LLM-Based Code Generation Method for Golang Compiler Testing

Qiuhan Gu*
State Key Laboratory for Novel Software Technology, Nanjing University, China
qiuhan.gu@smail.nju.edu.cn

ABSTRACT

Modern optimizing compilers are among the most complex software systems humans build. One way to identify subtle compiler bugs is fuzzing. Both the quantity and the quality of testcases are crucial to the performance of fuzzing. Traditional testcase-generation methods, such as Csmith and YARPGen, have been proven successful at discovering compiler bugs. However, such generated testcases have limited coverage and quantity. In this paper, we present a code generation method for compiler testing based on LLM to maximize the quality and quantity of the generated code. In particular, to avoid undefined behavior and syntax errors in generated testcases, we design a filter strategy to clean the source code, preparing a high-quality dataset for the model training. Besides, we present a seed schedule strategy to improve code generation. We apply the method to test the Golang compiler and the result shows that our pipeline outperforms previous methods both qualitatively and quantitatively. It produces testcases with an average coverage of 3.38%, in contrast to the testcases generated by GoFuzz, which have an average coverage of 0.44%. Moreover, among all the generated testcases, only 2.79% exhibited syntax errors, and none displayed undefined behavior.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Large model, Code generation, Compiler testing, Go language

ACM Reference Format:
Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3611643.3617850

1 INTRODUCTION

Compilers are notoriously hard to test, and modern optimizing compilers tend to contain many subtle bugs, leading to, potentially, the introduction of security vulnerabilities that cannot be detected without knowledge of a compiler flaw [14]. The literature on compiler testing is extensive [9]. One core approach to testing compilers is based on the generation of random programs [15]. Csmith 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.

The number of testcases generated by random program generation methods is still limited. Moreover, when using methods like Csmith to generate testcases, it is important to consider the coverage and diversity of the testcases, and further manual filtering and supplementation are necessary to ensure tests are all effective. Also, Csmith is focused on a single, although extremely important, language: C. There is no effective tool for efficiently generating Go programs. The language Go, albeit not widely used in the research community, has significant applications in many domains, such as high-performance server development and microservices architecture. Go is primarily fuzzed at the official tool Go Fuzzing [3], which has the same defects as Csmith. As a result, we turn to a large-scale model to address the issue of universality. However, although deep learning is an effective way of fuzzing, experience shows that the effectiveness of testcases generated by large-scale model-based fuzzing is limited in practical applications [11].

In this paper, we present an LLM-based high-quality code generation method for testing the Golang compiler. To summarize, our contributions to this work include:

(1) A LLM-based high-quality code generation method.
(2) We employed the method on the Golang compiler, producing testcases that achieved an average coverage of 3.38%. Among these testcases, only 2.79% exhibited syntax errors, and none manifested undefined behavior.

2 RELATED WORK AND BACKGROUND

2.1 Undefined Behavior

A significant practical problem with using random testing to find miscompilation bugs is that programming languages are unsafe. For example, when a Go-code program executes an erroneous action such as dividing a floating-point number, 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. There are three main kinds of untrapped errors in Go that are referred to as undefined behaviors (UBs) [5]: (1) The initialization order of global variables in packages; (2) The divisor is zero; (3) Race condition.

2.2 Large Language Model

Recently, approaches using large-scale pretrained language models (LMs) have shown promising results in Program synthesis or code

*Advisors: Yu WANG

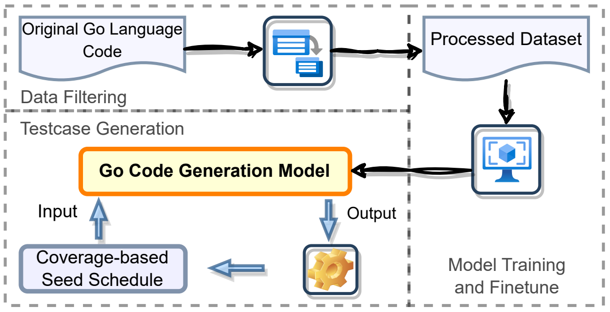

Figure 1: The whole workflow of our experiment. This workflow consists of three parts: (1) data filtering, which involves removing undefined behaviors and syntax errors from the dataset; (2) model training and fine-tuning; (3) a loop including seed schedule, generating a great number of testcases.

generation [8, 10, 12, 16–19], such as GPT-3 [16], which is a massive pretrained language model with 175 billion parameters based on the Transformer architecture. 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. However, they are also difficult to deploy on regular hardware devices due to high parameter count and large size.

CodeT5 [18] is a multi-lingual code-aware language model pre-trained on large-scale source code corpora curated from Github. With a unified encoder-decoder architecture, CodeT5 achieves state-of-the-art performance in a wide range of code intelligence tasks in the CodeXGLUE benchmark [13].

3 CODE GENERATION FOR GO BASED ON LLM

3.1 Training Data Processing

In this section, we process Go language code files with the help of the syntax analysis tool tree-sitter [1] to obtain Go language code files that meet our requirements.

