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

GABRIEL ZORTEA SALVI

SOLFORGE: UMA ABORDAGEM BASEADA EM MODELOS DE LINGUAGEM DE GRANDE ESCALA PARA O TESTE DIFERENCIAL DE COMPILADORES SOLIDITY

CHAPECÓ
2025

GABRIEL ZORTEA SALVI

SOLFORGE: UMA ABORDAGEM BASEADA EM MODELOS DE LINGUAGEM DE GRANDE ESCALA PARA O TESTE DIFERENCIAL DE COMPILADORES SOLIDITY

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. Dr. Samuel da Silva Feitosa

CHAPECÓ
2025

Salvi, Gabriel Zortea

SOLFORGE: UMA ABORDAGEM BASEADA EM MODELOS DE LINGUAGEM DE GRANDE ESCALA PARA O TESTE DIFERENCIAL DE COMPILADORES SOLIDITY / Gabriel Zortea Salvi – 2025.

38 f.

Orientador: Dr. Samuel da Silva Feitosa

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

1. solidity 2. contratos inteligentes 3. geração de código 4. fuzzing 5. testes diferenciais I. Feitosa, Dr. Samuel da Silva, orient. II. Universidade Federal da Fronteira Sul. III. Título.

GABRIEL ZORTEA SALVI

SOLFORGE: UMA ABORDAGEM BASEADA EM MODELOS DE LINGUAGEM DE GRANDE ESCALA PARA O TESTE DIFERENCIAL DE COMPILADORES SOLIDITY

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. Dr. Samuel da Silva Feitosa

Este Trabalho de Conclusão de Curso foi avaliado e aprovado pela banca avaliadora em: 11/12/2025.

BANCA AVALIADORA

Dr. Samuel da Silva Feitosa - UFFS

Dr. Andrei de Almeida Sampaio Braga

Dr. Guilherme Dal Bianco

DEDICATÓRIA

Dedico este trabalho a meus pais.

“The greatest obstacle to discovery is not ignorance — it is the illusion of knowledge.”
(Daniel J. Boorstin)

RESUMO

A linguagem Solidity é atualmente a mais utilizada para o desenvolvimento de contratos inteligentes em plataformas blockchain como o Ethereum. Garantir a correta execução desses contratos e a confiabilidade dos compiladores é essencial, visto que erros podem acarretar prejuízos financeiros e comprometimento da segurança. No entanto, os métodos tradicionais de teste manual não são suficientes para expor todas as vulnerabilidades potenciais nos compiladores Solidity. Este trabalho tem como objetivo o desenvolvimento de uma ferramenta para geração automática de contratos inteligentes em Solidity, que servirão como casos de teste para análise e validação alvos de teste. Para aumentar a eficiência e profundidade dos testes, serão utilizadas estratégias de fuzzing, possibilitando explorar uma grande quantidade de caminhos de execução e revelando comportamentos inesperados dos compiladores, contribuindo assim para a robustez do ecossistema de contratos inteligentes.

Palavras-Chave: solidity, contratos inteligentes, geração de código, fuzzing, testes diferenciais.

ABSTRACT

Solidity is currently the most widely used language for developing smart contracts on blockchain platforms such as Ethereum. Ensuring the correct execution of these contracts and the reliability of their compilers is essential, since errors can lead to financial losses and security breaches. However, traditional manual testing methods are not sufficient to expose all potential vulnerabilities in Solidity compilers. This work aims to develop a tool for the automatic generation of Solidity smart contracts that will serve as test cases for compiler analysis and validation. To enhance the efficiency and depth of the testing process, fuzzing strategies will be employed, enabling the exploration of a large number of execution paths and revealing unexpected compiler behaviors. This contributes directly to strengthening the robustness of the smart contract ecosystem.

Keywords: solidity, smart contracts, code generation, fuzzing, differential testing.

LISTA DE TABELAS

LISTA DE SIGLAS

EVM – Ethereum Virtual Machine

SUMÁRIO

1. INTRODUÇÃO

A tecnologia blockchain consolidou-se como uma das transformações mais significativas do século XXI, sustentada por pilares como descentralização, imutabilidade e transparência. Embora sua aplicação tenha se expandido para áreas críticas, desde registros de propriedade a moedas digitais (TRIPATHI; AHAD; CASALINO, 2023), essa onipresença impõe uma necessidade rigorosa de confiabilidade. Diferentemente de softwares convencionais, onde atualizações são triviais, os contratos inteligentes operam em um ambiente imutável: falhas implantadas tendem a ser permanentes ou de correção custosa (POLITOU et al., 2019). Adicionalmente, como esses contratos frequentemente movimentam ativos financeiros de alto valor, qualquer vulnerabilidade pode acarretar prejuízos econômicos severos e comprometer a credibilidade do ecossistema (LIANG et al., 2025).

Nesse contexto, a segurança da linguagem Solidity, predominante na plataforma Ethereum, torna-se central. A robustez dos contratos depende não apenas do código fonte, mas da integridade do seu compilador oficial, o *solc*. Falhas nesta etapa são particularmente perigosas, pois podem gerar bytecodes vulneráveis a partir de códigos aparentemente corretos, introduzindo erros silenciosos e críticos (CHEN et al., 2020; MA et al., 2024).

Diante da necessidade de mitigar tais riscos, este trabalho investiga a viabilidade de aplicar Modelos de Linguagem de Grande Escala na geração automática de código para fins de teste. Parte-se da hipótese de que essa abordagem possibilita a criação sistemática de casos de teste eficientes na detecção de inconsistências em compiladores. Para validar essa premissa, o objetivo central desta pesquisa é desenvolver uma ferramenta que automatize a geração de código Solidity visando estressar e validar os compiladores Solidity em diferentes configurações. A execução deste objetivo fundamenta-se, inicialmente, em uma revisão bibliográfica sobre técnicas de teste de compiladores, seguida pela aplicação prática de Modelos de Linguagem de Grande Escala para a síntese de código. Por fim, o estudo concentra-se na análise crítica do comportamento dos alvos de teste frente a essas entradas e na documentação da efetividade da ferramenta na identificação de comportamentes irregulares ou indevidos.

2. REVISÃO BIBLIOGRÁFICA

2.1 CONTRATOS INTELIGENTES

O conceito de contratos inteligentes (*smart contracts*) antecede o surgimento das criptomoedas modernas, embora tenha sido popularizado por elas. A ideia foi introduzida na década de 1990 pelo criptógrafo Nick Szabo, que os definiu como protocolos de transação computadorizados capazes de executar automaticamente os termos de um acordo entre partes (SZABO, 1996). A implementação prática dessa proposta, porém, permaneceu limitada por décadas, devido à ausência de uma infraestrutura digital capaz de garantir execução segura, imutável e descentralizada.

Esse cenário mudou com o surgimento da rede Ethereum, que introduziu a Ethereum Virtual Machine, uma máquina virtual Turing-completa que ampliou significativamente as possibilidades de programação em blockchain ao permitir a implementação de lógicas arbitrárias e de qualquer complexidade, transformando a rede em um ambiente completo para aplicações distribuídas (WOOD et al., 2014). Nesse contexto, os contratos inteligentes destacam-se pela combinação entre imutabilidade e autoexecução: uma vez implantado na blockchain, seu código torna-se invariável, e sua execução é garantida pela rede de validadores, eliminando a necessidade de intermediários.

