLM-based fuzzers. Considering regression rate and absolute number of regressions is important, because LLMs currently demand considerable computational resources and energy to generate programs. Nevertheless, the trend points toward reduced costs over time, which may eventually allow the generation of large numbers of programs with much lower resource requirements. Analyzing results from a proportional standpoint thus offers valuable insight into the future potential and scalability of LLM-based fuzzing. That said, in the current landscape, state-of-the-art traditional fuzzers remain a more cost-effective and efficient solution for uncovering code-size bugs.

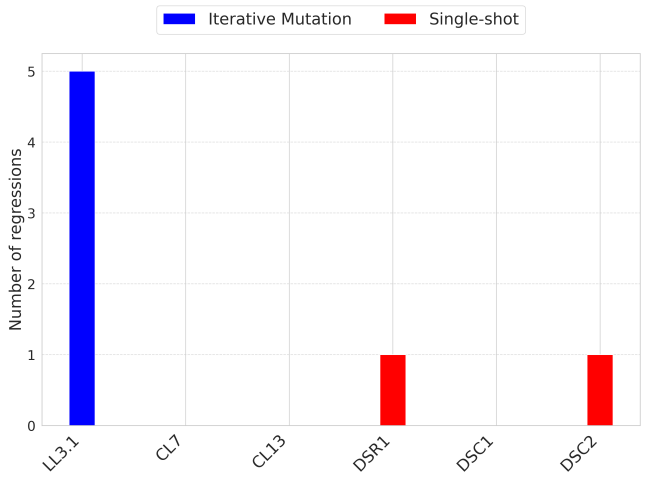

Figure 6. Model total regression comparison.

4.4 RQ4: Iterative Mutation vs One-shot Fuzzing

This section evaluates the benefits of the iterative mutation testing approach by comparing it with a non-iterative test generation methodology, which we call the "*Single-Shot*" approach. In the single-shot setting, the model is queried once per test case, with each test generated independently of any previous ones. This experiment allows us to assess whether Italiano and Cummins's gradual construction of test cases is more effective than simply using the LLM as a memoryless fuzzer. Along the paper,

Discussions: Figure 6 compares the machine learning models evaluated in our study. Our analysis reveals that LLAMA 3.1 8B using iterative mutation achieved superior performance in detecting binary regressions, identifying 5 distinct regression cases. This approach, implementing the methodology from Italiano and Cummins, produces regression rates higher than nearly all other LLMs seen in figure 4, establishing its value for AI-assisted code generation. While figure 4 shows this method still trails traditional fuzzers in overall regression rates, its performance suggests potential for future development.

We observe large variation in effectiveness across models. Whereas LLAMA 3.1 demonstrates strong performance only on iterative mutation, both DEEPSEEKCODER V2 and DEEPSEEK R1 DISTILL LLAMA 3.1 each detected only a single regression in the single-shot experiment. These results indicate model-dependent performance that may evolve with architectural improvements and specialized training.

Specialized vs Non-Specialized Models: A noteworthy finding concerns the performance comparison between code-specialized and general-purpose models. Among the four code-specialized models evaluated (DeepSeekCoder.v2, DeepSeekCoder.v1, CodeLlama 13B Instructe, and CodeLlama 7B) only DeepSeekCoder.v2 detected a regression. In contrast, the general-purpose models Llama3.1 and DeepSeek R1 Distill Llama3.1 achieved measurable results, with both showing higher regression rates in figure 4, when compare the other LLM-based fuzzers. This suggests that code-specific training does not necessarily enhance regression detection capability, challenging conventional assumptions about model code specialization for fuzzing tasks.

Notably, none of the evaluated LLMs received specific training for regression detection. We hypothesize that targeted fine-tuning could significantly enhance both iterative mutation and single-shot approaches, particularly benefiting the more complex iterative method. LLMs specifically trained for regression-aware code completion could potentially achieve substantially better results. However, current limitations in code databases for binary regression induction present challenges, given that this kind of performance bug is not explored in the programming language community. This fact that we could observe encouraging results with untrained models leads us to speculate that the development of specialized datasets containing known regression-inducing code patterns could dramatically improve the performance of models in this application of detecting code-size regressions.

