

MODELOS DE LINGUAGEM DE GRANDE ESCALA APLICADOS À GERAÇÃO DE CÓDIGO

LUIZ PAULO GRAFETTI TERRES^{1,2*}, SAMUEL DA SILVA FEITOSA^{2,3}

1 Introdução

Na interação do usuário com os componentes do computador, estão envolvidas Linguagens Formais (LFs) de diferentes níveis de abstração. Essas linguagens são usadas para instruir computadores a executar uma sequência de ações (instruções), a fim de produzir um resultado esperado. Por exemplo, por trás do documento desse resumo expandido, estão LFs de estilização, que definem a formatação do documento final. Na computação, o uso de LFs, de forma prática, depende de programas interpretadores e tradutores (compiladores) dessas linguagens. Esses programas são importantes porque permitem a manipulação de diferentes linguagens e níveis de abstração, mantendo o valor semântico das instruções.

No desenvolvimento de programas, desenvolvedores escrevem o código fonte e utilizam programas compiladores para traduzir da LF de programação (ou, simplesmente linguagem de programação) para código binário (programa executável). A centralidade dos compiladores no desenvolvimento de todo o ecossistema digital torna imprescindível garantir o funcionamento seguro desses programas (Thompson, 1984). A abordagem mais tradicional para isso é o teste dinâmico, em que compiladores processam códigos fonte aleatórios e os códigos binários resultantes são analisados, em busca de incongruências (Chen, 2018).

No teste de compiladores, a tarefa de gerar códigos fonte aleatórios possui desafios como a complexidade de implementação de programas geradores e a eficiência de diferentes abordagens (Jiang, 2024). Novas abordagens aplicam Modelos de Linguagem de Grande Escala (MLGE) na geração de códigos fonte aleatórios e mitigam a complexidade de implementação de geradores, porém apresentam um alto custo computacional e de eficiência (Xia, 2024). Nesse sentido, este trabalho busca explorar o uso eficiente de MLGEs para a geração de código para testes de compiladores como geradores de programas mutadores de código.

¹ Graduado em Ciência da Computação, Universidade Federal da Fronteira Sul, campus Chapecó, contato: luiz.terres@estudante.uffs.edu.br

² Grupo de Pesquisa: IDT - Inovação e Desenvolvimento Tecnológico.

³ Doutor em Ciência da Computação, Universidade Federal da Fronteira Sul, Orientador.

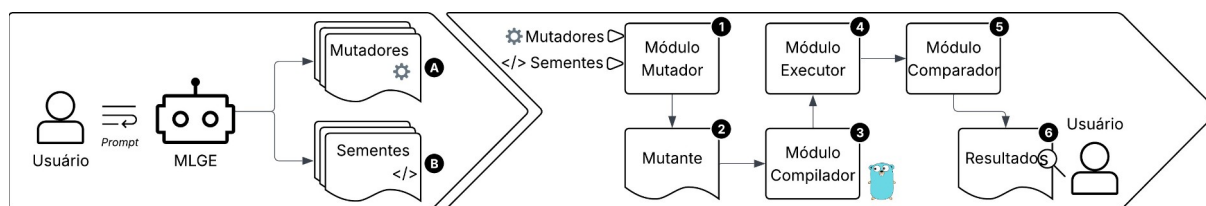
2 Objetivos

- Requisitar, a um modelo de linguagem de grande escala, a implementação de mutadores de código para a linguagem de programação Go.
- Implementar uma ferramenta para teste dinâmico de compiladores da linguagem Go que utilize os mutadores de código para geração de programas aleatórios.
- Testar o compilador de Go usando a ferramenta implementada.
- Analisar resultados, buscando falhas do compilador de Go e verificando a viabilidade do uso de MLGEs como geradores de mutadores de código aleatório.

3 Metodologia

A Figura 1 auxilia a apresentação da metodologia deste trabalho, que pode ser dividida em duas etapas. Na primeira etapa, o usuário requisita ao Modelo de Linguagem de Grande Escala (Google Gemini 3.1 Pro) a sintetização de Mutadores (Figura 1 (A). São programas que alteram o código fonte de outros programas) e Sementes (Figura 1 (B). São programas base aleatórios, sob quais posteriormente serão aplicados os Mutadores). Na requisição, o modelo Gemini Pro é informado sobre o objetivo da tarefa e porções de código do compilador de Go responsáveis por aplicar otimizações aos programas compilados (Google, 2025). Dessa forma, a solução proposta nesse trabalho busca estressar a execução da porção do compilador responsável pela aplicação de otimizações no código compilado.