2.2 SOLIDITY

A linguagem de programação Solidity, proposta inicialmente em 2014 pela equipe do Ethereum, foi criada para desenvolver contratos inteligentes que podem ser executados na EVM. Desde sua concepção, o objetivo era fornecer uma linguagem de alto nível que simplificasse o desenvolvimento para a blockchain, evitando a complexidade da programação diretamente em bytecode ou assembly da EVM. Essa linguagem se destaca por ser uma mistura de convenções de redes, linguagem assembly e desenvolvimento web, o que a torna uma ferramenta robusta e versátil para o ecossistema blockchain (DANNEN, 2017).

Além disso, Solidity possui uma sintaxe inspirada em linguagens amplamente utilizadas, como JavaScript, Python e C++, o que facilita a curva de aprendizado para desenvolvedores de diferentes origens. A forma como essa linguagem foi criada, com tipagem estática, ajuda bastante na segurança dos contratos, possibilitando detectar

erros de tipo em tempo de compilação. O modelo de programação orientada a contratos da linguagem permite a criação de estruturas que encapsulam lógica e dados, garantindo que tudo funcione de forma previsível e exata na blockchain. Por tudo isso, Solidity se consolidou como a linguagem mais popular para escrever contratos inteligentes na rede Ethereum (PINNA et al., 2019), sendo fundamental para o desenvolvimento de aplicativos descentralizados (DApps) e para o mundo das finanças descentralizadas (DeFi), que estão mudando a forma como as pessoas interagem com dinheiro e serviços.

Dado o amplo uso do Solidity no desenvolvimento de contratos inteligentes, torna-se essencial garantir que o compilador responsável por traduzir esses contratos para linguagem de máquina funcione de forma correta e eficiente. O compilador Solidity (solc) é o responsável por converter o código-fonte em bytecode executável pela EVM, e qualquer falha nesse processo pode resultar em vulnerabilidades ou comportamentos inesperados nos contratos implantados. A segurança é um pilar fundamental no desenvolvimento com Solidity, pois os contratos inteligentes, uma vez implantados na blockchain, são imutáveis e operam em ambientes públicos e descentralizados. Erros ou vulnerabilidades podem levar a perdas financeiras significativas, como demonstrado por diversos incidentes históricos. Portanto, a adoção de boas práticas de segurança, como testes unitários e de integração, auditorias independentes de código e a aderência a padrões de design seguros, é imperativa para reduzir riscos e construir sistemas confiáveis e de qualidade.

O contínuo desenvolvimento da linguagem e de suas ferramentas auxiliares, juntamente com a crescente maturidade da comunidade de desenvolvedores, aponta para um futuro promissor para Solidity como a linguagem central do desenvolvimento blockchain na Ethereum e em outras EVM-compatíveis.

Exemplo de contrato inteligente em Solidity que implementa um contador simples:

```
pragma solidity ^0.8.20;

contract Counter {
    uint256 private counter;
    address public owner;

    event Incremented(uint256 newValue);

    constructor() {
        owner = msg.sender;
```

```
        counter = 0;
    }

    function increment() public {
        counter += 1;
        emit Incremented(counter);
    }

    function reset() public {
        require(msg.sender == owner, "Apenas o dono pode resetar o contador!");
        counter = 0;
    }

    function getCounter() public view returns (uint256) {
        return counter;
    }
}
```

2.3 FUZZING

O fuzzing é uma técnica de teste de software automatizada que envolve a execução repetida de um programa com entradas sintaticamente ou semanticamente inválidas para identificar falhas (MANÈS et al., 2019). O seu objetivo principal é gerar vários casos de teste que, ao serem executados pelo programa alvo, revelam erros e vulnerabilidades (LIANG et al., 2018).

Com o avanço da capacidade computacional, o fuzzing consolidou-se como uma forma poderosa para a descoberta de falhas e vulnerabilidades em softwares (PENG; SHOSHITAISHVILI; PAYER, 2018). Embora existam diversas abordagens de fuzzing, com diferentes métodos utilizados na geração dos casos de teste. Podemos encaixar esses métodos em três categorias:

- *Black-box fuzzing.* Também chamado de teste aleatório em caixa-preta, o black-box fuzzing é uma abordagem tradicional que não exige conhecimento prévio sobre o programa alvo ou o formato detalhado de suas entradas. Nesse método, os arquivos de entrada são modificados aleatoriamente com o intuito de criar casos de teste, que são enviados ao sistema em busca de falhas. Uma limitação

dessa técnica é que, por não ter uma visibilidade do código-fonte, os casos de teste resultarão em uma baixa cobertura de código (MANÈS et al., 2019).

- *White-box fuzzing.* É uma forma de geração automática de testes dinâmicos que se baseia em execução simbólica e na resolução de restrições (GODEFROID; KIEZUN; LEVIN, 2008). Esse método é projetado para a análise de segurança de aplicações de grande porte, pois permite explorar caminhos internos do programa de maneira sistemática. Ao contrário do black-box, o white-box fuzzing utiliza o conhecimento do código-fonte para gerar entradas que exercitam partes específicas do software, aumentando significativamente a probabilidade de identificar vulnerabilidades ocultas ou erros lógicos complexos.
- *Gray-box fuzzing.* Combina características do black-box e do white-box, usando informações parciais do programa, como cobertura de código e fluxo de dados, para guiar a geração de testes. Por meio de instrumentação, ele adapta as mutações para explorar mais caminhos ou achar bugs mais rápido, mas sem examinar diretamente o código-fonte como o white-box faz. Assim, consegue reduzir a aleatoriedade do black-box e tornar o processo de teste mais eficiente, embora sem garantir explorar todos os caminhos possíveis (MANÈS et al., 2019).

Em resumo, o fuzzing se mostra uma técnica muito interessante na engenharia de software, fornecendo meios eficazes de automatizar o processo de identificação de erros e vulnerabilidades nos softwares. A evolução das estratégias de fuzzing, desde as abordagens black-box mais simples até técnicas mais sofisticadas, como white-box e gray-box, contribui para o aumento da robustez dos softwares atuais, pois, ao aplicar essa técnica, é possível qualificá-los, tornando-os mais seguros e confiáveis.

2.4 TESTES DIFERENCIAIS NO COMPILADOR

Os compiladores são responsáveis por traduzir código de alto nível das linguagens para código que pode ser executado por computadores. (KOSSATCHEV; POSYPKIN, 2005). Por conta disso, eles são ferramentas fundamentais no ciclo de desenvolvimento de software, visto que constituem a base para a construção de praticamente todo o ecossistema de software. Quase todos os programas que executamos em um computador passam por algum tipo de processamento por compiladores ou ferramentas equivalentes (CHEN et al., 2020).

Por conta dessa participação no ciclo de desenvolvimento, é importante que os compiladores, mais do que qualquer outro software, sejam confiáveis e sem falhas. Pois, um bug no compilador pode gerar código binário incorreto a partir de um código-fonte válido, fazendo com que qualquer aplicação construída sobre esse binário esteja sujeita a falhas e comportamentos inesperados. Dessa forma, um único erro no compilador tem o potencial de se propagar por toda a cadeia de software que depende dele, comprometendo a confiabilidade do sistema como um todo (CHEN et al., 2020).

