Explorando o Uso de LLMs para Fuzzing de Código Lua

Exploring the Use of LLMs for Lua Code Fuzzing

Richard Facin Souza [Universidade Federal da Fronteira Sul | *richard.souza@estudante.uffs.edu.br*]
Samuel da Silva Feitosa [Universidade Federal da Fronteira Sul | *samuel.feitosa@uffs.edu.br*]

Universidade Federal da Fronteira Sul, Chapecó-SC, 89815-899, Brasil.

Resumo. O *fuzzing* é uma técnica crucial para encontrar vulnerabilidades em software, mas sua eficácia em linguagens de script como Lua é limitada pela dificuldade de gerar entradas de teste semanticamente válidas. Este trabalho propõe uma metodologia que utiliza Modelos de Linguagem de Grande Escala (LLMs) para gerar mutações semanticamente ricas para o *fuzzing* de scripts Lua. Nossa abordagem envolve o desenvolvimento de um protótipo de fuzzer que explora a capacidade de aprendizado em contexto de um LLM para criar variações de código sintática e semanticamente plausíveis. Validaremos esta metodologia avaliando o ganho de cobertura de código das mutações em relação às sementes originais, a eficácia na identificação de bugs e o tempo de execução do processo, visando demonstrar a viabilidade do uso de LLMs na segurança de sistemas que utilizam Lua.

Abstract. Fuzzing is a crucial technique for finding vulnerabilities in software, but its effectiveness in scripting languages such as Lua is limited by the difficulty of generating semantically valid test inputs. This work proposes a methodology that uses Large Language Models (LLMs) to generate semantically rich mutations for fuzzing Lua scripts. Our approach involves developing a fuzzer prototype that leverages an LLM's in-context learning capabilities to create syntactically and semantically plausible code variations. We will validate this methodology by evaluating the code coverage gain of mutations relative to original seeds, the effectiveness in bug identification, and the execution time of the process, aiming to demonstrate the feasibility of using LLMs for the security of systems utilizing Lua.

Palavras-chave: Fuzzing, LLM, Lua, Teste de Software, Segurança

Keywords: Fuzzing, LLM, Lua, Software Testing, Security

1 Introdução

A linguagem Lua é conhecida por ser leve, eficiente e de fácil integração, sendo amplamente utilizada em áreas como desenvolvimento de jogos, sistemas embarcados e automação de tarefas [Lua.org, 2025; Ierusalimschy, 2016; Siberoloji, 2021]. Devido à sua popularidade e aplicabilidade, garantir a confiabilidade e a segurança de aplicações desenvolvidas em Lua é uma preocupação crescente, principalmente quando se considera o impacto potencial de falhas em ambientes críticos.

Uma das formas mais eficazes de identificar vulnerabilidades e falhas em softwares é por meio da técnica de *fuzzing*, uma abordagem que se destaca pela capacidade de gerar entradas inesperadas ou malformadas que exercitam diferentes caminhos de execução nos programas [Godefroid, 2020; Manès *et al*., 2021]. No entanto, o *fuzzing* tradicional, especialmente o baseado em mutação, enfrenta limitações importantes, como a dificuldade de gerar mutações semanticamente relevantes ou de explorar caminhos menos triviais no código [Huang *et al*., 2024; Eom *et al*., 2024].

Recentemente, os Modelos de Linguagem de Grande Escala (LLMs, do inglês *Large Language Models*) consolidaram-se como ferramentas promissoras na geração de código, incluindo a criação de casos de teste [Wang and Chen, 2023; Deckker and Sumanasekara, 2025]. No contexto do *fuzzing*, esses modelos apresentam um potencial significativo para melhorar a geração de entradas, permitindo mutações mais inteligentes e direcionadas, que consideram tanto a sintaxe quanto a semântica da linguagem [Huang *et al*., 2024; Xia *et al*., 2024]. Esse avanço abre espaço para superar limitações históricas das técnicas convencionais e explorar novas formas de automatização de testes.

Neste trabalho, propomos e avaliamos uma metodologia para a geração de mutações semanticamente ricas em *fuzzing*, utilizando LLMs como um motor de transformação de código. Diferentemente de abordagens tradicionais que se baseiam em alterações a nível de bytes, nossa técnica explora a capacidade dos LLMs de compreender a sintaxe e a semântica do código-fonte para criar variações complexas e válidas [Huang *et al*., 2024; Xia *et al*., 2024].