Figura 1: Fluxo da metodologia do trabalho



Fonte: Autores, 2026

A segunda etapa da metodologia do trabalho consiste na implementação da ferramenta de teste do compilador de Go e execução do experimento para análise posterior dos resultados. A ferramenta implementada consiste em quatro módulos Figura 1 (1,3,4,5) encadeados. No experimento, os módulos encadeados da ferramenta são executados em ciclo (várias iterações).

O Módulo Mutador Figura 1 (1) é responsável por aplicar os Mutadores Figura 1 (A) em Sementes Figura 1 (B). Esse módulo cria, a cada iteração do experimento, um Mutante Figura 1 (2), que consiste em uma Semente alterada. Ainda nesse módulo, a cobertura de código

executada do compilador é capturada. Essa métrica indica quantas linhas da implementação do compilador é executada (testada) durante a compilação do novo Mutante. O Mutante criado em uma iteração torna-se Semente para iterações posteriores. Inicialmente, na execução do experimento, todos os Mutadores são aplicados a todas as Sementes geradas pelo MLGE. Em iterações posteriores, todos os Mutadores são aplicados a novas Sementes (geradas em iterações anteriores) que, indicadas pela métrica de cobertura de código, exploraram novas linhas na implementação do compilador.

No Módulo Compilador Figura 1 (3), o código Mutante gerado a cada iteração, é compilado por diferentes versões do compilador de Go. Para o experimento, foram testados quatro compiladores: Go 1.25.10; Go 1.25.10 sem otimizações; Go 1.26.3 e Go 1.26.3 sem otimizações. A captura dos resultados dos compiladores de Go em diferentes versões e diferentes opções de otimização possibilita a tarefa posterior de análise dos resultados usando comparação direta entre as saídas obtidas. Esse método para oráculo de testes é conhecido como teste diferencial e amplamente adotado para teste de compiladores, para linguagens de programação que dispõe de múltiplos compiladores (Chen, 2018).

O Módulo Executor Figura 1 (4) executa os programas resultantes das compilações de cada Mutante pelos diferentes compiladores. Um Mutante, quando compilado (pelo Módulo Compilador), pode gerar um programa executável de saída ou erros de compilação. No caso de um programa executável, este é executado e o seu resultado é capturado. O objetivo desse módulo é capturar incongruências semânticas entre saídas de diferentes compiladores para um mesmo código fonte.

No Módulo Comparador, Figura 1 (5) são comparados entre si os resultados para cada Mutador, capturados nas compilações Figura 1 (3) e nas execuções de seus programas executáveis Figura 1 (4). Resultados incongruentes, que apresentem claramente ou não falhas sintáticas ou semânticas nos compiladores em teste, são evidenciados para posterior análise manual dos pesquisadores.

Os Resultados Figura 1 (6) consistem em todas as informações, capturadas em módulos anteriores, que não coincidem quando comparadas diretamente. Essas informações consistem, para cada Mutador, diferentes mensagens de erro do processo de compilação ou diferentes resultados de execução dos mutadores executados. A análise manual desses resultados é feita pelos autores.

4 Resultados e Discussão

Esta sessão apresenta e discute dados capturados na (i) interação dos pesquisadores com o MLGE, para sintetização dos mutadores de código e no (ii) experimento para teste dos compiladores de GO com duração de 13 horas e 20 minutos, utilizando os mutadores por meio da ferramenta implementada em python.

A interação dos usuários com o MLGE buscou gerar mutadores para 47 diferentes operações de otimização suportadas pelo compilador oficial de Go. Do total de operações, 26 operações tiveram os mutadores gerados no primeiro *prompt*, com o modelo. Para outras 20 operações, foram necessários retornos informativos ao modelo para correção de falhas nos mutadores implementados. Uma operação de otimização teve seus mutadores descartados por exigir modificações em uma opção do compilador para seu funcionamento, o que não está no escopo do experimento. No total, foram gerados 157 mutadores (em média, o MLGE gerou aproximadamente 3 mutadores para cada operação de otimização).

