UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FERNANDO SCHREINER MAGNABOSCO

GERAÇÃO DE CÓDIGO RUST ALEATÓRIO UTILIZANDO MODELOS GRANDES DE LINGUAGEM PARA TESTES DE COMPILADORES

CHAPECÓ
2025

FERNANDO SCHREINER MAGNABOSCO

GERAÇÃO DE CÓDIGO RUST ALEATÓRIO UTILIZANDO MODELOS GRANDES DE LINGUAGEM PARA TESTES DE COMPILADORES

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal da Fronteira Sul (UFFS), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Samuel da Silva Feitosa

CHAPECÓ
2025

Magnabosco, Fernando Schreiner

GERAÇÃO DE CÓDIGO RUST ALEATÓRIO UTILIZANDO MODELOS GRANDES DE LINGUAGEM PARA TESTES DE COMPILADORES / Fernando Schreiner Magnabosco – 2025.

3 f.

Orientador: Samuel da Silva Feitosa

Trabalho de Conclusão de Curso (Graduacão) – Universidade Federal da Fronteira Sul, Curso de Ciência da Computação, Chapecó, SC, 2025.

1. Rust 2. Teste de compiladores 3. Modelos de linguagem ampla 4. Geração de código aleatório 5. Fuzzing I. Feitosa, Samuel da Silva, orient. II. Universidade Federal da Fronteira Sul. III. Título.

FERNANDO SCHREINER MAGNABOSCO

GERAÇÃO DE CÓDIGO RUST ALEATÓRIO UTILIZANDO MODELOS GRANDES DE LINGUAGEM PARA TESTES DE COMPILADORES

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal da Fronteira Sul (UFFS), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Samuel da Silva Feitosa

Este Trabalho de Conclusão de Curso foi avaliado e aprovado pela banca avaliadora em: 09/12/2025

BANCA AVALIADORA

gov.br Documento assinado digitalmente
SAMUEL DA SILVA FEITOSA
Data: 17/12/2025 18:03:55-0300
Verifique em https://validar.iti.gov.br

Samuel da Silva Feitosa - UFFS

gov.br Documento assinado digitalmente
ANDREI DE ALMEIDA SAMPAIO BRAGA
Data: 18/12/2025 10:19:50-0300
Verifique em https://validar.iti.gov.br

Andrei de Almeida Sampaio Braga - UFFS

gov.br Documento assinado digitalmente
GIANCARLO DONDONI SALTON
Data: 18/12/2025 13:23:30-0300
Verifique em https://validar.iti.gov.br

Giancarlo Dondoni Salton - UFFS

Geração de código Rust aleatório utilizando modelos grandes de linguagem para testes de compiladores

Fernando Magnabosco
fernando.magnabosco@uffs.edu.br
Universidade Federal da Fronteira Sul
Chapecó, Santa Catarina, Brasil

Resumo

This work proposes an approach for testing Rust compilers using Large Language Models (LLMs) to generate random code focused on unstable features. The methodology implements a differential testing system that uses the official rustc compiler as an oracle to filter semantically valid test cases before submitting them to gccrs, an alternative compiler under development. The system targets complex features such as trait specialization and constant evaluation. Results from 1,403 generated instances show that while LLMs struggle with the strict semantic rules of Rust's nightly features (yielding a 3.7% validity rate), the filtering process proved crucial. The validated test cases revealed that the target compiler (gccrs) is currently limited by infrastructure bottlenecks—specifically due to missing language items and standard library linkage—rather than logic errors in complex passes. Crucially, the absence of critical crashes (ICEs) suggests stability in the compiler's frontend parsing, even if backend integration remains incomplete. This study demonstrates that LLM-based fuzzing can serve as an effective "maturity probe"for emerging compilers.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Language features*; • **Computing methodologies** → *Machine learning*.

Keywords

Rust, Compiler Testing, Large Language Models, Code Generation, Differential Testing, GCC Rust, Software Testing

ACM Reference Format:
Fernando Magnabosco. 2025. Geração de código Rust aleatório utilizando modelos grandes de linguagem para testes de compiladores. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introdução

De acordo com Bugden and Alahmar [1], Rust é uma linguagem de baixo nível que vem crescendo em popularidade nos últimos anos. Entre suas vantagens, destacam-se a segurança no gerenciamento de memória [6], a concorrência e a performance. Atualmente, Rust é utilizada em diversos projetos críticos, substituindo linguagens como C e C++ no kernel de sistemas operacionais e em aplicações de alta escala. Devido a esse aumento expressivo de adoção, muitas bases de código estão sendo convertidas para Rust e novos projetos estão sendo desenvolvidos nativamente na linguagem.

Os conceitos de *ownership*, *borrowing* e *lifetimes* garantem a segurança de memória em Rust [6, 7]. No entanto, para desenvolver certas estruturas de dados de forma eficiente, a linguagem oferece o conceito de *unsafe*, que pode reintroduzir bugs que o sistema de tipos normalmente evitaria. Além disso, muitos compiladores de Rust são relativamente novos comparados aos de outras linguagens consolidadas, e diversas otimizações e garantias de segurança ainda carecem de testes mais exaustivos [9]. Compiladores são ferramentas complexas e centrais no desenvolvimento de software; garantir seu funcionamento correto é, portanto, imperativo.

Diante deste contexto, o problema de pesquisa que norteia este trabalho é: como garantir a corretude dos compiladores de Rust por meio do desenvolvimento de uma ferramenta de teste que utiliza Modelos Grandes de Linguagem (*LLMs*)? A hipótese levantada é que as *LLMs* são capazes de gerar código Rust aleatório em larga escala, e que este código pode ser eficaz para validar a corretude dos compiladores através de testes diferenciais.

