Qeios

Peer Review

Review of: "Finding Missed Code Size Optimizations in Compilers using LLMs"

Fernando Magno Quintão Pereira[1]

1. Computer Science, Universidade Federal de Minas Gerais, Brazil

Summary

This paper outlines a methodology for utilizing a large language model (LLM) as a fuzzer to uncover issues related to size-oriented compiler optimizations. It provides an overview of the pipeline used to generate programs, collect feedback from the compiler, and highlights several real bugs discovered through this approach.

Comments

Overall, I thoroughly enjoyed reading this paper. The writing is clear, the organization is logical, and the text flows smoothly from general concepts to specific findings. The idea of leveraging LLMs as effective fuzzing tools is both natural and forward-thinking: it's a direction the community will inevitably embrace. Moreover, having read a lot of papers that use LLMs in various compiler-related tasks, this paper is one of the few that really uses an LLM in a way that does not run into correctness issues. It does not matter if the program that the model generates is incorrect, as long as it is just indicating the possibility of a bug existing in the compiler.

That said, the paper could have gone deeper at some points. While it provides interesting examples of bugs and sheds light on features that might unexpectedly cause code growth, it largely comes across as a catalog of bug reports. Early on, the paper raised several questions that I expected would be addressed but ultimately remained unanswered. Here are some points for consideration:

1. What is the average number of full runs of the pipeline in Figure 2 required to detect a bug in C, Swift, and/or Rust?

2. How many unique bugs were detected per mutation level (e.g., no mutation, one mutation, etc.)?

3. Of all the bugs reported, how many were confirmed by the respective compiler developers?

 doi.org/10.32388/QCCHWP

4. How does the proposed methodology compare to CSmith? For example, replacing the first six steps in Figure 2 with CSmith might yield insights into relative effectiveness. Granted, CSmith does not support C++, Rust, or Swift, but still, it could be compared in an experiment restricted to C.

5. Are the generated programs actually executable? If so, how are inputs provided to them? I am under the impression that programs are executable because the authors control the function signature (e.g., `f()` for C/C++). They compile with a main method that calls it. But that should be clarified.

7. Regarding the fuzzed programs, how many can execute without triggering errors when compiled with AddressSanitizer (ASAN)?

8. The paper suggests detecting dead code by running an instrumented version of the program and testing coverage. However, automatically diffing results is non-trivial. Did the authors develop a specific tool to perform this diffing, or is this an idea that could be implemented in the future?

9. I have some issues with the statement below:

> "The majority of effort has been expended on validating that a compiler produces correct code for a given input, while less attention has been paid to ensuring that the compiler produces performant code."

I think this statement is somewhat misleading. While the intent is clear, the formulation could be improved. In practice, much of the effort in industrial-strength compilers is dedicated to optimizations in the middle end. For example, in LLVM, the middle end (`opt`) comprises over three-quarters of the codebase, with optimizations occupying the majority of the code lines.

Declarations

Potential competing interests: No potential competing interests to declare.