Esse risco é ainda mais crítico no contexto de contratos inteligentes, onde a garantia do funcionamento correto e da segurança é um dos pilares do desenvolvimento blockchain. Dada a imutabilidade e o potencial para perdas financeiras significativas em caso de falhas, a fase de testes assume uma importância ainda maior. Para atender a essas demandas rigorosas de confiabilidade, torna-se necessário, além da inspeção humana, explorar estratégias de teste mais robustas e automáticas.

Nesse contexto, os testes diferenciais surgem como uma alternativa viável. Na aplicação de testes diferenciais, o mesmo caso de teste é submetido a múltiplos programas equivalentes, como por exemplo, diferentes compiladores, para que seja possível comparar os resultados e analisar possíveis divergências. Essas divergências nos resultados indicam a provável presença de um bug. Embora poucas diferenças apareçam em softwares estáveis, a possibilidade de gerar milhões de testes faz com que mesmo raras discrepâncias revelem uma quantidade significativa de falhas (MCKEEMAN, 1998).

Neste trabalho, a técnica de testes diferenciais será aplicada ao contexto do ecossistema Ethereum, utilizando contratos inteligentes escritos em Solidity como casos de teste para verificar o comportamento do compilador. A ferramenta desenvolvida busca automatizar a geração desses contratos, explorando amplamente as possibilidades sintáticas e semânticas do Solidity, com o objetivo de induzir comportamentos inesperados durante a compilação. Dessa forma, ao comparar a saída do compilador sob diferentes configurações ou versões, é possível identificar inconsistências que poderiam comprometer a confiabilidade dos contratos implantados na blockchain, contribuindo assim para um ambiente mais seguro e robusto.

3. TRABALHOS RELACIONADOS

3.1 TESTES NO COMPILADOR SOLIDITY POR MEIO DE MUTAÇÃO SENSÍVEL À SINTAXE

(MITROPOULOS et al., 2023) utilizaram o conceito de *syntax-aware mutation* para testar o compilador solc, o compilador oficial da linguagem Solidity. Esse conceito foi aplicado para desenvolver o fuzzol, uma ferramenta de fuzzing que gera casos de teste preservando a validade sintática dos contratos. Para atingir isso, três estratégias de mutação foram aplicadas:

- A primeira altera o fluxo de controle dos programas, modificando comandos, operadores e tipos de dados.
- A segunda combina características de dois contratos distintos, realizando permutações nos nós folha de suas árvores sintáticas (AST), o que permite gerar variantes estruturalmente válidas, mas semanticamente diversas.
- A terceira foca em trechos escritos em inline assembly, realizando mutações específicas para estressar o otimizador inline do compilador solc.

Além dessas estratégias, o fuzzol incorpora um algoritmo de priorização que avalia, dinamicamente, a efetividade de cada estratégia aplicada a um contrato específico. Apenas as estratégias mais bem avaliadas são usadas nas próximas execuções, o que torna o processo de fuzzing mais eficiente e focado na exploração de caminhos promissores.

Essa abordagem resultou em um fuzzer híbrido que combina técnicas sensíveis e não sensíveis à sintaxe. Para validar sua eficácia, compararam o fuzzol com quatro outros fuzzers de referência, utilizando 33 versões diferentes do compilador. Como resultado, foram encontrados 19 bugs distintos, demonstrando a capacidade do fuzzol em revelar falhas reais no compilador Solidity.

Diferentemente do fuzzol, que gera casos de teste a partir da mutação sintática de contratos existentes usando três estratégias principais (alteração de fluxo de controle, combinação de ASTs e mutações em inline assembly), o SolForge adota uma abordagem de geração baseada em Modelos de Linguagem de Grande Escala. Além disso, enquanto o fuzzol se concentra exclusivamente no compilador oficial solc por meio de um fuzzer híbrido com priorização dinâmica, o SolForge executa testes

diferenciais entre compiladores, comparando simultaneamente o comportamento do solc e do solang. Isso permite identificar não apenas bugs isolados, mas também divergências de implementação entre diferentes compiladores.

3.2 GERAÇÃO ALEATÓRIA CONTROLADA DE PROGRAMAS PARA TESTE DE COMPILADORES E ANALISADORES DE SOLIDITY

O trabalho de (MA et al., 2025) propõe uma técnica voltada especificamente para a geração de programas Solidity com foco na detecção de bugs relacionados a compiladores e analisadores estáticos da linguagem.

Os autores partem da constatação de que geradores de código aleatório convencionais sofrem com um problema em que programas são gerados sem foco específico dentro do vasto espaço de busca definido pela linguagem. Como consequência, casos de teste potencialmente capazes de desencadear falhas acabam sendo pouco explorados ou adiados por dependerem de eventos aleatórios improváveis. Embora abordagens baseadas em mutações e métodos baseados em templates possam atenuar esse problema, elas também apresentam limitações, como a dependência de seeds bem escolhidos ou de templates manuais restritos.

Para superar essas limitações, os autores propõem a técnica de geração aleatória controlada de programas, que combina a aleatoriedade da geração de templates com a sistematicidade da enumeração exaustiva de valores para placeholders relacionados a atributos críticos da linguagem. Essa abordagem é dividida em dois estágios: (i) geração aleatória de templates de programas incompletos contendo lacunas em locais estrategicamente selecionados; e (ii) exploração exaustiva, mas limitada, das possíveis combinações válidas de preenchimento dessas lacunas, com o objetivo de explorar de forma controlada diferentes variações de programas com potencial para revelar falhas.

A técnica foi implementada na ferramenta Erwin, voltada à linguagem Solidity. Em testes realizados ao longo de seis meses, Erwin foi capaz de identificar 23 novos bugs em compiladores (solc, solang) e em um analisador estático (Slither), além de superar ferramentas como ACF e Fuzzol tanto na quantidade quanto na diversidade dos bugs detectados.

Enquanto o Erwin utiliza geração aleatória controlada baseada em templates incompletos com lacunas estrategicamente posicionadas, seguida por enumeração exaustiva mas limitada dos possíveis preenchimentos, o SolForge emprega uma abordagem fundamentalmente diferente ao usar modelos pré-treinados para geração

de contratos completos. A principal distinção está na ausência de dependência de templates pré-definidos ou enumeração sistemática de valores. O SolForge permite que o modelo crie contratos de forma mais orgânica e menos estruturada, guiado por prompts customizados, o que elimina as limitações de escalabilidade e qualidade de templates enfrentadas pelo Erwin, embora introduza diferentes desafios relacionados à controlabilidade da geração.

3.3 TESTE DIFERENCIAL DO COMPILADOR SOLIDITY POR MEIO DE MANIPULAÇÃO E MUTAÇÃO PROFUNDA DE CONTRATOS

O trabalho de (TIAN et al., 2024) visou explorar possíveis defeitos no compilador Solidity (solc) por meio da geração automática de smart contracts e testes diferenciais. Para tal, eles propuseram o DeSCDT, uma ferramenta projetada para descobrir falhas de compilação e inconsistências do compilador através do código gerado, alinhando-se com o escopo do trabalho proposto, que também busca empregar Modelos de Linguagem de Grande Escala para gerar código Solidity e identificar vulnerabilidades, bugs e crashes.

