Differential Fuzzing Go Compilers using LLMs: A Methodological Proposal

Luiz Paulo Grafetti Terres[1], **Samuel da Silva Feitosa**[1]

[1] Universidade Federal da Fronteira Sul
Campus Chapecó

luiz.terres@estudante.uffs.edu.br, samuel.feitosa@uffs.edu.br

Abstract. *Compilers are essential tools for software development. Ensuring their reliability is vital for the security of the software ecosystem. Traditional compiler testing exposes them to many generated programs, but conventional code generators struggle with its own complex implementation and effective test case generation. Recent advancements in Large Language Models (LLMs) and their code-related proficiency present an opportunity to address these challenges. This work proposes a methodology for the differential fuzzing of Go compilers that leverages LLMs as test case generators. The proposed method employs a cross-compiler differential testing strategy to test three compilers: GOLLVM, GCCGO, and the official Go compiler.*

1. Introduction

Compilers are fundamental software for building software. They are designed to allow developers to build hardware-agnostic applications, by translating code in programming languages into machine code to be executed by the user's hardware [Aho et al. 2006]. Modern compilers are highly complex software, conceptually split into a series of steps that compose their analysis and synthesis part [Aho et al. 2006]. Due to their complexity, full compilers usually are not formally verified [Leroy et al. 2016]. As a result, they remain susceptible to bugs and may produce compilation results that deviate from the target programming language specification because of faulty implementations, causing security concerns to all ecosystem of software development [Bauer et al. 2015]. For that reason, most compilers depend on robust testing techniques to detect implementation faults before they can be exploited.

Random testing (also addressed as fuzzing) emerges as a viable and most common approach for testing compilers [Chen et al. 2020, Tang et al. 2020]. The intuition behind fuzzing is to feed the compiler under test with a large number of inputs and analyze each output [Manes et al. 2019]. Previous research address many challenges on the process of testing compilers (further discussed in Section 2.1), among them, the challenge of generating adequate test cases. Random strings are examples of inadequate test cases, as it usually causes the compiler to execute only a shallow part of it's logic [McKeeman 1998]. Traditional approaches often use crafted tools for generating test cases, adhering to strategies of either code generation or seed mutation [Manes et al. 2019]. The literature highlights those tool require expertise to develop, are costly to implement and maintain [Chen et al. 2020] and also lack generalization [Xia et al. 2024].

Language models are fundamental software for the field of natural language processing. These models leverage mathematical methods for predicting linguistic units, thereby simulating aspects of human language [Wang et al. 2024b]. In recent years, Large Language Models (LLMs) set a new state-of-the-art for the field, employing the Transformer architecture, proposed by [Vaswani et al. 2017]. Large models are trained on large text corpora, from diverse sources and domains of knowledge, and exhibit remarkable capabilities in retrieving this information during inference [Wang et al. 2024b]. They have been employed to diverse code-related tasks, including code completion, summarization and generation among different programming languages [Jiang et al. 2024]. Given those recent advancements in Large Language Models (LLMs) and their proficiency in code-related tasks, this work explores their application to address the challenges of compiler testing.

We present a methodological proposal for the differential fuzzing of Go compilers, leveraging LLMs as test case generators. This proposal is informed by the results of a (not yet published) Systematic Literature Review conducted by the authors on the use of LLMs in compiler testing. The remainder of this paper is structured as follows: Section 2 provides the necessary background on compiler testing and LLMs. Section 3 presents the methodology of the SLR conducted by the authors and partially discusses its result. Section 4 details our proposed methodology for differential fuzzing. Finally, Section 5 discusses the proposal and future work, followed by the conclusion in Section 6.

2. Background

This section presents an overview of content related to our research. Subsection (2.1) covers what are compilers and how they have been tested by the literature and (2.2) what are LLMs and how they have been employed for software engineering tasks.

2.1. Compilers and compiler testing

Compilers represent a class of programs that are usually associated with translating high-level representation into lower-level ones. Traditional compilers are primarily designed to translate programming language source code into equivalent executable machine code [Aho et al. 2006]. They represent a central software for the software development industry, and beyond their direct use from technical developers, casual users execute compiled programs on their daily life [Chen et al. 2020]. Modern compilers go beyond a simple translation operation. Figure 1 overviews a simplified workflow of the compilation process: the input source code is analyzed by the front end, which produces an Intermediary Representation (IR) in which optimization passes are performed and the result is translated into machine code for the targeted platform [Aho et al. 2006]. Most compilers provide various options (compilation flags, Figure 1 ②) for controlling the occurrence of compilation operations such as optimization levels.