O experimento com a ferramenta contou com 157 mutadores para 46 operações de otimização diferentes e 46 programas Semente iniciais. Ao final do experimento, um total de 165.321 mutantes foram geradas. Da quantia total, aproximadamente 58,78% das mutantes geradas foram novas e únicas e 41,22% eram repetidas e foram descartadas. Do total de mutantes não descartadas, 82,62% foram compiladas sem apontamentos de erros de compilação e gerando um binário executável. A execução dos mutadores implementados pelo MLGE ocorreu sem erros durante o total de 165.321 aplicações, indicando robustez desses modelos de linguagem na implementação dos programas.

A captura de cobertura de código explorado pelas Sementes iniciou-se abrangendo 11,63% da porção do compilador responsável pelas otimizações. Ao final do experimento, esse valor foi ampliado em 4,11% pelas mutantes geradas. Mesmo mutantes geradas ao final do experimento exploraram novos caminhos de execução, indicando potencial para o exercício de novas porções do compilador caso o experimento fosse continuado por mais tempo.

Os resultados do teste diferencial, que compara mensagens de erros de compilação e execução dos binários, indicaram diversas diferenças mas nenhuma falha nas versões estáveis do compilador. Do número total de 5725 mutantes que causaram resultados conflitantes, 12,38% ocorreram em função de melhorias nas mensagens de diagnóstico de compilação, 1,62% resultados de execução de binários aleatórios e 85,99% erros em tempo de execução.

5 Conclusão

O experimento executado indica MLGEs, como o Gemini 3.1 Pro, uma alternativa viável ao usuário sem perícia para sintetização de mutadores de código de Go. A campanha de teste diferencial de Go evidencia a capacidade desses mutadores em gerar programas compiláveis, que exploram porções do código de otimizações, profundas no compilador. É uma limitação importante do trabalho a característica indeterminística de modelos de linguagem.

Referências Bibliográficas

THOMPSON, Ken. Reflections on trusting trust. **Communications of the ACM**, New York, v. 27, n. 8, p. 761–763, aug. 1984. Disponível em: <https://doi.org/10.1145/358198.358210>. Acesso em: 31 jul. 2026.

CHEN, Junjie; PATRA, Jibesh; PRADEL, Michael; XIONG, Yingfei; ZHANG, Hongyu; HAO, Dan; ZHANG, Lu. A survey of compiler testing. **ACM Computing Surveys**, New York, v. 53, n. 1, art. 4, 2020. Disponível em: <https://doi.org/10.1145/3363562>. Acesso em: 31 jul. 2026.

JIANG, Yu; LIANG, Jie; MA, Fuchen; CHEN, Yuanliang; ZHOU, Chijin; SHEN, Yuheng; WU, Zhiyong; FU, Jingzhou; WANG, Mingzhe; LI, Shanshan; ZHANG, Quan. When Fuzzing Meets LLMs: Challenges and Opportunities. In: ACM INTERNATIONAL CONFERENCE ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 32., 2024, Porto de Galinhas. **Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering**. New York: Association for Computing Machinery, 2024. p. 492-496. Disponível em: <https://doi.org/10.1145/3663529.3663784>. Acesso em: 31 jul. 2026.

XIA, Chunqiu Steven; PALTENGHI, Matteo; LE TIAN, Jia; PRADEL, Michael; ZHANG, Lingming. Fuzz4All: Universal Fuzzing with Large Language Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 46., 2024, Lisbon. **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. New York: Association for Computing Machinery, 2024. art. 126. Disponível em: <https://doi.org/10.1145/3597503.3639121>. Acesso em: 31 jul. 2026.

GOOGLE. **Introduction to the Go compiler's SSA backend**. [S. l.], 2025. Disponível em: <https://go.dev/src/cmd/compile/internal/ssa/README>. Acesso em: 31 jul. 2026.

Palavras-chave: Compilador, Modelo de Linguagem de Grande Escala, Geração de código, Fuzzing, Teste diferencial

Nº de Registro no sistema Prisma: PES-2025-0313

Financiamento

