LLM-Based Code Generation Method for Golang Compiler Testing

QIUHAN GU  
 Computer Science and Technology  
 Nanjing University  
 China  
 qiuhan.gu@smail.nju.edu.cn

ABSTRACT

∗Article Title Footnote needs to be captured as Title Note

†Author Footnote to be captured as Author Note

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*WOODSTOCK’18, June, 2018, El Paso, Texas USA*

© 2018 Copyright held by the owner/author(s). 978-1-4503-0000-0/18/06...$15.00

https://doi.org/10.1145/1234567890

Modern optimizing compilers are among the most co-mplex software systems humans build. One way to i-dentify subtle compiler bugs that are hard to find with human-constructed tests is fuzzing. Both the quantity of the testcases and quality of them are crucial to the performance of fuzzing. Traditional fuzzing methods, such as Csmith[[4](#q4)] and YARPGen, have been pro-ven successful at discovering compiler bugs and helping improve the robustness and correctness of compilers. However, these tools require a significant amount of manual effort to write generation rules and templates based on the specifications of the programming language.Their development cycle is long, and the generated testcases have limited coverage.

In this paper, we present a code generation method for go language compiler testing based on LLM. We design a filter strategy targeted at the syntax error and undefined behavior in code to filter the dataset. Based on that, we finetune the open-source pre-trained model CodeT5, making it capable of generating corresponding go language code based on the context of code of the input. Finally,we present a seed selection strategy based on the coverage of the code to improve code generation. After implementing our method on the Golang compiler, we detected 2.79% of syntax errors and 0% of undefined behavior in the testcases generated by the tool. As a result, our pipeline outperforms previous testing methods both qualitatively and quantitatively.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; Source code gen

eration.

KEYWORDS

Large Language Model; Golang; Compiler Testing

ACM Reference format:

FirstName Surname, FirstName Surname and FirstName Surname. 2018. Insert Your Title Here: Insert Subtitle Here. In *Proceedings of ACM Woodstock conference (WOODSTOCK’18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

**1** INTRODUCTION

Compilers are notoriously hard to test, and modern optimizing compilers tend to contain many subtle bugs. Compiler bugs can have serious consequences, including, potentially, the introduction of security vulnerabilities that cannot be detected without knowledge of a compiler flaw [[1](#q1)]. The literature on compiler testing is extensive [[2](#q2)].

As McKeeman’s [[3](#q3)] widely cited paper suggests, one core approach to testing compilers is based on the generation of random programs. Csmith [[4](#q4)] is perhaps the most prominent example of this method. Building a tool such as Csmith is a heroic effort, requiring considerable expertise and development time. However, although Csmith can generate a large number of random testcases, the number of testcases it generates is still limited. Moreover, when using Csmith to generate testcases, it is important to consider the coverage and diversity of the testcases and further manual filtering and supplementation may be necessary to ensure that the tests are effective in detecting potential compiler defects and errors. Also, Csmith is focused on a single, albeit extremely important, language: C. There is no effective tool for generating random Go programs. Go is primarily (or perhaps only) fuzzed at official tool Go Fuzzing (<https://go.dev/security/fuzz/>), which has the same defects with Csmith.

In this paper, we present LLM-based code generation method for Golang compiler testing.

To summarize, our contributions in this work include:

1. A finetuned large language model that generates infinite testcases.
2. A filtered strategy that ensures no undefined behaviours and syntax errors in training data and generated testcases.
3. A coverage-based seed scheduling in fuzzing that maximizes the performance of generated testcases.

2 RELATED WORK AND BACKGROUND

**2.1** Undefined Behavior

A significant practical problem with using random testing to find miscompilation bugs is that Go is unsafe programming languages: when a program executes an erroneous action such as dividing a floating-point number or a complex number by zero

, the Go implementation does not typically flag

the violation by throwing an exception or terminating the program. Rather, the erroneous program

may continue to execute, but with a corrupted memory state. Thus, randomly generated testcases

that are erroneous are useless for differential testing. There are three main kinds of untrapped

errors in Go that are collectively referred to as undefined behaviors (UBs). (1) The initialization order of global variables in packages; (2) The divisor is zero; (3) Multi-threading competition.