Being software themselves, compilers are not free from bugs [Sun et al. 2016]. Fuzzing is an automated software testing technique proposed by [Miller et al. 1990] for testing the reliability of UNIX systems. According to [Manes et al. 2019] definition, fuzzing is the execution of the Program Under Test (PUT) using inputs sampled from an input space that protrudes the expected input space of the PUT. This means even malformed programs or random strings count as valid test cases for fuzzing compilers. Those

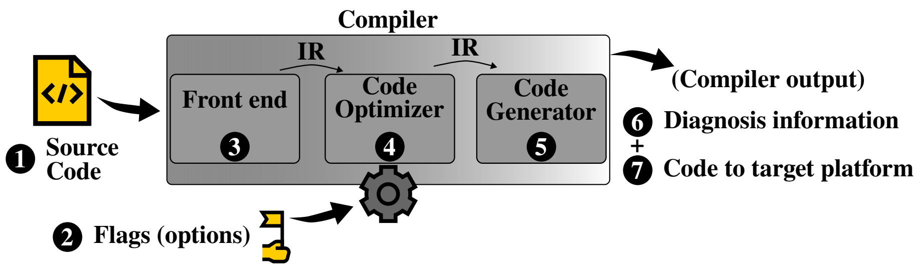


Figure 1. Simplified workflow of the compilation process.

inputs could in fact trigger bugs in the compiler's front end, however would soon fall short in reaching deeper stages on the compilation process (Figure 1) missing the execution of those portions of the compiler [McKeeman 1998]. In fact, the generation of adequate test cases is a challenge addressed by previous research that concerns the test suite generation (Figure 2, ①) step on the compiler testing process. Figure 2 overviews the main steps in compiler fuzzing.

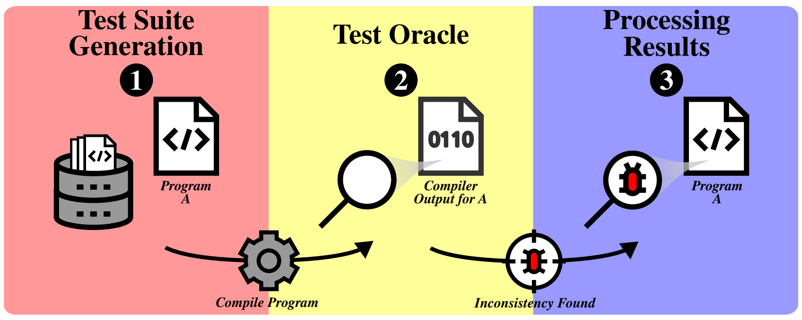


Figure 2. The main steps in compiler testing.

As shown by Figure 2, compiler testing can be divided into 3 main stages. To the Test Suite Generation step ①, concerns the generation of good inputs for the compiler to process. Although programs are their primary input, compilation flags (Figure 1, ②) enable fuzzing of compilers' configuration space, often referred as option fuzzing [Zhang et al. 2023]. Test Oracle ② step aims to identify which test cases provoke bugs or miscompilations by analyzing the compiler's output. The Processing Results step ③ refers to refining bug-triggering test cases into minimal programs that can be reported to compiler developers. Note that Figure 2 does not depict the workflow of the compiler testing process, instead it highlights key steps. During a compiler fuzzing campaign, often the steps ① and ② are executed one after another in a loop: a program is generated, compiled and evaluated by the oracle (then repeat). Step ③ is often performed after the fuzzing campaign, manipulating potential bug-triggering test cases selected by the test oracle ②.

The following list presents some challenges addressed by previous research related to the key steps for compiler testing:

- **Adequate test cases ①**: Complexity of tools for generating valid test cases; test cases should strive to be diverse and meet certain requirements for testing; different test cases finding the same bug hurts efficiency [Chen et al. 2020].
- **Test oracle problem ②**: Compiler testing lacks test oracles to determine semantic equivalence between a test case and its compiled output. [Tang et al. 2020].

- **Undefined behavior ③**: Test cases containing undefined behavior often causes the test oracle to select false positives, negatively affecting tasks such as bug isolation on the processing results step [Tang et al. 2020].