O DeSCDT, um framework para testar compiladores Solidity através de código gerado, consiste em três módulos principais. Primeiramente, o módulo de treinamento offline constrói um modelo generativo baseado na arquitetura Transformer. Esse modelo foi treinado com um conjunto inicial de contratos inteligentes diversificado, que foram selecionados com base em critérios semânticos, utilizando técnicas de codificação semântica e agrupamento para identificar e organizar exemplos com comportamentos semelhantes. Esses contratos foram obtidos e pré-processados a partir de fontes reais, principalmente o Etherscan (ETHERSCAN. . . ,). O pré-processamento inclui a remoção de comentários e espaços em branco desnecessários para focar na sintaxe central do Solidity.

No módulo de geração de contratos, os contratos iniciais, que foram codificados semanticamente e agrupados em clusters, são manipulados usando o modelo generativo treinado. Esse processo garante a geração de contratos sintaticamente válidos e diversos por meio de estratégias predefinidas e mutações detalhadas. As mutações continuam a alterar minimamente o código do contrato para preservar a validade sintática, incentivando simultaneamente a exposição de potenciais erros.

Por fim, com o intuito de identificar possíveis falhas no processo de otimização do compilador Solidity, os autores desenvolveram o último módulo do projeto adotando uma abordagem baseada em testes diferenciais. Essa técnica consiste em

compilar o mesmo código-fonte de um contrato em duas versões, uma com otimizações ativadas e outra sem, e comparar os resultados de execução para uma mesma entrada. Os contratos que são compilados foram analisados de forma mais detalhada em uma EVM focada em otimização, comparando o consumo de gás, tamanhos de opcodes e resultados computacionais entre versões otimizadas e não otimizadas. Qualquer divergência observada entre os comportamentos das duas versões é interpretada como um indício de uma otimização incorreta, que altera indevidamente o funcionamento do contrato.

Os experimentos demonstram que o DeSCDT se destaca na geração de contratos Solidity sintaticamente válidos, atingindo uma taxa de sucesso de 90,8%, e produz uma diversidade significativa de contratos, ampliando a cobertura dos testes. Em execuções contínuas, o modelo identificou seis bugs distintos capazes de causar falhas críticas no compilador, além de inconsistências na otimização.

O SolForge diferencia-se do DeSCDT em aspectos fundamentais de sua arquitetura e estratégia de testes. Enquanto o DeSCDT realiza treinamento offline de um modelo Transformer customizado, aplica clustering semântico de contratos reais da Etherscan e executa testes diferenciais comparando versões otimizadas e não-otimizadas do solc, o SolForge utiliza Modelos de Linguagem pré-treinados sem necessidade de treinamento específico, dispensa técnicas de clustering semântico, e implementa testes diferenciais cross-compiler, comparando compiladores diferentes. Conforme dito anteriormente, essa abordagem permite ao SolForge detectar não apenas inconsistências de otimização dentro de um mesmo compilador, mas também divergências comportamentais entre implementações distintas da especificação Solidity, além de requerer significativamente menos infraestrutura de preparação e treinamento.

3.4 GERADOR DE PROGRAMAS ALEATÓRIOS EM SOLIDITY PARA TESTES DE COMPILADOR

Em (LI; LIU; YU, 2025) os autores arquitetaram o Solsmith, uma ferramenta que gera código Solidity válido e diversificado com o intuito de descobrir falhas no compilador da linguagem. Para a criação desse software, os autores definiram três objetivos principais para alcançar, sendo eles, consistência, diversidade e conformidade com a especificação da linguagem Solidity.

A implementação do Solsmith envolve a construção do ambiente do contrato inteligente, a geração da estrutura da função e o preenchimento de declarações com expressões, loops, condicionais e blocos de assembly inline.

(LI; LIU; YU, 2025) adotaram uma estratégia de cross-optimization para realizar os testes diferenciais utilizando o módulo de otimização e de geração de bytecode do compilador solc. Ou seja, eles executaram os programas de teste gerados pelo Solsmith sob diferentes configurações do compilador e compararam os resultados obtidos, de forma que, caso haja discrepâncias nos resultados ou o compilador apresente falhas, isso pode indicar problemas em configurações específicas.

Os resultados experimentais, obtidos através da análise dos testes diferenciais, revelaram quatro defeitos confirmados no compilador Solidity versão 0.8.0. Com isso, (LI; LIU; YU, 2025) concluíram que o Solsmith é eficaz na geração de programas de teste e na descoberta de defeitos, com trabalhos futuros focados em explorar tipos de variáveis mais diversos, relacionamentos de herança de contratos mais complexos e testes sob uma maior variadade de combinações de otimização.

Diferentemente do Solsmith, que gera código Solidity através de um processo estruturado e algorítmico em três etapas (construção do ambiente do contrato, geração da estrutura de funções, preenchimento com declarações e expressões) e foca em testes diferenciais com base apenas no módulo de otimização do compilador solc, o SolForge emprega Modelos de Linguagem de Grande Escala para geração de contratos de forma mais simples. Além disso, o SolForge adota uma estratégia de testes diferenciais cross-compiler em vez de somente comparar diferentes configurações de otimização do mesmo compilador, permitindo identificar divergências entre implementações independentes da especificação Solidity.

3.5 TESTE DIFERENCIAL DA MÁQUINA VIRTUAL ETHEREUM NO NÍVEL DE OPCODE COM APOIO DE MODELOS DE LINGUAGEM DE GRANDE ESCALA

O trabalho de (MA et al., 2025) introduz o OpDiffer, um framework de testes diferenciais para a EVM, cuja principal contribuição é a geração automatizada de casos de teste semanticamente válidos no nível de opcodes, a menor unidade de execução na EVM. O objetivo do trabalho é identificar comportamentos inconsistentes entre diferentes implementações da EVM, explorando potenciais falhas que não são detectadas por técnicas tradicionais.

Para avaliar o funcionamento da ferramenta, (MA et al., 2025) estruturaram o trabalho em três etapas principais:

- *Geração de entradas de teste*. Utilizam Modelos de Linguagem de Grande Escala para gerar trechos de bytecode EVM semanticamente corretos, garantindo cenários reais e representativos.
- *Testes diferenciais*. Os testes são executados comparando diferentes implementações da EVM dentro de um ambiente unificado, com o mesmo fork da EVM e estados sincronizados, assegurando que divergências sejam atribuídas a diferenças nas implementações e não ao ambiente de execução.
- *Análise de inconsistências*. São usadas três métricas principais para identificar inconsistências entre as EVMs: **dados de retorno** (divergências indicam erros nos resultados), **uso de gás** (diferenças podem indicar variações nos caminhos de execução ou problemas nos opcodes) e **armazenamento** (alterações inesperadas apontam falhas no estado persistente). As anomalias detectadas são reproduzidas para validar sua consistência, e, ao identificar um bug, o OpDiffer aplica um algoritmo que localiza o opcode e a fase de execução associada à divergência, utilizando Modelos de Linguagem de Grande Escala para rastrear a origem do erro no código-fonte da EVM.