O objetivo geral deste trabalho é propor e implementar uma aplicação que utilize *LLMs* para a geração automatizada de código Rust, a fim de testar e validar compiladores. Para alcançar este propósito, foram definidos objetivos específicos que compreendem: a realização de uma revisão bibliográfica sobre técnicas de geração de código aleatório para testes de compiladores; o estudo do formato e da especificação da linguagem Rust; a seleção de uma *LLM* adequada para a tarefa; o desenvolvimento da aplicação geradora; e, por fim, a avaliação da eficácia da ferramenta desenvolvida por meio da validação dos códigos gerados e da análise dos resultados obtidos.

2 Fundamentação Teórica

2.1 Rust

De acordo com Bugden e Alahmar [1], Rust é uma linguagem de programação introduzida pela *Mozilla* em 2010. A linguagem foi projetada para abordar problemas importantes de segurança e performance em sistemas de software, posicionando-se como uma sucessora de linguagens como C++. Inicialmente focada na segurança da manipulação de memória, Rust expandiu seu escopo para incluir performance, adotando os princípios de abstração de custo zero. Este equilíbrio entre segurança e performance é notável, especialmente porque as linguagens de baixo nível historicamente

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

enfrentam dificuldades em combinar esses dois aspectos de forma eficaz.

A segurança em Rust é assegurada por um sistema de tipos robusto, baseado nos conceitos de *ownership* e *borrowing*. *Ownership* é um conceito que assegura que cada valor em Rust tem um único proprietário, e quando esse proprietário sai de escopo, o valor é liberado da memória, eliminando a possibilidade de *memory leaks* e *dangling pointers*. *Borrowing* permite que múltiplas referências a um valor sejam criadas, mas com restrições. Rust assegura que um valor pode ser referenciado múltiplas vezes de forma não mutável, ou mutavelmente, mas apenas por uma única referência, prevenindo assim conflitos de acesso e modificações concorrentes.

O compilador oficial de Rust é o `rustc`, desenvolvido e mantido pela comunidade Rust. Ele utiliza LLVM como *backend* para geração de código e é considerado a implementação de referência da linguagem. Dada a crescente adoção de Rust, surgiram implementações alternativas do compilador, como o `gccrs`—um *frontend* Rust para a coleção de compiladores GCC (*GNU Compiler Collection*). O `gccrs` visa integrar Rust ao ecossistema GCC, permitindo o compartilhamento de otimizações e *backends* com outras linguagens suportadas. No entanto, por estar em desenvolvimento ativo, o `gccrs` ainda não oferece suporte completo a todas as funcionalidades de Rust, particularmente aquelas em fase experimental (*nightly features*). Esta diferença de maturidade entre `rustc` e `gccrs` cria uma oportunidade para testes diferenciais, onde o compilador oficial pode atuar como oráculo para validar a corretude de implementações alternativas.

2.2 Modelos Grandes de Linguagem

Modelos grandes de linguagem (*Large Language Models - LLMs*) são modelos de aprendizado de máquina que são treinados para prever a próxima palavra de uma sentença. Estes modelos são capazes de capturar a complexidade de uma linguagem natural, e são capazes de gerar texto coerente e semanticamente correto. Alguns dos modelos de linguagem mais utilizados atualmente são o *GPT-3* (Generative Pre-trained Transformer 3) e o *BERT* (Bidirectional Encoder Representations from Transformers).

Os *LLMs* são construídos sobre uma arquitetura conhecida como *transformer*, proposta por Vaswani et al. [15], que se baseia em três conceitos principais: codificação posicional, atenção e auto-atenção.

Codificação Posicional: A codificação posicional (*positional encoding*) se refere à técnica de enumerar as palavras do prompt de entrada a fim de preservar a informação sobre a ordem destas [15]. Cada uma destas representações numéricas é chamada de *token*. A partir disto, o *transformer* gera um vetor que consegue capturar a frequência em que os determinados *tokens* se posicionam sequencialmente em um texto [15]. Desta forma, o modelo é capaz de compreender as nuances de sintaxe de uma linguagem, como a capacidade de identificar dependência entre tokens (um determinado token sempre deve aparecer acompanhado de outro token) e a estruturação de frases. Esta técnica representa um avanço notável para os modelos de linguagem, já que sua estrutura difere das técnicas anteriores como a *RNN* que faziam processamento sequencial dos tokens de uma frase [15]. Desta forma, os modelos podem ser treinados de maneira paralela, o que acelera significativamente o processo de treinamento do modelo [15].

Atenção: Atenção é uma técnica que permite ao modelo atribuir valores de relevância para cada um dos *tokens* da entrada. Em um mecanismo de atenção, para cada *token* de entrada, o modelo calcula um valor de atenção relativo a tokens de outra sentença (por exemplo, a sentença de saída de um modelo de tradução) [15]. Este valor de atenção é calculado a partir de uma função de similaridade entre os vetores de representação dos *tokens* de entrada e saída [15].

Auto-atenção é uma extensão da atenção que difere no escopo em que atua. A auto-atenção captura a relação entre os tokens de uma mesma sentença [15], permitindo ao modelo capturar dependências de longo alcance [15]. A auto-atenção é um componente relevante das *LLMs*, e é responsável por grande parte de sua capacidade de capturar dependências sintáticas e semânticas em textos [15].

2.2.1 Modelo Llama 3.1. Diferentemente de modelos proprietários acessíveis apenas via nuvem, a família Llama (*Large Language Model Meta AI*) representa uma mudança de paradigma em direção a modelos de pesos abertos (*open-weights*). Essa característica permite que pesquisadores executem modelos de estado da arte localmente, garantindo privacidade de dados e reprodutibilidade científica sem dependência de serviços externos [4].