5 Related Work

The present paper reevaluates Italiano and Cummins's work, in terms of throughput, latency and practicality. That work differs from prior fuzzing techniques by systematically combining LLM-driven code generation with differential testing to detect code-size regressions in compilers. Unlike earlier studies, which focus on semantic or correctness bugs, their methodology targets missed optimizations in terms of code size. They also emphasize the adaptability of their approach across multiple programming languages. Nevertheless, fuzzing has a long history within the programming language community, a history that we now revisit.

Compiler Fuzzing. Compilers have been tested using randomized methods for more than 60 years, as surveyed by Chen et al. [2]. One of the earliest examples of fuzzing in this domain dates back to 1962, when Sauder [12] proposed a general test data generator for COBOL compilers. In 1970, Hanford [5] introduced a program generator driven by a PL/1 grammar, which included semantic checks to ensure that variables were declared before use. Shortly after, Purdom [11] presented a syntax-directed method for generating test sentences to verify parsers. Decades later, in 1996, Burgess and Saidi [1] developed an automatic generator of test cases for FORTRAN compilers. This was followed by McKeeman [9], who introduced a family of program generators for the C language that conformed to various levels of the standard. McKeeman's work also coined the term *differential testing*, which has since become a widely used testing technique.

In the 2000s, random program generation became more prominent for testing C compilers. Lindig [7] applied randomized testing to verify C calling conventions. Sheridan

[13] developed a generator that systematically applied operators to constants of different arithmetic types to uncover bugs in C compilers. Zhao et al. [18] created an automated generator specifically for testing an embedded C++ compiler. A major milestone was the release of CSmith by Yang et al. [17] in 2011, which remains one of the most effective and widely used C compiler fuzzers to date. YarpGen [8], one of the fuzzers used in this study, is a direct descendant of CSmith, developed by the same research group.

LLM-Based Fuzzing. More recently, LLM-based compiler fuzzing has emerged as a new line of research. Gu [4], Gao et al. [3], and Ou et al. [10] independently explored the use of large language models to generate programs for testing compilers and language-processing systems. Yang et al. [16] applied LLMs to trigger bugs in compiler optimizations within the PyTorch framework. The portability advantage of LLM-based fuzzing has also been demonstrated. For example, Xia et al. [15] introduced Fuzz4All, a language-agnostic fuzzer powered by LLMs, which successfully found bugs in tools like `gcc`, `clang`, Z3, Cvc5, and OpenJDK. Additionally, Ou et al. [10] showed that LLM-based mutations of randomly generated programs can reveal bugs in both `gcc` and `clang`.

6 Final Remarks

This paper revisited the methodology introduced by Italiano and Cummins [6] for uncovering code-size regressions in compilers using large language models (LLMs). While their approach offers an elegant and language-agnostic framework for compiler fuzzing, our evaluation shows that, in practice, traditional compiler fuzzers like CSmith [17] remain substantially more effective under the constraints of throughput and computational efficiency.

Key Observations. Our experiments demonstrate that CSmith, running on a commodity CPU, produces nearly two orders of magnitude more compilable test programs than state-of-the-art LLMs running on an A100 GPU on the cloud. This higher throughput directly translates to a greater number of code-size regressions identified within a fixed time budget. Although LLaMa3.1 in iterative mutation mode achieved a slightly higher regression density than CSmith, it still lagged behind in absolute counts due to significantly lower generation speed. Our findings also confirm that the gradual mutation strategy proposed by Italiano and Cummins provides meaningful gains over single-shot generation for certain models, particularly for LLaMa3.1. However, this advantage does not consistently generalize to other LLMs.

Threat to Validity: Regressions vs. Bugs. In this work, we measured the ability of fuzzers to detect *code-size regressions*—that is, cases where a compiler, when optimizing for code size, produces a larger binary than when optimizing for performance or applying no optimizations at all. It is important to note that code-size regressions are not necessarily *code-size bugs*. We did not investigate whether these regressions result from poor compiler tuning or design trade-offs, nor did we submit our findings as bug reports to the maintainers of `gcc`, the compiler tested in this study. In contrast, Italiano and Cummins submitted the regressions identified in their study as bug reports to the maintainers of both `clang` and `gcc`. Some of those reports were confirmed as genuine bugs.