The literature presents effort to address the highlighted challenges. For addressing adequate test case generation, many compiler testing solutions employ tools for generating syntactically valid random programs [Boujarwah and Saleh 1997, Aschermann et al. 2019]. Instrumenting compilers for gathering compilation metrics categorizes approaches in white-box (the most information-guided), grey-box and black-box fuzzing (not guided by internal compiler information) [Manes et al. 2019]. In grey-box approaches, metrics such as code coverage attempts to measure the effectiveness of test cases, by quantifying the portions of the compiler a test case triggers execution [Herrera et al. 2021]. Differential testing proposed by [McKeeman 1998] provides partial solution for the test oracle problem. It consists in comparing the outputs of comparable systems given the same input. If the outputs differ, or if one system presents inconsistent behavior (loop or crash), the input is a possibly bug-exposing test case. Table 1 presents existing strategies for comparing results in the context of compiler testing:

Table 1. Common strategies for differential testing compilers. [Chen et al. 2020]

Cross-compiler	**Cross-optimization**	**Cross-version**
Compares results produced by different compilers of a same programming language.	Compares results produced by the same compilers under different options (levels) of optimization.	Compares results produced by different versions of a compiler.

After the selection of possibly bug-provoking test cases by the test oracle, often those test cases need to be further refined before they are issued to compiler developers [Tang et al. 2020]. For files containing bug provoking code, techniques for program reduction and duplicated bug identification are fundamental to identify and fix those bugs [Chen et al. 2020].

2.2. Large language models

Large Language Models (LLMs) constitute the state of the art of language models, which aim to predict language units (e.g., words) to a given context using mathematical methods, at the core of the natural language processing field [Wang et al. 2024b]. According to [Wang et al. 2024b], LLMs represent a transition from specialized to more general-purpose models (capable of performing tasks among various domains), attributing this to the data diversity they are trained on, computational advancements and algorithmic innovation brought by the Transformer architecture [Vaswani et al. 2017]. Training is a central step for those models, in which they process large datasets, for adjusting tens of billions of parameters and undergo supervised finetuning, to enhance their task-specific capabilities (e.g. following instructions)[Liu et al. 2025]. In the software engineering domain, including code-related tasks, LLMs have been widely employed, from code understanding to code generation [Jiang et al. 2024].

A remarkable feature of LLMs is their ability to adapt to new tasks through processes such as fine-tuning and prompting [Liu et al. 2023]. Fine-tuning involves adjusting the model's weights with additional data, enabling task specialization of the model

[Liu et al. 2024]. Prompting, in contrast, requires no parameter updates. It relies on natural language inputs to guide the model's behavior. Prompting is a key practice for extracting the capabilities of the LLMs [Liu et al. 2023], that range from feeding the LLM with examples to guide expected responses, up to combining whole external knowledge bases, in advanced techniques [Liu et al. 2023].

3. Related work

Are among the survey literature reviewed by the authors works on compiler testing that do not address the use of LLMs [Chen et al. 2020, Tang et al. 2020] and works that explore the employment LLMs for software testing, but not focused on compiler testing [Wang et al. 2024a, Jiang et al. 2024]. To the best of our knowledge, no systematic survey has been conducted specifically to investigate compiler testing employing large language models. Noticing this gap in the literature, the authors conducted a Systematic Literature Review (SLR) on the topic. This section presents the methodology for the SRL and summarizes 3 handpicked works from its selected works. Those works present tools conceptually close to our methodology for testing Go compilers, in which LLMs are employed as test case (programs) generators.