A versão utilizada neste trabalho, Llama 3.1, baseia-se em uma arquitetura *Transformer* padrão (apenas decodificador), mas introduz otimizações significativas como a Atenção de Consulta Agrupada (*Grouped Query Attention* - GQA) para aumentar a eficiência de inferência. Treinado em mais de 15 trilhões de *tokens*, o modelo suporta uma janela de contexto de 128k *tokens*, permitindo a manipulação de entradas longas e complexas [4].

Sua variante de 8 bilhões de parâmetros (8B) destaca-se por equilibrar desempenho e requisitos de *hardware*. Ela viabiliza raciocínio complexo e geração de código em GPUs de consumo com memória limitada (VRAM), tornando-a um motor ideal para fluxos de trabalho de teste automatizado que exigem alto volume de geração sem os custos proibitivos de APIs comerciais.

2.2.2 Ollama e Inferência Local. Para operacionalizar o uso de LLMs locais, ferramentas de orquestração são essenciais. O Ollama atua como uma camada de abstração que simplifica a execução de modelos como o Llama em ambientes locais. Ele fornece uma API RESTful que permite aos desenvolvedores interagir com o modelo de forma análoga a serviços em nuvem, enviando *prompts* e recebendo respostas estruturadas, mas com todo o processamento ocorrendo na infraestrutura do usuário.

No contexto deste trabalho, a API do Ollama foi fundamental para automatizar o ciclo de geração de código. Ela permitiu o controle preciso sobre parâmetros de inferência (como temperatura e semente aleatória) e facilitou a integração do modelo Llama 3.1 com o orquestrador de testes desenvolvido em Node.js, eliminando preocupações com latência de rede e limites de taxa (*rate limits*) impostos por provedores externos.

2.3 Geração de Código Aleatório

A geração de código aleatório é uma técnica não determinística que funciona por meio da criação de entradas de teste para descobrir

falhas e vulnerabilidades em sistemas computacionais. Esta abordagem surge como resposta à necessidade de identificar comportamentos inesperados, erros de implementação e falhas de segurança que são difíceis de detectar por testes convencionais ou pela utilização regular dos usuários. A técnica é útil em contextos onde a verificação formal é impraticável devido a sua complexidade e custo elevado [11].

Fuzzing: *Fuzzing* é uma técnica de teste de software que tem ganhado popularidade devido à sua eficiência e reprodutibilidade na descoberta de *bugs* e vulnerabilidades. Esta abordagem envolve a geração e mutação de entradas para testar programas alvo enquanto coleta cobertura de código como feedback [3].

A eficácia do *fuzzing* é demonstrada pela sua adoção na indústria de software. Por exemplo, o OSS-Fuzz, uma plataforma que implanta *fuzzers* para software de código aberto, conseguiu identificar e resolver mais de 8.900 vulnerabilidades e 28.000 bugs em 850 projetos até fevereiro de 2023 [10].

O impacto das ferramentas de *fuzzing* na identificação de *bugs* até então não descobertos é notável: ferramentas como o jsfunfuzz descobriram mais de 1.700 bugs no SpiderMonkey (o motor JavaScript do Firefox), enquanto o LangFuzz descobriu mais de 500 bugs no mesmo motor. De forma similar, a ferramenta de fuzzing do Google, o ClusterFuzz, processa aproximadamente cinquenta milhões de casos de teste diariamente no Chrome, detectando 95 vulnerabilidades em um período de quatro meses, e o Csmith identificou mais de 450 bugs até então desconhecidos em compiladores C [2].

2.4 Desafios Específicos da Validação em Rust

Embora técnicas de *fuzzing* sejam amplamente bem-sucedidas em linguagens como C e C++, sua aplicação direta em Rust enfrenta barreiras arquiteturais significativas. A principal delas reside na rigidez do verificador de empréstimos (*borrow checker*).

Em linguagens tradicionais, um gerador aleatório pode produzir sequências de bytes ou *tokens* que, embora semanticamente sem sentido, são sintaticamente válidos o suficiente para exercitar o *parser* ou o otimizador do compilador. Em Rust, no entanto, a camada de análise semântica é extremamente estrita. Estudos recentes sobre o ecossistema de testes revelam que a grande maioria dos programas gerados aleatoriamente (*naive fuzzing*) é rejeitada antes mesmo de atingir as etapas de geração de código intermediário (MIR/LLVM IR), tornando o teste ineficaz para encontrar falhas profundas no *backend* [3, 12].

Este fenômeno cria o que a literatura denomina de "Problema da Validade de Entrada". Para testar efetivamente um compilador Rust, a ferramenta de geração não deve apenas respeitar a gramática livre de contexto da linguagem, mas também satisfazer regras de contexto sensíveis, como tempos de vida (*lifetimes*) válidos, regras de mutabilidade exclusiva e traços (*traits*) coerentes.

Ferramentas como o RustSmith [12] tentam resolver isso através de gramáticas construtivas determinísticas. No entanto, esta abordagem muitas vezes falha em gerar o tipo de código "criativo"ou idiomático que humanos escrevem. É nesta lacuna que o uso de LLMs se insere: ao invés de construir código via regras rígidas, utiliza-se a natureza probabilística do modelo para gerar programas que são estatisticamente propensos a serem válidos, delegando a validação final a um oráculo externo, como proposto neste trabalho.