Para validar essa abordagem, desenvolvemos um protótipo de *looping* de *fuzzing* que aplica a mutação guiada por LLM especificamente à linguagem Lua. A escolha de Lua como caso de estudo se justifica por suas características dinâmicas e seu uso prevalente em sistemas críticos onde a segurança é primordial [Ierusalimschy, 2016; Marbux, 2021]. O objetivo central é validar a ferramenta analisando três aspectos principais: o ganho incremental de cobertura de código proporcionado pelas mutações em comparação às sementes originais, a capacidade efetiva de descoberta de erros (bugs) e o desempenho temporal (tempo de execução) da abordagem.

2 Fundamentação Teórica

Esta seção apresenta a base conceitual necessária para compreender a proposta do trabalho. São abordados a linguagem de programação Lua, Modelos de Linguagem de Grande Escala e da técnica de *fuzzing*, com foco no uso combinado de LLMs como apoio à mutação em testes automatizados.

2.1 Linguagem Lua

Lua é uma linguagem de programação interpretada, eficiente e leve, desenvolvida no Brasil na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Desde sua criação em 1993, destaca-se por sua simplicidade sintática, portabilidade e capacidade de extensão [Ierusalimschy, 2016, 2020; Lua.org, 2025]. Projetada para ser uma *glue language*, integra-se facilmente a programas em C/C++ e é amplamente usada em jogos, sistemas embarcados e aplicações de automação.

A linguagem é dinamicamente tipada e possui uma única estrutura de dados nativa, a *table*, que pode representar vetores, registros ou objetos com grande flexibilidade. Inclui ainda conceitos como funções de primeira classe, coleta de lixo automática e suporte a corrotinas, recursos que favorecem concorrência cooperativa e programação orientada a eventos [Ierusalimschy, 2016]. Além disso, seu design minimalista e modular contribui para um baixo consumo de memória e processamento, o que a torna adequada para dispositivos com recursos limitados e sistemas de tempo real, um dos fatores que explica sua forte presença em softwares embarcados [Lua.org, 2025; Siberoloji, 2021].

Outro aspecto relevante é sua curva de aprendizado relativamente suave, favorecida pela sintaxe simples e documentação acessível, o que estimula sua adoção tanto em ambientes acadêmicos quanto industriais. A comunidade ativa e a ampla disponibilidade de bibliotecas também reforçam sua utilização em diferentes domínios, do desenvolvimento de jogos digitais até ferramentas de segurança e automação de redes [Ierusalimschy, 2016; Marbux, 2021]. Seu pequeno tamanho e eficiência explicam sua adoção em motores de jogos, plataformas como Roblox e ferramentas como Nmap, consolidando-se como uma linguagem de uso global.

A flexibilidade de Lua pode ser observada na Listagem 1, que demonstra três características fundamentais da linguagem: o uso de tabelas como estrutura de dados universal, a sintaxe de açúcar para orientação a objetos e o uso de metatables e corrotinas para estender a semântica da linguagem.

```lua
-- 1. Tabelas e Sintaxe de Objeto
local pessoa = {
    nome = "Maria",
    idade = 30,
    falar = function(self)
        print("Oi, meu nome e " .. self.nome)
    end
}
-- O operador ':' passa 'pessoa' como argumento 'self' implicitamente
pessoa:falar()

-- 2. Metatables
local vetor = {x = 1, y = 2}
local mt = {
    __add = function(a, b) -- Sobrecarga do operador +
        return {x = a.x + b.x, y = a.y + b.y}
    end
}
setmetatable(vetor, mt)

-- 3. Corrotinas
function range(a, b)
    return coroutine.wrap(function()
        for i = a, b do
            coroutine.yield(i) -- Pausa e retorna valor
        end
    end)
end

for i in range(1, 5) do
    print("Valor:", i)
end
```

Listing 1. Exemplo de estruturas fundamentais da linguagem Lua.

2.2 Modelos de Linguagem de Grande Escala (LLMs)

LLMs representam um avanço disruptivo no processamento de linguagem natural, fundamentados na arquitetura Transformer e treinados em volumes massivos de dados textuais [Naveed *et al*., 2024].

No domínio da engenharia de software, a aplicação de LLMs transformou-se em uma mudança de paradigma. Wang and Chen [2023] destacam que, devido à similaridade estrutural entre linguagens de programação e linguagens naturais, os modelos treinados em vastos repositórios de código (como GitHub) desenvolveram uma capacidade sem precedentes de compreender, gerar e completar código-fonte. Diferentemente de ferramentas tradicionais baseadas em análise estática ou templates, os LLMs possuem uma adaptabilidade contextual que lhes permite lidar com nuances semânticas e gerar soluções para problemas complexos baseados em descrições em linguagem natural [Deckker and Sumanasekara, 2025].