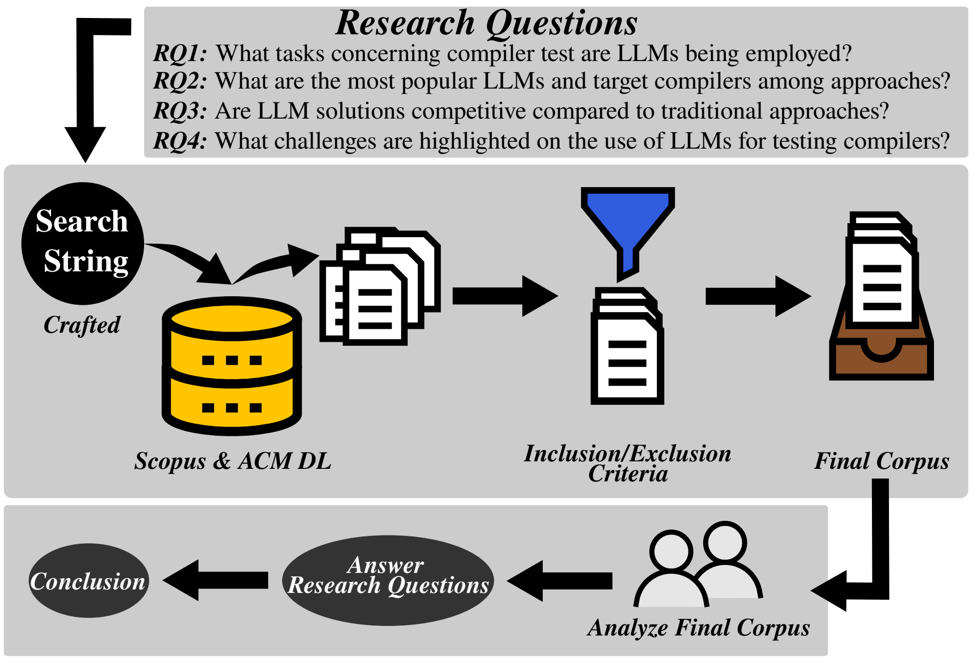

Research Questions
RQ1: What tasks concerning compiler test are LLMs being employed?
RQ2: What are the most popular LLMs and target compilers among approaches?
RQ3: Are LLM solutions competitive compared to traditional approaches?
RQ4: What challenges are highlighted on the use of LLMs for testing compilers?
Search String
Crafted
Scopus & ACM DL
Inclusion/Exclusion Criteria
Final Corpus
Analyze Final Corpus
Answer Research Questions
Conclusion

Figure 3. Workflow of the systematic literature review developed by the authors.

Figure 3 illustrates the workflow of the SLR conducted by the authors and the research questions it addresses. The search for relevant works was conducted in Scopus and ACM Digital Library, using a search string of relevant keywords crafted by the authors. We searched the full-text papers in both databases for the terms "large language model" OR "llm" and their plural form, AND the keywords "fuzz", "fuzzer", "fuzzing", "test", "tester" and "testing" within 5 words of the keyword "compiler". Papers retrieved were then filtered using explicit inclusion and exclusion criteria. Were selected works that meet all criteria: primary research; published after 2019; states focus on compiler testing or bug handling; employ LLM for a task in compiler testing process; written in english or portuguese. This process gave us a final corpus of interesting papers, that were carefully analyzed by the authors to respond to the proposed research questions.

Among the works selected by the SRL were tools compiler testing and bug handling. The following list briefly presents 3 handpicked selected works with similar purpose to our proposed tool. Those tools focus in testing compilers employing large language models as test case generator, addressing the challenges on traditional approaches.

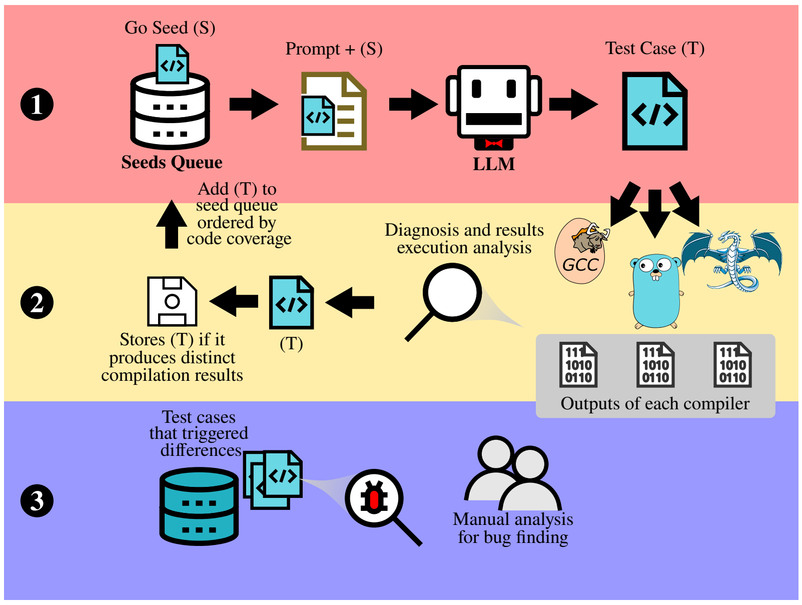
- [Xia et al. 2024] **Fuzz4All**: An universal fuzzer for all systems that take in programming or formal languages as input (e.g. compilers), addressing the lack of generality challenge in traditional code generators (Section 2.1) by combining LLM agents. A *distillation* LLM is prompted with information about the system under test and its input format. This *distillation* LLM process the initial prompt and instruct a second *generation* LLM to be a test case generator.
- [Yang et al. 2024] **RustTwins**: A differential fuzzer for the rust compiler `rustc`. Initially the LLM is instructed to mutate a program from a seed pool, using 18 crafting prompts. This mutated seed is then a baseline for generating 2 semantic-equivalent macros using the LLM. The LLM invokes those semantic-equivalent macros that should give allways the same output. Differences in their compilation reveal bugs. This work explores adequate test cases and test oracle challenges by leveraging semantic capabilities of LLMs.
- [Eom et al. 2024] **CovRL-Fuzz**: A fuzzer for JavaScript engines. A LLM mutates masked seeds from a seed queue. Those mutated seeds are executed by JS engines. Coverage information about interesting mutated seeds is captured and is used (via TF-IDF technique) to finetune the LLM model via reinforcement learning. In this approach, the model is constantly being finetuned to perform better seed mutations. This work explores LLMs to overcome challenges of adequate test cases.