3 Trabalhos Relacionados

3.1 RustSmith: Random Differential Compiler Testing

Liu et al. [12] propõem o RustSmith, um gerador de programas aleatórios que constrói Árvores de Sintaxe Abstrata (AST) válidas, respeitando rigorosamente as regras de tipos e *borrowing* do Rust. A ferramenta é robusta ao testar a cadeia de compilação de ponta a ponta através de uma gramática formal. Enquanto o RustSmith foca na correção estrutural garantida, este trabalho investiga o potencial das *LLMs* como uma estratégia alternativa e probabilística, buscando gerar padrões de código que diferem daqueles produzidos por gramáticas estritas, com foco na validação do compilador experimental *gccrs*.

3.2 AID: LLM-Powered Test Case Generation

Liu et al. [8] apresentam o AID, uma técnica que combina *LLMs* com testes diferenciais para gerar oráculos e casos de teste para programas existentes. O sistema utiliza a IA para criar geradores de entrada baseados em restrições, auxiliando na detecção de falhas complexas. A distinção deste trabalho reside no objeto de teste: enquanto o AID busca detectar falhas na lógica de softwares desenvolvidos por humanos, nossa abordagem utiliza as *LLMs* para gerar os próprios programas, visando avaliar o comportamento do processo de tradução do compilador diante de códigos sinteticamente gerados.

3.3 SyRust e Crabtree

Para mitigar vulnerabilidades em códigos que utilizam o bloco *unsafe*, Li et al. desenvolveram o SyRust [14] e o Crabtree [13]. O SyRust utiliza síntese baseada em SAT a partir de especificações de API, enquanto o Crabtree evolui essa técnica adicionando *fuzzing* para escalabilidade. Estes trabalhos concentram seus esforços na validação de bibliotecas e no uso seguro de APIs. Em contrapartida, o presente trabalho direciona-se ao teste do *frontend* e de funcionalidades da linguagem Rust — como o sistema de *traits* e macros — avaliando a conformidade do compilador *gccrs* em desenvolvimento.

3.4 Rust-twins

Zhang et al. [17] propõem o Rust-twins, que utiliza *LLMs* para mutar funções Rust e transformá-las em macros equivalentes, comparando suas expansões para detectar inconsistências. O Rust-twins realiza uma validação aprofundada especificamente no sistema de expansão de macros. Este trabalho, por sua vez, adota uma abordagem exploratória voltada às diversas funcionalidades suportadas atualmente pelo *gccrs*, buscando avaliar componentes como a resolução de tipos e avaliação de constantes sob uma perspectiva diferente da equivalência de macros.

3.5 Geração de Código para Compiladores (Go e Fuzz4All)

Xia et al. [5] utilizam *LLMs* para testar a cobertura do compilador Go, enquanto o Fuzz4All [16] propõe uma abordagem universal

de *fuzzing* para múltiplas linguagens. Embora compartilhem o uso de IA generativa, a aplicação dessas técnicas em Rust enfrenta desafios específicos devido às regras de memória da linguagem. Nossa metodologia busca mitigar essa dificuldade implementando um ciclo de *feedback* com o oráculo *rustc*, visando assegurar a validade semântica necessária para o teste diferencial do compilador alvo.

4 Metodologia e Desenvolvimento

Para validar a hipótese de pesquisa, foi desenvolvida uma ferramenta de teste diferencial automatizada. A metodologia adotada consiste em utilizar LLMs como geradores de casos de teste (*fuzzers* semânticos) e o compilador oficial `rustc` como oráculo para validar a correção dos códigos antes de submetê-los ao compilador alvo, o `gccrs`.

A implementação do sistema e a definição do ambiente experimental são detalhadas a seguir, descrevendo como cada etapa do processo foi concretizada.

4.1 Arquitetura da Ferramenta

O sistema foi desenvolvido em JavaScript e executado no ambiente Node.js. A escolha se baseou na ampla disponibilidade de bibliotecas para integração com APIs e manipulação de processos do sistema operacional. A aplicação foi estruturada de forma modular para garantir o desacoplamento entre as etapas de geração e compilação:

Módulo de Geração: Encapsula a interação com as LLMs através de uma interface agnóstica que permite a troca entre diferentes provedores (como Gemini e Ollama). Para este trabalho, os experimentos concentraram-se em modelos executados localmente via Ollama (Llama 3.1:8b), escolhido por contornar limitações de taxa de requisição de APIs públicas e também limitações de hardware disponível. O sistema foi implementado com suporte a múltiplos tipos de modelos, permitindo a integração de outros provedores e versões do modelo; contudo, apenas o Llama 3.1:8b foi efetivamente executado durante a validação experimental.

Módulo de Compilação (`code_compilation`): Responsável por abstrair as chamadas de sistema necessárias para invocar o compilador alvo. Este módulo gerencia o ciclo de vida da compilação e captura os códigos de saída e fluxos de erro para análise posterior.

Persistência de Dados (`database`): Para garantir a reprodutibilidade, utilizou-se o banco de dados SQLite. O esquema foi desenhado para rastrear a linhagem completa do teste: desde o *prompt* inicial (tabela `prompts`) e o lote de execução (`batches`), até as tentativas de correção pelo oráculo (`code_retries`) e o resultado final no alvo (`tests`).

Orquestração de Eventos: Um barramento de eventos foi implementado para gerenciar a comunicação assíncrona. Isso permitiu que o sistema paralelizasse a geração e a validação de múltiplos casos de teste sem bloqueios de I/O.