A aplicação do OpDiffer revelou 26 bugs previamente desconhecidos em nove diferentes implementações da EVM. Desses, 22 já foram confirmados ou corrigidos. Além disso, (MA et al., 2025) demonstraram que 7,21% dos contratos inteligentes existentes na blockchain podem ativar esses bugs em determinadas configurações.

O SolForge distingue-se do OpDiffer pelo nível de abstração e alvo dos testes. Enquanto o OpDiffer opera no nível de bytecode/opcodes da EVM, gerando sequências de opcodes semanticamente válidas usando Modelos de Linguagem de Grande Escala e testando diferentes implementações da Ethereum Virtual Machine em busca de inconsistências de execução, o SolForge atua no nível de código-fonte Solidity, gerando contratos completos e testando compiladores da linguagem em busca de bugs de compilação, crashes e divergências de otimização. Portanto, os trabalhos têm focos complementares: o OpDiffer busca falhas na camada de execução da EVM, enquanto o SolForge visa identificar problemas na camada de compilação de Solidity para bytecode EVM.

4. DESENVOLVIMENTO

Este capítulo descreve o processo de desenvolvimento do SolForge, uma ferramenta de fuzzing cujo objetivo é encontrar bugs em compiladores Solidity. A ferramenta foi construída para automatizar a geração de contratos inteligentes Solidity (os casos de teste), através do uso de Modelos de Linguagem de Grande Escala e, subsequentemente, aplicar técnicas de testes diferenciais nos compiladores.

4.1 Tecnologias e Ambiente de Desenvolvimento

A etapa inicial do desenvolvimento do SolForge consistiu na configuração do ambiente de desenvolvimento e na instalação das ferramentas necessárias, conforme listado abaixo:

- **Containerização: Docker**. Utilizado para criar um ambiente isolado e replicável do projeto, garantindo que o sistema operacional e as versões das dependências sejam as mesmas, independentemente de onde o projeto está sendo executado (MERKEL, 2014).
- **Linguagem de Programação: Haskell (GHC 9.6)**. Uma linguagem de programação de uso geral, estaticamente tipada, puramente funcional, com inferência de tipos e avaliação preguiçosa (MARLOW et al., 2010). A escolha desta linguagem foi motivada pelo interesse no aprofundamento do paradigma funcional. Além disso, a decisão considerou a expertise do orientador do trabalho nesta tecnologia.
- **Gerenciamento de Build: Cabal**. Utilizado para o gerenciamento de dependências e automação do processo de compilação dos arquivos Haskell (Haskell Cabal Development Team, 2025).
- **Automação de Tarefas: GNU Make**. Utilizado para abstrair a complexidade do ambiente de desenvolvimento. O Makefile (STALLMAN; MCGRATH; SMITH, 2002) serve como ponto de entrada para as operações do sistema, simplificando a execução dos comandos de geração de prompts, geração de contratos inteligentes e execução dos testes diferenciais.
- **Servidor de Modelos de Linguagem de Grande Escala: Ollama**. Utilizado para servir localmente os modelos de linguagem (Ollama Team, 2025), fornecendo uma API REST para a aplicação Haskell.

- **Modelo de Linguagem de Grande Escala: DeepSeek Coder V2**. Um modelo com 16 bilhões de parâmetros especializado em tarefas de programação, selecionado por sua arquitetura otimizada para geração de código e por ser atual (ZHU et al., 2024).
- **Compiladores: Solidity Compiler (`solc`)** e **Solang**. Ambos (Ethereum Foundation, 2025; Hyperledger Foundation, 2025) foram utilizados em conjunto para a realização de testes diferenciais, permitindo a comparação de comportamentos e detecção de divergências na compilação dos contratos inteligentes gerados.

4.2 Módulo de Geração de Prompts

Para garantir a eficácia do processo de fuzzing, o SolForge implementa um módulo de geração aleatória de prompts com o intuito de gerar grandes quantidades de instruções que cubram diferentes aspectos do Solidity. Ele é responsável por criar instruções de linguagem natural únicas e diversificadas, que servirão de entrada para o Modelo de Linguagem de Grande Escala na etapa de geração de contratos inteligentes.

4.2.1 Arquitetura e Sistema de Tokens

A geração baseia-se em um sistema combinatório que gerencia quatro categorias distintas de tokens. Cada categoria corresponde a um bloco estrutural fundamental da instrução, delimitando o escopo do código solicitado. O módulo seleciona aleatoriamente um elemento de cada conjunto para compor uma instrução complexa e semanticamente coerente.

As categorias de tokens definidas são:

1. **Objetivos:** Define o propósito funcional do contrato (ex: "um sistema de votação", "um mecanismo de leilão"). O sistema dispõe de 24 definições base.
2. **Estruturas de Dados:** Especifica os tipos de dados que devem ser utilizados (ex: "mappings aninhados", "arrays dinâmicos"). Conta com 22 variações.
3. **Lógica Computacional:** Determina que funcionalidade da linguagem deve ser implementada (ex: "operações bitwise", "loops iterativos", "manipulação de assembly"). Possui 28 padrões lógicos.

4. **Restrições:** Define condições que o programa deve seguir (ex: "declara somente variáveis imutáveis", "requer quantidade exata de gas para uma determinada chamada"). Totaliza 26 restrições.

4.2.2 Algoritmo de Geração e Singularidade

A implementação em Haskell utiliza a recursividade e estruturas de dados imutáveis para garantir a singularidade dos prompts gerados. O algoritmo trabalha preenchendo um template de linguagem natural estruturado da seguinte forma:

> `"Create a Solidity contract that implements [OBJETIVO]. The contract MUST use [ESTRUTURA DE DADOS]. Crucially, the logic [LÓGICA COMPUTACIONAL]. Finally, ensure the contract [RESTRIÇÃO]."`

Para assegurar que não haja instruções duplicadas no conjunto de dados de entrada, o sistema utiliza a estrutura de dados `Set` (conjunto), que não permite elementos repetidos. O fluxo lógico do algoritmo segue os passos:

1. O sistema recebe como parâmetro a meta de prompts desejada (ex: 100);
2. Uma função de seleção aleatória escolhe um token de cada uma das quatro listas;
3. Os tokens são concatenados no template;
4. A string resultante é inserida no Set;
5. O processo recursivo continua até que a quantidade de elementos presentes no Set seja igual à meta de prompts estabelecida.

4.2.3 Análise do Espaço de Busca

A abordagem combinatória foi escolhida como uma tentativa de cobrir mais funcionalidades da linguagem Solidity para testes do compilador. A diversidade potencial do sistema pode ser calculada pelo produto cartesiano das categorias de tokens disponíveis:

$$\text{Total de Combinações} = |\textit{Objetivos}| \times |\text{Estruturas de Dados}| \times |\text{Lógica}| \times |\text{Restrições}|$$

$$\text{Total} = 24 \times 22 \times 28 \times 26 = 384.384 \text{ prompts únicos}$$

Este vasto espaço de possibilidades assegura que o sistema seja capaz de submeter os compiladores `solc` e `solang` a uma ampla variedade de cenários sintáticos e semânticos, desde estruturas de dados triviais até lógicas mais complexas de baixo nível, aumentando a probabilidade de detecção de divergências ou falhas internas.