Inspired by the effectiveness of selected works on addressing the challenges associated with compiler testing techniques, this work proposes a methodology for fuzzing Go compilers in the next section.

4. Methodology Proposal

This section covers in high level a methodology for fuzzing Go compilers using large language models as test case generator. This section also strongly relates to the next section (Section 5), which discusses keypoints and future improvements of this approach. Our tool consists in a differential fuzzer for Go compilers in a cross-compiler strategy (Table 1) employing large language models as test case generators using one-shot prompting, as depicted by Figure 4.

The testing campaign consists in looping the test case generation ① and test oracle steps ②. After the results of the campaign are collected, the stored possibly bug-triggering programs are analyzed manually ③. In the test suite generation step ① a LLM is guided to generate test cases via one-shot prompting using examples from a queue of interesting programs. This queue initially contains valid programs from previous research on test suite for testing the Go compiler [Gu 2023]. LLM-generated test cases are then compiled by 3 different Go compilers: GOLLVM, GCCGo and the official Go compiler. The official Go compiler is instrumented for yielding the code coverage achieved by the test case



1
Go Seed (S)
Seeds Queue
Prompt + (S)
LLM
Test Case (T)
2
Add (T) to seed queue ordered by code coverage
Diagnosis and results execution analysis
GCC
Stores (T) if it produces distinct compilation results
(T)
Outputs of each compiler
3
Test cases that triggered differences
Manual analysis for bug finding

Figure 4. Simplified workflow of the proposed tool for compiler testing.

processed. The generated program enters the queue of examples if it presents good code coverage compared to programs already in the queue. After the compilers generate their diagnosis and (optionally) their program files as a result start the test oracle step ②. The program file is executed for observable outputs (execution time must be limited, in case the program contains an infinite loop). Diagnosis and observable outputs are then compared. If differences are found, the generated program is stored for manual analysis. In case of no difference, the LLM prompted to generate the next program. Finally ③, all test cases producing divergent compilation outcomes are manually inspected by the authors to identify the underlying causes of potential bugs.

5. Discussion and Future Work

This section discusses the methodology proposal of the previous section and point directions to refine it.

Our methodology proposal does not discuss the large language model used to generate test programs. The fundamental reason is that the model's choice depends on the hardware our tool is being executed on. The cost of inferring LLMs is widely acknowledged by the literature [Jiang et al. 2024, Wang et al. 2024b, Liu et al. 2024] and may result in fewer test cases generated when compared to traditional code generators and mutators, as observed by similar works [Xia et al. 2024, Eom et al. 2024]. Solutions proposed to address this efficiency problem often remove the LLM from the loop (generate - execute) in the fuzzing campaign. [Ou et al. 2025] combine LLM and traditional approaches by using a LLM to generate traditional code mutators through natural language description of mutation operations. Similarly, [Ni and Li 2025] divides the the test case generation in offline and online stages, the offline stage being the one occurring during the fuzzing loop, combining code snippets generated beforehand by the online stage.

Another keypoint on our proposal is the lack of a defined prompt structure. Existing works in the literature explore prompting techniques such as template, RAG and one-shot for compiler test suites [Munley et al. 2024]. Our solution mentions the use of one-shot prompting, but this aspect of our methodology is not rigid.