4.2 Fluxo de Teste Diferencial (Pipeline)

O funcionamento do sistema segue um ciclo de geração, validação e reparo. Ao iniciar um lote, o orquestrador dispara instâncias de teste que operam conforme o seguinte fluxo:

(1) **Geração:** O sistema solicita ao LLM um código Rust baseado em um *prompt* específico.
(2) **Validação pelo Oráculo:** O código é submetido imediatamente ao `rustc`. Se a compilação falhar, o erro é capturado e enviado de volta ao LLM como *feedback* para uma nova tentativa de correção. Este laço se repete até que o código seja válido ou o limite de tentativas (configurado em 15) seja atingido.
(3) **Teste no Alvo:** Apenas códigos validados pelo oráculo são submetidos ao `gccrs`. O resultado (sucesso, erro ou *crash*) é classificado e armazenado.

Essa abordagem de oráculo ativo foi fundamental para garantir que o `gccrs` fosse testado apenas com programas Rust válidos, evitando falsos positivos decorrentes de sintaxe incorreta. O Algoritmo 1 detalha a lógica implementada para o laço de geração e reparo iterativo.

Algoritmo 1 Geração e Reparo Iterativo com Oráculo

Entrada: Prompt Inicial P, Limite de Tentativas N
Saída: Código Válido C ou Falha $\perp$

$C \leftarrow \text{LLM}(P)$ ▷ Geração inicial do código
$tentativas \leftarrow 0$
while $tentativas < N$ **do**
  $resultado \leftarrow \text{OracleCompiler}(C)$ ▷ Compilação com rustc
  if $resultado$ é Sucesso **then**
    return C ▷ Código validado pelo oráculo
  else
    $erro \leftarrow \text{ExtrairErro}(resultado)$
    $P' \leftarrow \text{ConstruirPromptDeCorrecao}(C, erro)$
    $C \leftarrow \text{LLM}(P')$ ▷ Geração de nova versão
    $tentativas \leftarrow tentativas + 1$
  end if
end while
return $\perp$ ▷ Falha após exceder limite de tentativas

4.3 Estratégias de Prompting e Funcionalidades

A análise prévia da especificação da linguagem Rust e do status de desenvolvimento do `gccrs` guiou a elaboração das estratégias de geração. Foram definidos dois tipos de abordagem:

Zero-shot Prompting: O modelo recebe apenas a instrução para gerar código com certas características, testando sua capacidade de generalização. Por exemplo, um prompt típico desta estratégia seria:

> *"You are a world-class Rust expert specializing in rustc's core language features, particularly its trait solver and type system. Your deep understanding of the compiler's internals allows you to craft code that targets its most complex and fragile analysis passes.*
> *Your mission is to generate a single Rust source file (.rs) designed to stress-test the Rust compiler. The code must be entirely self-contained, compiling directly with rustc without needing Cargo or any external crates. Your goal is to expose latent bugs by creating an extreme edge case for a single, advanced language feature.*
> *Core Strategy: Deep min_specialization and Trait Overlap. Your goal is to push the min_specialization feature to its absolute limits by creating a complex hierarchy*

of traits with default implementations that are then specialized. The complexity should arise from multiple layers of specialization, potential ambiguities that the compiler must resolve, and interactions with other features like associated types or complex where clauses. Output Rules: The output must be a single, self-contained Rust file (.rs) compilable with rustc alone. The code must contain a fn main() entry point. No Cargo.toml or external crates. The code must target the latest nightly Rust toolchain and include #![feature(min_specialization)] at the top of the file. Return only the raw Rust code."

Few-shot Prompting: Exemplos de pares "instrução-código"são fornecidos para guiar o estilo e reforçar o uso de construções suportadas pelo compilador alvo. Um exemplo desta abordagem seria um prompt direcionado a *min_specialization*, fornecendo exemplos de complexidade crescente seguidos da instrução de geração:

"You are a world-class Rust expert specializing in rustc's core language features. Your mission is to generate a single, self-contained Rust source file (.rs) to stress-test the min_specialization feature in the rust compiler.
Below are three examples of the kind of code structure required. Study how they escalate in complexity regarding trait overlap and default implementations.
Example 1: Basic Overlap Resolution [...code example with trait Abstract, generic impl, and specialized impl for u8...]
Example 2: Specialization with Associated Types [...code example with trait Container and specialized associated types...]
Example 3: Complex Lifetime and Bound Interaction [...code example with trait Convert and lifetime interactions...]
Your Task: Based on the patterns above, generate a NEW, unique, and significantly more complex test case. It must focus on min_specialization, introduce deep nesting or ambiguous trait bounds that force the compiler to work hard to resolve the correct impl. It must be self-contained, use #![feature(min_specialization)], and compile with rustc. Do not output markdown or explanations. Return only the raw code."

Os testes focaram em funcionalidades complexas que exigem resolução de tipos avançada, divididas em categorias como: **Sistema de Tipos** (especialização de traits, limites recursivos) e **Avaliação de Constantes** (macros `offset_of!`, inferência de constantes adiada).

4.4 Ambiente Experimental

Para assegurar a viabilidade da execução dos modelos locais e a reprodutibilidade, os experimentos foram conduzidos em um ambiente controlado com as seguintes especificações:

- **Hardware:** Processador AMD Ryzen 5 1400, 16 GB RAM DDR4 e GPU NVIDIA GeForce GTX 1070 (8 GB VRAM), essencial para o carregamento dos pesos do modelo Llama 3.1.
- **Software:** Sistema Operacional Linux Fedora 43, Node.js v22.20.0.
- **Compiladores:** Oráculo `rustc 1.91.0-nightly` e Alvo `gccrs 16.0.0 (experimental)`.