4.2.4 Execução e Persistência

A execução do gerador é realizada via linha de comando, onde define-se a quantidade de instruções a serem criadas. O processo é acionado através do *Makefile*, utilizando o comando:

```
make prompts count=100
```

Após o processamento, os prompts são salvos em um arquivo de texto (`data/prompts.tx`
Nesse arquivo, cada linha corresponde a um prompt completo e único, pronto para ser utilizado pelo módulo de geração de contratos.

4.3 Módulo de Geração de Contratos

Nessa etapa, foi implementado o módulo gerador de contratos inteligentes Solidity. Este módulo opera por meio da orquestração de dois serviços distintos e interdependentes, definidos no ambiente Docker:

- `ollama-service`**:** Serviço responsável pela execução do servidor de inferência local, expondo o modelo `deepseek-coder-v2:16b` através de uma API REST.
- `haskell-app`**:** O núcleo da aplicação, desenvolvido em Haskell. Este serviço atua como controlador, sendo responsável carregar os *prompts*, gerenciar a comunicação HTTP com o Ollama, processar as respostas textuais e persistir os contratos inteligentes gerados.

A arquitetura do módulo foi projetada para operar em um ciclo contínuo de geração e validação, assegurando a produção de contratos robustos e sintaticamente

corretos. O fluxo de execução, controlado pelo arquivo de configuração *ollama-config.json*, segue um algoritmo que é executado até que a meta predefinida de contratos válidos seja alcançada. O processo é apresentado a seguir:

1. **Inicialização:** O sistema carrega um conjunto de *prompts* que cobre diversas funcionalidades da linguagem Solidity a partir do arquivo `prompts.txt`. Estes prompts são organizados em uma estrutura de lista cíclica, permitindo que um número finito de instruções alimente uma geração contínua, explorando o não-determinismo do modelo para criar variações de código.

2. **Requisição e Inferência:** A aplicação seleciona o próximo *prompt* da fila e envia uma requisição POST para o serviço `ollama-service`. O modelo, condicionado por regras estabelecidas em seu System Prompt (Modelfile), retorna um objeto JSON contendo o texto gerado.

3. **Requisição e Inferência:** A aplicação seleciona o próximo prompt da fila e envia uma requisição POST para o serviço `ollama-service`. A inferência é governada pelo *Modelfile*, um arquivo de configuração que define os parâmetros do modelo e o *System Prompt*. Neste trabalho, o *System Prompt* foi projetado para impor restrições comportamentais ao modelo, instruindo ele a atuar como um especialista em Solidity e a retornar exclusivamente o código-fonte solicitado, minimizando a ocorrência de texto conversacional ou explicativo na resposta.

4. **Processamento e Extração:** Após a validação do status HTTP (200 OK), o módulo realiza o *parsing* da resposta JSON. Foi implementada uma lógica de extração que identifica e isola o bloco de código Solidity, removendo marcadores de Markdown e textos conversacionais excedentes através da função de limpeza de strings, caso existam.

5. **Validação Sintática e Persistência:** O código extraído é salvo temporariamente e submetido imediatamente ao compilador `solc`. Nesta etapa, o compilador não é utilizado para testes diferenciais, mas como um validador de sintaxe:

    - **Sucesso (Exit Code 0):** O contrato é classificado como válido, movido para o diretório `contracts/valid/`, e o contador de progresso é incrementado.
    - **Falha (Exit Code 1):** O contrato contendo erros de sintaxe é movido para o diretório `contracts/invalid/`, sem incrementar a meta.

6. **Logs:** Para viabilizar uma análise estatística, cada iteração é registrada no arquivo `data/logs/contract-generation.csv`. O registro foi padronizado para capturar métricas de desempenho e rastreabilidade, incluindo:

- **Rastreabilidade:** Identificação sequencial (`attempt_number`), temporal (`timestamp`) e vínculo com a instrução geradora (`prompt_id`);
- **Performance:** Tempo de inferência do modelo em milissegundos (`generation_time_ms`
- **Características do Código:** Contagem de linhas (`line_count`) e *hash* do conteúdo (`contract_hash`) para garantia de integridade e detecção de duplicatas.

Uma decisão de projeto fundamental nesta etapa foi optar pelo descarte imediato dos contratos inválidos, ao invés da implementação de um mecanismo de correção automática (como o reenvio do erro ao modelo). Esta estratégia baseia-se em três justificativas principais:

a) **Eficiência Computacional:** A criação de novos contratos é muito mais rápida do que o processo complexo de revisar e corrigir os antigos, levando em conta que o modelo pode causar um novo erro a cada solicitação de corrigir o código.

b) **Prioridade de Volume e Variedade:** Para fins de *fuzzing*, a diversidade de estruturas é essencial. A natureza não-determinística do modelo assegura que, mesmo repetindo os mesmos prompts ciclicamente, novas variações de código sejam produzidas, compensando as instâncias descartadas;

c) **Delimitação de Escopo:** O objetivo do trabalho é avaliar o compilador Solidity e a geração de código, e não o desenvolvimento de um agente corretor de código baseado em Modelos de Linguagem de Grande Escala.

Esta abordagem assegura que o conjunto de dados final seja composto exclusivamente por programas compiláveis. Embora o descarte possa reduzir a frequência de estruturas sintáticas complexas que o modelo tem dificuldade em gerar, a repetição cíclica dos prompts maximiza a probabilidade de obter, eventualmente, amostras válidas para cada categoria funcional testada.

4.4 Módulo de Testes Diferenciais

Após a fase de geração dos contratos inteligentes, o sistema avança para a etapa mais importante do experimento: a execução dos testes diferenciais. Este módulo foi projetado para submeter os casos de teste a três configurações distintas de compiladores — `solc` (*standard*), `solc` (*optimized*) e `solang` — comparando seus comportamentos para identificar divergências de implementação e otimização.

4.4.1 Implementação em Haskell

A lógica de teste também foi desenvolvida usando o serviço *haskell-app*. O sistema gerencia a comparação submetendo cada contrato às três configurações de compilador sequencialmente e, em seguida, avalia o retorno de cada um individualmente para determinar o veredito final do status de compilação daquele contrato inteligente.

Status de Compilação

Representa o resultado individual de cada ferramenta:

- OK: Compilação sem falhas (exit 0);
- REJ: Código rejeitado por erro de sintaxe (exit 1);
- ERR: Erro interno do compilador (*Internal Compiler Error*).

O sistema orquestra a comparação submetendo cada contrato às três configurações sequencialmente e, em seguida, invoca o Oráculo para determinar o veredito final.

4.4.2 Lógica de Classificação

O oráculo é o componente responsável pela lógica de classificação. Ele agrega os status individuais em um único veredito. A estratégia de classificação prioriza a detecção de falhas críticas (*crashes*) sobre divergências comportamentais.

A Tabela 4.1 detalha as regras de transição utilizadas para definir cada categoria de resultado.

Tabela 4.1 – Matriz de Decisão do Oráculo

Veredito Final	**Significado**	**Condição Lógica**
CRASH	Erro Crítico	Pelo menos um compilador retornou `ERR`
PASS	Consenso Positivo	Todos os compiladores retornaram `OK`
REJECT	Consenso Negativo	Todos os compiladores retornaram `REJ`
DIVERGENCE	Discrepância	Resultados mistos entre `OK` e `REJ`

Nota: A detecção de ERR é realizada via análise de palavras-chave no stderr, como "internal compiler error", "panic"e "segmentation fault".