O impacto dessas ferramentas estende-se por todo o ciclo de desenvolvimento de software. Segundo Deckker and Sumanasekara [2025], a integração de LLMs aumenta substancialmente a produtividade dos desenvolvedores, auxiliando não apenas na escrita de código (boilerplate), mas também na depuração (*debugging*), documentação e prototipagem rápida. Contudo, essa adoção traz desafios, como a possibilidade de alucinações (geração de código incorreto mas plausível) e a introdução de vulnerabilidades de segurança, exigindo supervisão humana e validação rigorosa [Wang and Chen, 2023; Deckker and Sumanasekara, 2025].

Recentemente, o desenvolvimento de LLMs especializados em código (*Code LLMs*) atingiu um novo estado da arte com modelos abertos como o StarCoder2 [Lozhkov *et al*., 2024]. Treinado sobre o *dataset* The Stack v2, este modelo diferencia-se por um treinamento focado na qualidade e transparência dos dados. Avaliações empíricas demonstram que arquiteturas especializadas, como a do StarCoder2, conseguem igualar ou superar o desempenho de modelos proprietários significativamente maiores, especialmente em linguagens com menos recursos disponíveis (*low-resource languages*), como é o caso de Lua, e em tarefas que exigem janelas de contexto estendidas para a compreensão de dependências globais em arquivos extensos [Lozhkov *et al*., 2024].

2.3 Fuzzing

O *fuzzing* é uma técnica de teste de software que consiste em submeter um programa a entradas inesperadas, inválidas ou malformadas, com o objetivo de revelar falhas, vulnerabilidades ou comportamentos incorretos [Godefroid, 2020; Manès *et al*., 2021]. Originalmente introduzida por Miller et al. na década de 1990, ao testar utilitários UNIX com entradas aleatórias, a técnica tornou-se uma das mais eficazes para a

detecção automatizada de erros e é atualmente amplamente utilizada tanto pela academia quanto pela indústria [Manès *et al*., 2021].

O processo básico de *fuzzing* envolve três elementos principais: *(i)* o programa sob teste, conhecido como *Program Under Test (PUT)*; *(ii)* um conjunto de entradas, chamadas de *seeds*, que servem como ponto de partida; e *(iii)* o *fuzzer*, a ferramenta responsável por gerar novas entradas e monitorar a execução do programa. Durante a execução, o fuzzer busca gatilhos para falhas como *crashes*, violações de memória, estouros de buffer e erros de validação de entrada.

2.3.1 Tipos de Fuzzing

Segundo Manès et al. [Manès *et al*., 2021], o *fuzzing* pode ser classificado em três categorias principais:

- **Black-box**: trata o programa como uma "caixa-preta", sem conhecimento interno de sua implementação. As entradas são geradas de maneira aleatória ou com mutações simples, e as falhas são observadas apenas pela saída ou comportamento externo. É simples e escalável, mas apresenta baixa eficácia.
- **White-box**: tem acesso completo ao código-fonte e aplica técnicas de análise simbólica e de fluxo de dados para guiar a geração de entradas. Apresenta alta taxa de cobertura, mas sofre com o problema da explosão de estados, tornando-se caro e de difícil aplicação em programas grandes.
- **Grey-box**: combina as duas abordagens anteriores, utilizando informações parciais, como métricas de cobertura de código, para guiar as mutações. Ferramentas como o AFL (*American Fuzzy Lop*) tornaram essa abordagem a mais popular, por equilibrar eficácia e custo.

2.3.2 Abordagens de geração de entradas

A qualidade e eficácia do *fuzzing* dependem fortemente de como as entradas são geradas. Existem duas estratégias principais [Godefroid, 2020; Huang *et al*., 2024]:

- **Fuzzing baseado em mutação**: inicia-se a partir de seeds válidas, que são modificadas de forma aleatória ou sistemática. Operações típicas incluem a inversão de bits, inserção de bytes especiais, duplicação ou remoção de blocos inteiros e alterações aritméticas em valores. Essa abordagem é eficiente quando há seeds de alta qualidade.
- **Fuzzing baseado em geração**: cria entradas do zero a partir de uma gramática formal ou modelos estruturais do formato de entrada aceito pelo programa. Essa técnica é indicada para sistemas que exigem entradas complexas ou altamente estruturadas, como compiladores ou interpretadores de linguagens.

Alguns trabalhos recentes combinam as duas estratégias, usando gramáticas para gerar entradas iniciais e, em seguida, aplicando mutações para explorar variações adicionais [Huang *et al*., 2024; Xia *et al*., 2024].