5 Análise dos Resultados

Este capítulo apresenta a análise quantitativa e qualitativa dos dados coletados durante os experimentos de validação cruzada. Os resultados demonstram a eficácia da arquitetura de geração iterativa proposta e revelam o estágio atual de maturidade do compilador alvo (`gccrs`) frente a construções complexas da linguagem Rust.

5.1 Visão Geral dos Experimentos

Durante a janela de experimentos, o sistema gerou um total de **1.403** instâncias de código fonte. Deste montante, o mecanismo de validação via oráculo (`rustc`) aprovou **52** casos de teste únicos, que foram subsequentemente submetidos ao compilador alvo.

A Tabela 1 resume o "funil"de execução do processo de *fuzzing*.

Tabela 1: Funil de execução e taxas de aprovação por estágio.

Estágio	Quantidade	Taxa de Retenção
Códigos Gerados (LLM)	1.403	100,00%
Aprovados pelo Oráculo	52	3,71%
Executados no Alvo	52	100,00%
Sucesso no Alvo	**1**	**1,92%**

A baixa taxa de aprovação inicial pelo oráculo (3,7%) evidencia a dificuldade intrínseca de gerar código Rust sintaticamente válido e semanticamente correto em funcionalidades instáveis (*nightly*), justificando a necessidade crítica do laço de reparo (*retry loop*) implementado na arquitetura.

5.2 Análise da Geração e Reparo de Código

A eficácia do modelo de linguagem (Llama 3.1) em gerar código válido em apenas uma iteração mostrou-se limitada no contexto de funcionalidades avançadas. A distribuição das tentativas necessárias para alcançar um código válido pelo oráculo é apresentada na Tabela 2.

Tabela 2: Distribuição do número de tentativas até a validação pelo oráculo.

Número da Tentativa	Quantidade de Sucessos
1	1
2	7
3	6
4	10
5	5
6 a 12	23

Observa-se que apenas um único caso de teste foi aceito na primeira tentativa. A moda da distribuição situa-se na **4ª tentativa**, indicando que o mecanismo de retroalimentação de erro (*error feedback*) foi essencial. O sistema precisou, em média, de múltiplos ciclos de correção para ajustar detalhes de sintaxe, resolução de tipos

e lifetimes que o modelo inicialmente alucinou ou implementou incorretamente.

5.3 Análise Comparativa de Estratégias

Para avaliar o impacto da engenharia de *prompt* na qualidade do código gerado, os experimentos foram segregados entre as estratégias *Zero-shot* (apenas instruções) e *Few-shot* (instruções + exemplos). A Tabela 3 apresenta a eficácia de cada abordagem na geração de códigos válidos para o oráculo.

Tabela 3: Taxa de aprovação pelo oráculo (`rustc`) por estratégia.

Estratégia	Total Gerado	Válidos (Oráculo)	Taxa (%)
Zero-Shot	1.197	50	**4,18%**
Few-Shot	206	2	0,97%

A estratégia *Zero-shot* demonstrou uma taxa de aprovação significativamente superior (4,18%) em comparação ao *Few-shot* (0,97%). Este resultado sugere que, para o modelo utilizado (Llama 3.1:8b) e no contexto de funcionalidades instáveis do Rust, a liberdade criativa do *Zero-shot*, combinada com o laço de reparo, foi mais eficaz do que a tentativa de emular padrões complexos fornecidos nos exemplos do *Few-shot*. É possível que os exemplos fornecidos tenham induzido o modelo a gerar construções excessivamente complexas, dificultando a validação sintática inicial. Ademais, considerando que o modelo possui apenas 8 bilhões de parâmetros, a complexidade dos exemplos fornecidos na estratégia *Few-shot* pode ter excedido sua capacidade de generalização, levando à geração de código com erros estruturais que o mecanismo de reparo não conseguiu corrigir eficientemente.

No entanto, ao analisar a eficiência do processo de reparo (Tabela 4), observa-se um comportamento distinto.

Tabela 4: Eficiência do mecanismo de reparo por estratégia.

Estratégia	Média de Tentativas	Min	Max
Zero-Shot	5,68	2	12
Few-Shot	**1,50**	1	2

Embora a estratégia *Zero-shot* tenha produzido a vasta maioria dos códigos válidos, ela dependeu pesadamente do mecanismo de *retry*, exigindo em média 5,68 tentativas para alcançar um código aceitável. Em contrapartida, nos raros casos em que o *Few-shot* funcionou, o código gerado estava praticamente correto desde o início (média de 1,5 tentativas). Isso indica que o *Zero-shot* tende a produzir esboços "quase corretos"que o orquestrador refina iterativamente, enquanto o *Few-shot* tende a ser uma estratégia de "tudo ou nada".

Por fim, cabe destacar que o único caso de teste que obteve sucesso na compilação pelo alvo (`gccrs`) originou-se da estratégia ***Zero-shot***, reforçando-a como a abordagem mais produtiva para a descoberta de comportamentos válidos no estágio atual da ferramenta.

5.4 Comportamento do Compilador Alvo

Dos 52 casos de teste validados pelo oráculo, o compilador alvo `gccrs` obteve sucesso na compilação de apenas 1 caso (1,9%). Nos 51 casos restantes, o compilador rejeitou o código.

Crucialmente, **não foram registrados travamentos críticos** (*crashes*, *segmentation faults* ou *Internal Compiler Errors* - ICEs) durante os testes. Todas as falhas resultaram em mensagens de erro controladas, o que atesta a estabilidade do *frontend* do `gccrs` em lidar com entradas inválidas para seu estado atual. No entanto, embora a ausência de ICEs sugira estabilidade no *parser*, a alta taxa de falhas em itens de infraestrutura (*Lang Items*), detalhada a seguir, indica que o `gccrs` rejeita os programas prematuramente, antes que as fases de otimização ou geração de código — onde *crashes* são mais comuns — sejam atingidas. Assim, a ausência de falhas críticas pode refletir mais a incompletude do *pipeline* de compilação do que propriamente a robustez dos estágios mais avançados.