Consequently, although our results show that traditional fuzzers detect significantly more code-size regressions than LLM-based fuzzers, it remains possible that many of these regressions stem from a single underlying *performance bug*. It is also possible that some regressions are not bugs at all but rather intentional design decisions made by compiler developers; for example, prioritizing runtime performance over binary size, even in optimization modes intended to reduce code size.

In summary, while our findings demonstrate that traditional fuzzers are more effective than LLM-based fuzzers at uncovering code-size regressions, we cannot conclude that this advantage directly translates to higher effectiveness in finding confirmed bugs.

Future Work. This work opens two promising directions for future research. First, following the discussion in the previous section, it would be valuable to investigate which of the identified code-size regressions correspond to genuine performance bugs. Reporting such bugs would contribute to the `gcc` community and help improve compiler quality.

Second, we have not compared LLM-based and traditional fuzzers in terms of *code diversity*. Code diversity could be assessed behaviorally—for example, through differences in test coverage when evaluating compilers—or syntactically, by analyzing metrics such as the variety of tokens present in each test case. It is possible that LLM-based fuzzers generate more diverse programs than traditional fuzzers. Indeed, one of the bugs reported by Italiano and Cummins was related to the interaction between compiler optimizations and library functions—a scenario that CSmith would be unable to produce, but an LLM-based fuzzer could.

The Promise of LLM-Based Fuzzing. Despite the limitations that we have identified in this paper, the LLM-based approach retains a compelling advantage: its portability. Adapting the methodology to new programming languages requires minimal effort—far less than what would be needed to engineer a new traditional fuzzer from scratch. As LLM technology continues to evolve, improving both throughput and correctness, we expect that the gap between LLM-based and traditional fuzzers will narrow. In the long term, the flexibility of LLM-driven fuzzing could outweigh its current inefficiencies, especially when portability or quick deployment is a priority.

Future work may explore fine-tuning LLMs specifically for compiler fuzzing tasks, developing specialized datasets

of regression-inducing code, and optimizing the inference pipelines to reduce the computational footprint of LLM-based fuzzers. Such improvements could make LLM-driven fuzzing not only competitive but, eventually, preferable for certain classes of compiler testing problems.

Acknowledgment

This project was supported by FAPEMIG (APQ-00440-23) and by Google, through a grant made possible through the intervention of Xinliang David Li and Victor Lee. We thank Davide Italiano and Chris Cummins for discussing this work with us: their comments and suggestions have greatly improved the quality of this manuscript.

References

[1] Colin J Burgess and M Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology* 38, 2 (1996), 111–119.

[2] 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 (Feb. 2020), 36 pages. https://doi.org/10.1145/3363562

[3] Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. 2025. ClozeMaster: Fuzzing Rust Compiler by Harnessing LLMs for Infilling Masked Real Programs. In *ICSE*. IEEE Computer Society, New York, US, 712–712.

[4] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2201–2203. https://doi.org/10.1145/3611643.3617850

[5] Kenneth V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9, 4 (1970), 242–257.

[6] Davide Italiano and Chris Cummins. 2025. Finding Missed Code Size Optimizations in Compilers using Large Language Models. In *International Conference on Compiler Construction* (Las Vegas, NV, USA). Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/3708493.3712686

[7] Christian Lindig. 2005. Random testing of C calling conventions. In *AADEBUG* (Monterey, California, USA). Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1085130.1085132

[8] 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 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

[9] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[10] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2025. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *ASPLOS* (Hilton La Jolla Torrey Pines, La Jolla, CA, USA). Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/3622781.3674171

[11] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12 (1972), 366–375.

[12] Richard L. Sauder. 1962. A general test data generator for COBOL. In *Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE)*. Association for Computing Machinery, New York, NY, USA, 317–323. https://doi.org/10.1145/1460833.1460869

[13] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Softw. Pract. Exper.* 37, 14 (Nov. 2007), 1475–1488.

[14] João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. 2025. Bottom-Up Generation of Verilog Designs for Testing EDA Tools. arXiv:2504.06295 [cs.AR] https://arxiv.org/abs/2504.06295

[15] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. https://doi.org/10.1145/3597503.3639121

[16] 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 (Oct. 2024), 27 pages. https://doi.org/10.1145/3689736

[17] 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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[18] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. 2009. Automated test program generation for an industrial optimizing compiler. In *Workshop on Automation of Software Test*. IEEE, New York, US, 36–43.