4.4.3 Infraestrutura de Dados e Logs

A arquitetura do sistema centraliza a persistência dos dados para facilitar a análise posterior, garantindo que cada execução do comando `make fuzz` limpe automaticamente os registros anteriores.

Relatório de Resultados

Os resultados são armazenados em `data/fuzzing/results.csv`. Os contratos são processados e registrados em ordem numérica natural para manter a sequência lógica. As colunas do arquivo são:

- FILENAME: Nome do contrato (sem extensão `.sol`);
- SOLC_STD / SOLC_OPT / SOLANG: Status individual de cada configuração;
- VERDICT: Classificação final concebida pelo oráculo.

Logs Detalhados de Diagnóstico

Para otimizar o armazenamento, o sistema gera logs individuais em `data/fuzzing/logs/` apenas para casos relevantes REJECT, DIVERGENCE e CRASH). Testes classificados como PASS não geram logs. O conteúdo inclui:

- Status de compilação e o código de saída do compilador;
- Mensagens de erro capturadas via *stderr*;

4.4.4 Fluxo de Execução

Os testes são executados a partir do envio do comando `make fuzz` via CLI, que orquestra as seguintes ações:

1. Limpeza: Remoção de resultados anteriores e logs antigos;
2. Descoberta: Localização e ordenação dos contratos em `data/contracts/valid/`;
3. Processamento Sequencial: Para cada contrato:

   (a) Compilação nas três configurações;
   (b) Determinação dos status individuais (`OK/REJ/ERR`);
   (c) Classificação via oráculo;
   (d) Persistência no CSV e geração condicional de logs.

4. Feedback: Exibição de resumo estatístico e progresso em tempo real no terminal.

5. RESULTADOS EXPERIMENTAIS E ANÁLISE

Após a implementação dos módulos do SolForge, foi conduzido um experimento controlado para avaliar a eficácia da ferramenta na geração de contratos sintaticamente válidos e na detecção de discrepâncias entre três configurações de compiladores: `solc` (v0.8.26 default), `solc` (v0.8.26 otimizado) e `solang` (v0.3.3).

Esta seção detalha as métricas obtidas e apresenta uma análise das divergências, focando em incompatibilidades arquiteturais e falhas de robustez.

5.1 Cenário Experimental

O experimento foi executado em ambiente isolado via Docker. A configuração de hardware e software utilizada para garantir a reprodutibilidade dos resultados é detalhada a seguir:

- **Hardware:** 1x NVIDIA RTX 6000 Ada (96GB VRAM), 188GB RAM, 16 vCPUs.
- **Modelo de Linguagem de Grande Escala:** DeepSeek Coder V2 (via Ollama).
- **Instruções:** 3.000 prompts gerados combinatoriamente.
- **Alvos de Teste:**
  - `solc v0.8.26`: Compilador oficial da Ethereum Foundation.
  - `solang v0.3.3`: Compilador alternativo da Hyperledger para Solana.

5.2 Avaliação da Performance da Geração

A primeira etapa avaliou a capacidade do modelo em produzir contratos válidos para o fuzzing. O processo operou sequencialmente até atingiro o marco de 1.357 contratos válidos. Número suficiente para executar os testes diferenciais.

A taxa de aproveitamento de 55,96% demonstra uma eficácia moderada, condizente com modelos genéricos sem *fine-tuning*. A validação sintática executada após a geração de cada caso de teste garantiu que apenas contratos compiláveis pelo `solc` (standard) avançassem para o teste diferencial, isolando erros de sintaxe de erros lógicos dos compiladores.

Tabela 5.1 – Eficiência da Geração e Validação Sintática

Métrica	**Valor**	**Detalhes**
Tempo Total	1h37	2,40 segundos por tentativa
Total de Tentativas	2.425	100,00%
Sintaticamente Válidos	1.357	55,96%
Descartados (Erro de Sintaxe)	1.068	44,04%

5.3 Resultados dos Testes Diferenciais

O *corpus* de 1.357 contratos foi submetido à execução dos corpos sob teste. A classificação pelo oráculo revelou a distribuição de comportamentos apresentada na Tabela 5.2.

Tabela 5.2 – Classificação dos Resultados pelo Oráculo

Veredito	**Significado**	**Qtd.**	**%**
PASS	Todos sem falhas	287	21,15%
REJECT	Todos com falhas	0	0,00%
DIVERGENCE	Discordância de Resultados	1.041	76,71%
CRASH	Erro Interno do Compilador	29	2,14%
Total		**1.357**	**100%**

Destaca-se a ausência de casos REJECT (0%), consequência direta da pré-validação sintática. As categorias de interesse (DIVERGENCE e CRASH) somam 78,85% do corpus, indicando uma alta taxa de descoberta de comportamentos de borda.

5.3.1 Decomposição Quantitativa das Falhas

Para compreender a origem das falhas, os resultados foram desagregados por padrão de comportamento dos compiladores.

Divergências (1.041 casos)

A quase totalidade das divergências (99,9%) segue o padrão `OK, OK, REJ`, onde o contrato é aceito pelo `solc` (ambas configurações) mas rejeitado pelo `solang`. Apenas um caso isolado apresentou divergência entre as versões do próprio `solc`.

Crashes (29 casos)

Os erros internos foram classificados conforme o compilador afetado (Tabela 5.3).

Tabela 5.3 – Distribuição dos Erros Internos (Crashes)

Padrão (Std, Opt, Solang)	Qtd.	Compilador Afetado
`OK, OK, ERR`	20	Exclusivo Solang
`ERR, ERR, REJ`	7	Exclusivo Solc
`ERR, ERR, OK`	1	Exclusivo Solc
`ERR, ERR, ERR`	1	Universal (Todos)

5.4 Análise Qualitativa e Discussão

A investigação dos logs permitiu categorizar as causas raízes dos comportamentos irregulares reportados na seção anterior.

5.4.1 Limitações Arquiteturais do Solang

As divergências significativas em que o solang rejeita contratos válidos (1.040 casos) não configuram bugs, mas refletem limitações decorrentes das diferenças de recursos disponíveis entre as máquinas virtuais de cada ecossistema blockchain: a EVM (Ethereum Virtual Machine) e o modelo BPF (Berkeley Packet Filter) utilizado pela Solana. Os principais fatores identificados foram:

1. **Inline Assembly (Yul):** ($\sim$73% das divergências). Instruções de baixo nível como `mload/mstore` não possuem tradução direta para a arquitetura Solana.
2. **Transferência de Valor:** O uso de `msg.value` falha pois o Solang exige o mecanismo de CPI (*Cross Program Invocation*) para transferência de ativos nativos.
3. **Variáveis de Bloco:** Propriedades como `block.difficulty` inexistem no modelo de consenso da Solana.

5.4.2 Defeitos Críticos no Ecossistema Ethereum (Solc)

Embora o `solc` seja considerado o padrão da indústria, o SolForge foi capaz de identificar falhas de robustez e otimização.