As future work, our methodology could expand the employment of LLMs to beyond just generating test cases. The model could be more aligned with the test oracle, generating inputs to feed compilation results as in [Yang et al. 2024]. Another possibility is to employ the LLM for a task inside the processing results step, where our proposal depends on manual analysis. In regard to the differential testing aspect of our tool, it could expand to support cross-optimization or cross-version strategies. On the generation of test cases, while the official Go compiler supports the latest version of the programming language, that's not the case for the other compilers. At the time of writing this work, GCCGo and GOLLVM share the same front end called `gofrontend`, currently supporting the 1.18 version of Golang. A possible refinement to our solution could be including additional compiler diagnostic information when prompting the model.

6. Conclusion

This work presented a methodological proposal for differential fuzzing of Go compilers using large language models as test case generators. Our proposal is motivated by similar works in the literature that address challenges associated with traditional techniques for compiler testing. While language model choice and prompt design remain open issues, we expect this methodology to serve as a foundation for future implementations and evaluations in Go compiler testing solutions.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. (2019). Nautilus: Fishing for deep bugs with grammars.

Bauer, S., Cuoq, P., and Regehr, J. (2015). Deniable backdoors using compiler bugs. *International Journal of PoC—— GTFO, 0x08*, pages 7–9.

Boujarwah, A. and Saleh, K. (1997). Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625.

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., and Zhang, L. (2020). A survey of compiler testing. *ACM Comput. Surv.*, 53(1).

Eom, J., Jeong, S., and Kwon, T. (2024). Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1656–1668, New York, NY, USA. Association for Computing Machinery.

Gu, Q. (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 2023, page 2201–2203, New York, NY, USA. Association for Computing Machinery.

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., and Hosking, A. L. (2021). Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT In-*

ternational Symposium on Software Testing and Analysis, ISSTA 2021, page 230–243, New York, NY, USA. Association for Computing Machinery.

Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. (2024). A survey on large language models for code generation.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016). CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France. SEE.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9).

Liu, Y., He, H., Han, T., Zhang, X., Liu, M., Tian, J., Zhang, Y., Wang, J., Gao, X., Zhong, T., Pan, Y., Xu, S., Wu, Z., Liu, Z., Zhang, X., Zhang, S., Hu, X., Zhang, T., Qiang, N., Liu, T., and Ge, B. (2024). Understanding llms: A comprehensive overview from training to inference.

Liu, Y., He, H., Han, T., Zhang, X., Liu, M., Tian, J., Zhang, Y., Wang, J., Gao, X., Zhong, T., Pan, Y., Xu, S., Wu, Z., Liu, Z., Zhang, X., Zhang, S., Hu, X., Zhang, T., Qiang, N., Liu, T., and Ge, B. (2025). Understanding llms: A comprehensive overview from training to inference. *Neurocomputing*, 620:129190.

Manes, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2019). The art, science, and engineering of fuzzing: A survey.

McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal*, 10(1):100–107.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44.

Munley, C., Jarmusch, A., and Chandrasekaran, S. (2024). Llm4vv: Developing llm-driven testsuite for compiler validation. *Future Generation Computer Systems*, 160:1–13.

Ni, Y. and Li, S. (2025). Interleaving large language models for compiler testing.

Ou, X., Li, C., Jiang, Y., and Xu, C. (2025). The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '24, page 298–312, New York, NY, USA. Association for Computing Machinery.

Sun, C., Le, V., Zhang, Q., and Su, Z. (2016). Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA. Association for Computing Machinery.

Tang, Y., Ren, Z., Kong, W., and Jiang, H. (2020). Compiler testing: a systematic literature analysis. *Frontiers of Computer Science*, 14(1):1–20.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg,

U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024a). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wang, Z., Chu, Z., Doan, T. V., Ni, S., Yang, M., and Zhang, W. (2024b). History, development, and principles of large language models-an introductory survey.

Xia, C. S., Paltenghi, M., Le Tian, J., Pradel, M., and Zhang, L. (2024). Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Yang, W., Gao, C., Liu, X., Li, Y., and Xue, Y. (2024). Rust-twins: Automatic rust compiler testing through program mutation and dual macros generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 631–642, New York, NY, USA. Association for Computing Machinery.

Zhang, Z., Klees, G., Wang, E., Hicks, M., and Wei, S. (2023). Fuzzing configurations of program options. *ACM Trans. Softw. Eng. Methodol.*, 32(2).