A análise das mensagens de erro permitiu categorizar as falhas em quatro grupos principais, conforme detalhado na Tabela 5.

Tabela 5: Categorização das falhas de compilação no `gccrs`.

Categoria do Erro	Ocorrências	Proporção
Biblioteca Padrão / Linkagem	22	43,1%
Infraestrutura Crítica (*Lang Items*)	17	33,3%
Funcionalidades Não Suportadas	10	19,6%
Erros de Parser	2	3,9%
Total de Falhas	**51**	**100%**

5.5 Estudos de Caso de Falhas e Sucesso

Para ilustrar a natureza das limitações encontradas no `gccrs`, foram selecionados três casos representativos extraídos dos testes: uma falha de infraestrutura, uma falha de vinculação e o único caso de sucesso.

5.5.1 Caso 1: Falha de Infraestrutura Crítica. O Código 1 foi gerado com o objetivo de testar a *feature* de especialização mínima (`min_specialization`). Ele define uma implementação genérica (`impl<X>`) restrita por um limite de *trait*.

Código 1: Código rejeitado por falha na resolução de Lang Items.

```
#![feature(min_specialization)]

trait A { fn foo(&self); }
struct S;
impl A for S { fn foo(&self) {} }

trait B: A { fn foo(&self); }
struct X;
impl A for X { fn foo(&self) {} }

// O compilador falha ao processar este generico <X>
impl<X> A for X where X: B {
    default fn foo(&self) {}
}

fn main() {
    let x = X;
    x.foo();
}
```

Erro reportado: `fatal error: failed to find lang item sized`

Análise: A falha ocorre na definição genérica `impl<X>`. Em Rust, parâmetros genéricos possuem implicitamente o limite `Sized` (tamanho conhecido em tempo de compilação), a menos que relaxados com `?Sized`. O erro indica que o `gccrs` não conseguiu localizar a definição interna desse conceito fundamental (*lang item*), abortando a compilação antes mesmo de tentar resolver a lógica complexa de especialização proposta pelo teste. Isso demonstra como falhas de infraestrutura básica impedem a validação de funcionalidades avançadas.

5.5.2 Caso 2: Falha de Vinculação da Biblioteca Padrão. O Código 2 apresenta uma lógica válida de implementação de *traits* (`Default` e um *trait* genérico customizado), mas falha na etapa de geração de saída.

Código 2: Código rejeitado por falha na expansão de macros da std.

```
#![allow(dead_code)]
struct S;

trait Trait<T> { fn method(&self) -> &T; }

impl Default for S {
    fn default() -> Self { S }
}

impl Trait<()> for S {
    fn method(&self) -> &() { &() }
}

fn main() {
    let s = S;
    // O erro ocorre especificamente nesta linha
    println!("{:?}", s.method());
}
```

Erro reportado: `error: could not resolve macro invocation 'println'`

Análise: Este caso evidencia a desconexão com a biblioteca padrão (`libstd`). Todo o código anterior à função `main` é semanticamente válido e aceito pelo oráculo. No entanto, o compilador alvo é incapaz de expandir a macro `println!`, uma ferramenta essencial para depuração e saída de dados. Esta categoria de erro foi responsável por 43% das falhas, sugerindo que a integração do *frontend* Rust com o *backend* do GCC ainda carece do "prelúdio"padrão completo.

5.5.3 Caso 3: O Caso de Sucesso. O único caso de teste que obteve sucesso na compilação pelo alvo (Código 3) originou-se da estratégia *Zero-shot*.

Código 3: O único código aceito pelo compilador alvo nos testes.

```
#![allow(dead_code)]

trait Recur<'a> {
        type AssocType;
        fn recur(&self) -> Self::AssocType;
    }

struct S;

impl Recur<'_> for S {
        type AssocType = ();
        fn recur(&self) -> () {}
    }

fn main() {
        let _s: S;
    }
```

Análise: O sucesso deste código é revelador. Diferente das falhas anteriores, este exemplo demonstra que o *frontend* do `gccrs` já possui suporte funcional para definição de *traits* com parâmetros de tempo de vida (`'a`), resolução de Tipos Associados e inferência de *lifetimes* anônimos. O código "sobreviveu"ao processo de compilação justamente por ser minimalista: não invoca macros da biblioteca padrão (evitando o erro do Caso 2) e utiliza tipos concretos em vez de genéricos abertos (evitando o erro do Caso 1).

5.6 Discussão

Os dados obtidos permitem uma análise crítica sobre três vertentes: a capacidade gerativa da *LLM*, a eficácia do oráculo e a maturidade do compilador alvo.

Eficiência da Geração e o Custo da Validade. A taxa de retenção de 3,7% pelo oráculo (52 códigos válidos de 1.403 gerados) evidencia a barreira imposta pela verificação semântica rigorosa do Rust. Ao contrário de *fuzzers* baseados em gramática, que garantem correção sintática por construção, a *LLM* opera probabilisticamente. O baixo rendimento não indica falha da metodologia, mas quantifica a dificuldade de gerar código válido para funcionalidades instáveis (*nightly*) sem supervisão formal. O mecanismo de reparo iterativo mostrou-se, portanto, não apenas útil, mas indispensável para viabilizar o uso de modelos de linguagem neste domínio.