In our experiment, we leverages tree-sitter as an auxiliary tool for syntax analysis.

**2.2** Large Language Model

Recently, approaches using large-scale pretrained language models (LMs) have shown promising results in Program synthesis or code generation. Additionally, they can address the challenges of long development cycles, high development complexity, and poor cross-language portability that are associated with traditional compiler development tools.

In our experiment, we choose basic large model codet5 to finetune and generate code.

CodeT5 [[5](#q5)] is a multi-lingual code-aware language model pretrained on large-scale source code corpora curated from Github. With a unifified encoder-decoder architecture, CodeT5 achieves state-of-the-art performance in a wide range of code intelligence tasks in the CodeXGLUE benchmark [[6](#q6)] including both code understanding and generation tasks.

3 CODE GENERATION FOR GO BASED ON LLM

**3.1** Avoiding Undefined Behavior

In this design, we process and filter Go language code files from the internet with the help of the syntax analysis tool tree-sitter (<https://github.com/tree-sitter/tree-sitter>) to obtain Go language code files that meet our requirements.

Firstly, we remove all comments from the code. While comments are an important part of the code, they do not play a role in compiler testing. Then we have designed a series of filtering criteria to remove files that contain content we consider inappropriate:

1. Remove files with syntax errors.
2. Remove files with a character length exceeding 10000.
3. Remove files with duplicate code.
4. Remove files with an English alphanumeric character ratio below 0.25.
5. Remove files that may contain partial undefined behavior.
6. Remove files that reference the "internal" package.

Thus, we prepare the filtered dataset for model training and inial seed choice.

**3.2** Generating Infinite Testcases Automatically

In our experiment, we based our work on the concode task (code generation task) in the CodeT5 model training framework and modified the input and output lengths to match our requirements. Additionally, we made changes to the code responsible for evaluating the model's generation performance, specifically the calculation of perplexity (ppl), to compute it on a batch basis. Furthermore, we developed a module that reads the training data and converts it into tensor format, according to the format we prepared. As a result, we obtained a model capable of automatically generating the body of Go language functions based on the given context of Go language classes provided in the input.By concatenating the input and output, we obtain a complete test case.

By continuously selecting a testcase and randomly removing one function body from it, and input it into the model, we can obtain different representations of the same function body. Therefore, through the iterative process of generating and modifying data, we can obtain an infinite number of testcases.

**3.3** High Coverage of Testcases

When selecting model inputs, we do not choose randomly; instead, we perform seed scheduling based on coverage. This process involves the following steps:

1. Use gotests (<https://github.com/cweill/gotests>) to calculate the coverage of each initial test case for the Golang compiler and sort them based on coverage.(The term "coverage" refers to the measure of how much of the source code of golang compiler is exercised by a given set of test cases.)
2. Select the testcase with the highest coverage, randomly remove a function body from it as the model input, then generate several new testcases by concatenating the input and output.
3. Add the new testcases to the original queue and re-sort them based on coverage.
4. Repeat steps 1-3 in a loop.

**3.4** Experiment Results

After implementing our method on the Golang compiler, we detected 2.79% of syntax errors and 0% of undefined behavior in the testcases generated by the tool, demonstrating our method does particularly well in this aspect.

ACKNOWLEDGMENTS

REFERENCES

1. Scott Bauer, Pascal Cuoq, and ohn Regehr. 2017. POC||GTFO. No Starch Press, Chapter Compiler Bug Backdoors.
2. Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1-36.
3. William McKeeman. 1998. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation* 10(1) (1998), 100-107.
4. 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.* 283-294.
5. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi. Codet5: Identififier-aware unifified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP* (1), pages 8696–8708. A*ssociation for Computational Linguistics*, 2021.
6. S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021.