Firstly, we remove all comments from the code. Then we designed a series of filtering criteria to remove low-quality programs:

(1) Remove files with syntax errors.
(2) Remove files with a character length exceeding 10000.
(3) Remove files with duplicate code[7].
(4) Remove files with an alphanumeric characters ratio below 0.25.
(5) Remove files that may contain undefined behavior.
(6) Remove files that reference the "internal" package.

In the end, we prepare the filtered dataset containing 1839 programs for model training and initial seed choice.

3.2 Maximizing Coverage of Testcases

Many Fuzzing strategies always base on known error reports, vulnerability records, or the boundary conditions of input to choose seed samples. Go Fuzz selects seeds to lead to the execution of more code paths, thereby increasing the probability of discovering potential issues. However, they are not suitable for our purpose, which is to test the compiler not executable code with parameters, it is important to consider the code coverage of the compiler itself rather than the code coverage of testcases.

In this paper, we propose a code coverage-based seed schedule strategy for the compiler, which includes the following steps:

(1) Use gotests [2] to calculate the coverage of each initial test-case for the Golang compiler and sort them based on coverage. ("Coverage" refers to the measure of how much of the source code of the Golang compiler is exercised by the given testcases.)
(2) Select the testcase with the highest coverage as the seed, randomly remove a function body from it as the input, and then generate several new testcases by combing the input and output.
(3) Add the new testcases to the original queue and resort them based on coverage.
(4) Repeat steps 1-3 until the coverage reaches a stable state.

3.3 Generating Testcases Automatically

We can generate tastcases automatically and infinitely. This is another main difference between our method and other common methods like GoFuzz, Csmith, and YARPGen.

In our experiment, we base our work on the concode task (code generation task) in the CodeT5 model and improve the performance and prediction accuracy of the model through feedback training. Finally, we obtain 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 testcase. Finally, we omit uncompilable testcases. By continuously selecting a testcase, randomly removing one function body from it, and transferring it into the model, we can obtain different implementations of the same function body, thus obtain an infinite number of testcases. The whole procedure is shown in Figure 1.

3.4 Experiment Results

In our experiment, we finetuned the CodeT5 through 30 epochs, at the learning rate of 10^{-5} and warmup steps of 1000. Then we implemented our method on the Golang compiler, looping 1000 times to generate 1157 testcases (cost about 100h).

We detected only 2.79% of syntax errors and 0% of undefined behavior in testcases our tool generated. The average coverage of the generated code is 3.38%, compared to testcases with 0.44% average coverage generated by Go Fuzzing [6], demonstrating our method does particularly well in this aspect. Our dataset and tool are publicly available at [4].

4 CONCLUSION

We present an LLM-based code generation method to test modern compilers efficiently. With the help of a large language model, we achieve the universality and portability of the tool, and increase the quantity of generated testcases, significantly reducing manual effort. We present a code-filtering strategy to avoid syntax errors and undefined behavior in generated testcases. Finally, we use a coverage-based seed schedule to improve the performance of testcases. As a result, our method outperforms previous testing methods both qualitatively and quantitatively.

5 ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 62232001 and No. 62202220.

REFERENCES

[1] 2013. Tree-sitter. https://github.com/tree-sitter/tree-sitter.

[2] 2016. Gotests. https://github.com/cweill/gotests.

[3] 2022. Go.dev - Fuzzing. https://go.dev/security/fuzz/.

[4] 2023. Experiment code: LLM-Based-Code-Generation-Method-for-Golang-Compiler-Testing, including our dataset, finutuned model and scripts. https://github.com/GuQiuhan/LLM-Based-Code-Generation-Method-for-Golang-Compiler-Testing.

[5] 2023. Go. https://go.dev/ref/spec.

[6] 2023. GoFuzzUrl. https://github.com/dvyukov/go-fuzz.

[7] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. arXiv:1812.06469 [cs.SE]

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations*. https://arxiv.org/abs/1611.01989

[9] 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. https://doi.org/10.1145/1234567

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Improving Language Understanding by Generative Pre-training. *arXiv preprint arXiv:1810.04805* (2018). https://arxiv.org/abs/1810.04805

[11] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. The Perils and Pitfalls of Mining GitHub. *Proceedings of the 13th International Conference on Mining Software Repositories* (2016). https://doi.org/10.1145/2934466

[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020). https://arxiv.org/abs/2001.08361

[13] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[14] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? OOPSLA (2019). https://doi.org/10.1145/3360581

[15] William McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal of Digital Equipment Corporation* 10, 1 (1998), 100–107.

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[17] Richard Shin, Dawn Song, and Xinyun Chen. 2020. AlphaCoder: Teaching an AI to Write Programs. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2005.03706

[18] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint arXiv:2109.00859* (2021). https://doi.org/10.48550/arXiv.2109.00859 arXiv:2109.00859 [cs.CL] Accepted to EMNLP 2021. 13 pages.

[19] Pengcheng Yin, Graham Neubig, and Prasanna Parthasarathi. 2021. CodeT5: Synthesizing Robust Testers for Language Primitives from Semi-Structured Natural Language Instructions. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*. 1843–1857.

Received 2023-06-05; accepted 2023-08-11