Diagnóstico de Maturidade do `gccrs`. A análise dos 51 casos rejeitados pelo `gccrs` revela que o compilador não falhou devido à complexidade lógica dos testes (como especialização de *traits*), mas sim por lacunas em sua infraestrutura basal. O fato de 76,4% das falhas serem atribuídas a problemas de linkagem com a biblioteca padrão ou ausência de *Lang Items* (Tabela 5) sugere que o compilador alvo ainda possui "gargalos de infraestrutura". Estes gargalos mascaram a validação das funcionalidades mais avançadas: o compilador aborta o processo antes de atingir as etapas de análise onde *bugs* lógicos profundos poderiam residir.

Estabilidade vs. Rejeição Precoce. A total ausência de falhas críticas (*Segmentation Faults* ou ICEs) é um indicador ambivalente. Por um lado, demonstra que o *frontend* do `gccrs` é resiliente e capaz de tratar erros de forma controlada. Por outro, essa estabilidade pode ser consequência da rejeição precoce dos códigos descrita acima. Conclui-se que o `gccrs`, em sua versão atual, possui um *parser* estável, mas uma integração de *backend* insuficiente para sustentar testes diferenciais semânticos em larga escala. A ferramenta desenvolvida, portanto, cumpriu o papel de "sonda de maturidade", identificando exatamente em qual camada de abstração o desenvolvimento do compilador se encontra bloqueado.

6 Conclusão

Este trabalho explorou a aplicação de Modelos Grandes de Linguagem para a geração de casos de teste aleatórios voltados a compiladores Rust. A combinação de *LLMs* com a técnica de teste diferencial demonstrou potencial como uma estratégia complementar para identificar inconsistências e avaliar o estágio de maturidade de compiladores em desenvolvimento, especificamente o `gccrs`.

Foi desenvolvido um sistema modular que utiliza *LLMs* para gerar código Rust, focado em construções da linguagem que exigem resolução complexa de tipos. A arquitetura implementada permitiu a integração com provedores de modelos locais e a aplicação

de estratégias de *prompting* direcionadas a funcionalidades como especialização de traits e avaliação de constantes.

Os experimentos resultaram na geração de 1.403 instâncias de código. Destas, 52 foram validadas semanticamente pelo oráculo (`rustc`). A submissão destes casos ao `gccrs` resultou em 1 compilação bem-sucedida, revelando que as limitações atuais do compilador alvo concentram-se majoritariamente na integração com a biblioteca padrão e na definição de itens de infraestrutura (*lang items*). A ausência de erros internos críticos (*ICEs*) sugere uma estabilidade preliminar no *frontend* do `gccrs`, embora o processo de compilação seja frequentemente interrompido antes das etapas de otimização.

Os resultados obtidos fornecem indícios que corroboram a hipótese de que *LLMs* podem auxiliar na geração de código Rust para validação de compiladores, apesar dos desafios impostos pelas regras estritas da linguagem. A taxa de retenção inicial de 3,7% pelo oráculo reitera a necessidade do mecanismo de reparo iterativo adotado. Desta forma, o sistema desenvolvido atua não apenas como um gerador de testes, mas como um instrumento de diagnóstico capaz de sondar as fronteiras de implementação de novos compiladores no ecossistema Rust.

6.0.1 Trabalhos Futuros. Diversas direções de pesquisa emergem deste estudo. Primeiramente, a exploração de modelos com maior capacidade de parâmetros (como Llama 3.3 70B ou Claude Sonnet) tende a elevar a qualidade sintática e semântica do código gerado, potencialmente reduzindo a dependência do laço de reparo. Em segundo lugar, o refinamento das técnicas de engenharia de *prompt*, como *chain-of-thought*, pode aprimorar a complexidade lógica dos testes. Adicionalmente, à medida que o `gccrs` evolui e supera seus gargalos de infraestrutura, este sistema poderá ser redirecionado para avaliar otimizações de nível intermediário (MIR) e a geração de código objeto. Por fim, a adaptação da metodologia para outros compiladores, como `rustc_codegen_cranelift`, ou ferramentas de análise estática, representa uma extensão natural para contribuir com a robustez do ecossistema Rust.

Referências

[1] William Bugden and Ayman Alahmar. 2022. Rust: The Programming Language for Safety and Performance. arXiv:2206.05503 [cs.PL]

[2] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. *SIGPLAN Not.* 48, 6 (June 2013), 197–208. doi:10.1145/2499370.2462173

[3] Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. 2024. RUG: Turbo LLM for Rust Unit Test Generation. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*. IEEE. https://rug4rust.github.io/.

[4] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[5] 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* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2201–2203. doi:10.1145/3611643.3617850

[6] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[7] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, San Francisco.

[8] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. arXiv:2404.10304 [cs.SE]

[9] Zixi Liu, Yang Feng, Yunbo Ni, Shaohua Li, Xizhe Yin, Qingkai Shi, Baowen Xu, and Zhendong Su. 2025. An Empirical Study of Rust-Specific Bugs in the rustc Compiler. arXiv:2503.23985 [cs.PL] https://arxiv.org/abs/2503.23985

[10] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677 [cs.CR]

[11] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. doi:10.1145/1982595.1982615

[12] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. doi:10.1145/3597926.3604919

[13] Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2024. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 293 (Oct. 2024), 30 pages. doi:10.1145/3689733

[14] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. doi:10.1145/3453483.3454084

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 5998–6008. doi:10.5555/3295222.3295349

[16] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, 1–13. doi:10.1145/3597503.3639121

[17] Wenzhang Yang, Cuifeng Gao, Xiaoyuan Liu, Yuekang Li, and Yinxing Xue. 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 631–642. doi:10.1145/3691620.3695059

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009