Bug de Otimização (1 caso)

O contrato `contract_2406` revelou uma divergência interna no `solc` (padrão `OK, REJ, REJ`). A causa foi o uso da instrução `msize` em conjunto com o otimizador Yul. O compilador bloqueia essa combinação para evitar inconsistências de memória, um comportamento documentado mas que gera um "falso positivo"de divergência interessante para análise de segurança.

Erros Internos do Compilador (8 casos)

Mais graves foram os 8 casos onde o `solc` abortou a execução com exceções não tratadas (Crashes). A análise do *stderr* revelou mensagens como:

```
Unimplemented feature: [...]
Copying of type struct Lottery.Player memory[] memory
to storage not yet supported.
```

Isso evidencia que certas combinações complexas de tipos (ex: cópia de arrays de structs da memória para storage) passam pela análise semântica mas falham na geração do código binário.

6. CONCLUSÃO

Este trabalho cumpriu seu objetivo geral ao desenvolver e validar a ferramenta SolForge, demonstrando a viabilidade da aplicação de Modelos de Linguagem de Grande Escala na automação de testes para o compilador Solidity. A pesquisa mostra que mesmo modelos sem treinamento específico prévio, são capazes de atuar como fuzzers eficazes, gerando entradas sintaticamente complexas e diversificadas.

O êxito na execução dos objetivos específicos evidenciou-se, sobretudo, na análise do comportamento do compiladores frente aos códigos gerados. A identificação de oito falhas críticas de interrupção no *solc* valida a relevância prática da ferramenta proposta, destacando sua contribuição direta para a segurança e robustez do ecossistema blockchain, uma infraestrutura que, como discutido, exige rigor extremo devido à imutabilidade e ao valor financeiro dos ativos envolvidos.

Contudo, tal restrição não diminui o mérito da abordagem; pelo contrário, aponta caminhos para trabalhos futuros focados na otimização de recursos e na escalabilidade da ferramenta, consolidando o uso de inteligência artificial como um aliado indispensável na garantia de qualidade de compiladores.

Nesse sentido, a evolução do SolForge não se limita apenas à infraestrutura, mas exige também o aprimoramento de sua robustez operacional. Uma implementação futura prioritária é a introdução de um sistema de checkpoints e persistência de estado. Na arquitetura atual, interrupções inesperadas ou falhas sistêmicas durante o processo de geração resultam na perda dos dados gerados, obrigando o reinício dos testes e o descarte dos contratos e logs já obtidos. A incorporação de mecanismos de salvamento incremental permitiria que a ferramenta retomasse a execução exatamente do ponto de parada, mitigando o desperdício de tempo computacional e garantindo a continuidade e a integridade das campanhas de testes de longa duração.

Em síntese, os resultados obtidos confirmam que a combinação entre modelos generativos e técnicas tradicionais de teste representa uma direção promissora para o desenvolvimento de ferramentas de análise e validação de compiladores.

REFERÊNCIAS BIBLIOGRÁFICAS

CHEN, J. et al. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 53, n. 1, p. 1–36, 2020.

DANNEN, C. Solidity programming. In: *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. [S.l.]: Springer, 2017. p. 69–88.

Ethereum Foundation. *Solidity: The Solidity Contract-Oriented Programming Language*. [S.l.], 2025. Versão 0.8.26. Disponível em: <https://docs.soliditylang.org/en/v0.8.26/>.

ETHERSCAN: The Ethereum Blockchain Explorer. <https://etherscan.io/>. Acessado em 25 de junho de 2025.

GODEFROID, P.; KIEZUN, A.; LEVIN, M. Y. Grammar-based whitebox fuzzing. In: *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. [S.l.: s.n.], 2008. p. 206–215.

Haskell Cabal Development Team. *Cabal User Guide*. [S.l.], 2025. Acessado em: dezembro de 2025. Disponível em: <https://cabal.readthedocs.io/>.

Hyperledger Foundation. *Solang: Solidity Compiler for Solana and Polkadot*. [S.l.], 2025. Versão 0.3.3. Disponível em: <https://solang.readthedocs.io/>.

KOSSATCHEV, A. S.; POSYPKIN, M. Survey of compiler testing methods. *Programming and Computer Software*, Springer, v. 31, n. 1, p. 10–19, 2005.

LI, L.; LIU, Z.; YU, Z. Solsmith: Solidity random program generator for compiler testing. *arXiv preprint arXiv:2506.03909*, 2025.

LIANG, H. et al. Fuzzing: State of the art. *IEEE Transactions on Reliability*, IEEE, v. 67, n. 3, p. 1199–1218, 2018.

LIANG, R. et al. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing. *IEEE Transactions on Information Forensics and Security*, IEEE, 2025.

MA, H. et al. Bounded exhaustive random program generation for testing solidity compilers and analyzers. *arXiv preprint arXiv:2503.20332*, 2025.

MA, H. et al. Towards understanding the bugs in solidity compiler. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2024. (ISSTA '24), p. 1312–1324. Disponível em: <http://dx.doi.org/10.1145/3650212.3680362>.

MA, J. et al. Opdiffer: Llm-assisted opcode-level differential testing of ethereum virtual machine. *arXiv preprint arXiv:2504.12034*, 2025.

MANÈS, V. J. et al. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, IEEE, v. 47, n. 11, p. 2312–2331, 2019.

MARLOW, S. et al. *Haskell 2010 language report*. [S.l.], 2010. Disponível em: <https://www.haskell.org/onlinereport/haskell2010/>.

MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal*, v. 10, n. 1, p. 100–107, 1998.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, Belltown Media, v. 2014, n. 239, p. 2, 2014.

MITROPOULOS, C. et al. Syntax-aware mutation for testing the solidity compiler. In: SPRINGER. *European Symposium on Research in Computer Security*. [S.l.], 2023. p. 327–347.

Ollama Team. *Ollama: Get up and running with large language models*. 2025. Software. Disponível em: <https://github.com/ollama/ollama>.

PENG, H.; SHOSHITAISHVILI, Y.; PAYER, M. T-fuzz: fuzzing by program transformation. In: IEEE. *2018 IEEE Symposium on Security and Privacy (SP)*. [S.l.], 2018. p. 697–710.

PINNA, A. et al. A massive analysis of ethereum smart contracts empirical study and code metrics. *Ieee Access*, IEEE, v. 7, p. 78194–78213, 2019.

POLITOU, E. et al. Blockchain mutability: Challenges and proposed solutions. *IEEE Transactions on Emerging Topics in Computing*, IEEE, v. 9, n. 4, p. 1972–1986, 2019.

STALLMAN, R. M.; MCGRATH, R.; SMITH, P. D. *GNU Make: A Program for Directing Recompilation*. [S.l.]: Free Software Foundation, 2002.

SZABO, N. Smart contracts: building blocks for digital markets. *Extropy*, The Extropy Institute, v. 16, n. 2, p. 18, 1996.

TIAN, Z. et al. Differential testing solidity compiler through deep contract manipulation and mutation. *Software Quality Journal*, Springer, v. 32, n. 2, p. 765–790, 2024.

TRIPATHI, G.; AHAD, M. A.; CASALINO, G. A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges. *Decision Analytics Journal*, Elsevier, v. 9, p. 100344, 2023.

WOOD, G. et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, v. 151, n. 2014, p. 1–32, 2014.

ZHU, Q. et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.