2.4 Fuzzing com LLMs

A incorporação de LLMs ao processo de *fuzzing* representa uma evolução das abordagens tradicionais. Diferentemente de mutações simples, como alteração de bits ou bytes, os LLMs conseguem aplicar transformações semanticamente relevantes em scripts de programação, aumentando as chances de descobrir falhas mais profundas [Huang *et al*., 2024; Deckker and Sumanasekara, 2025]. As principais estratégias incluem:

- **Geração direta de entradas**: o LLM cria scripts inteiros a partir de descrições ou instruções;
- **Mutação guiada por LLM**: o modelo recebe um script válido e o modifica preservando sua sintaxe e adicionando variações mais ricas.

Essa integração amplia a cobertura dos testes e facilita a adaptação a novas versões de linguagens, uma vez que os LLMs não dependem de gramáticas manualmente definidas. Assim, o *fuzzing* apoiado por LLMs se apresenta como um caminho promissor para aumentar a eficácia da detecção de vulnerabilidades em linguagens como Lua.

3 Trabalhos Relacionados

Diversos estudos recentes têm explorado a integração de Modelos de Linguagem de Grande Escala em técnicas de *fuzzing*, propondo soluções que ampliam a capacidade de descoberta de falhas em compiladores e interpretadores.

O Fuzz4All [Xia *et al*., 2024] apresenta um *fuzzer* universal orientado por LLMs, capaz de gerar testes diversificados para múltiplas linguagens. Sua principal contribuição é o uso de *autoprompting* e um ciclo iterativo de geração e mutação, que resultaram em melhorias significativas de cobertura e descoberta de bugs em diversos compiladores como GCC e Clang.

O trabalho Rust-Twins [Yang *et al*., 2024] aplica mutação baseada em LLMs no contexto da linguagem Rust. Para superar as restrições sintáticas da linguagem, o método utiliza mutações específicas guiadas por *prompts* e técnicas de geração de macros duplas para testes diferenciais. Os resultados mostraram maior cobertura e a identificação de vulnerabilidades inéditas no compilador *rustc*.

No contexto de JavaScript, o CovRL-Fuzz [Eom *et al*., 2024] combina mutações guiadas por LLM com aprendizado por reforço e métricas de cobertura. A técnica de *mutation by mask* demonstrou eficácia na criação de casos de teste válidos e na detecção de vulnerabilidades de segurança, superando fuzzers tradicionais.

Já o FlowFusion [Jiang *et al*., 2025] propõe a fusão de fluxos de dados para gerar novos casos de teste a partir da suíte oficial do PHP. A abordagem, aliada a mutações e recombinações, possibilitou a descoberta de centenas de bugs no interpretador da linguagem.

Por fim, o MetaMut [Ou *et al*., 2024] apresenta um framework para criação automática de operadores de mutação utilizando LLMs. Em vez de apenas gerar entradas, o sistema sintetiza mutadores reutilizáveis, que foram aplicados com sucesso em compiladores robustos como GCC e Clang, resultando em ampla cobertura e na descoberta de falhas críticas.

Embora esses trabalhos demonstrem a viabilidade e a relevância do uso de LLMs no *fuzzing*, a maioria foca em linguagens de compilação estática como C/C++ e Rust. A presente pesquisa diferencia-se ao focar na linguagem Lua, um ambiente de script dinâmico raramente explorado na literatura de fuzzing. Nossa contribuição não se limita a aplicar técnicas

existentes, mas em desenvolver uma metodologia de engenharia de prompts especializada para explorar as particularidades de Lua (como tabelas, metatables e corrotinas) e avaliar se mutações semanticamente ricas, guiadas por LLM, podem de fato ampliar a eficácia na identificação de vulnerabilidades em interpretadores de linguagens dinâmicas.

4 Metodologia

A metodologia proposta divide-se em duas fases distintas e complementares: a geração inicial de um grupo de sementes e o ciclo contínuo de *fuzzing* evolutivo. A arquitetura completa do sistema e o fluxo de dados entre os componentes são apresentados na Figura 1.

O processo, conforme ilustrado, segue a seguinte sequência lógica:

1. **Geração de Sementes:** Esta etapa visa mitigar o problema de *cold start*. O modelo *StarCoder2:instruct* é alimentado com arquivos da suíte de testes oficial da linguagem Lua, que servem como contexto para o aprendizado em poucas etapas. O gerador usa *temperature: 0.8* sendo um pouco mais conservador para garantir que os *seeds* iniciais sejam válidos e *num_predict: 400* para arquivos mais curtos e simples. O sistema seleciona aleatoriamente um dentre 10 *prompts* projetados para instigar a criação de novos casos de teste. O código gerado passa imediatamente para verificar a sintaxe pelo próprio parser de Lua. Apenas scripts sintaticamente válidos são armazenados, formando o Grupo de Sementes inicial.
2. **Fuzzer:** Esta fase constitui o núcleo da ferramenta e é gerenciada por um sistema de filas duplas (Fila Atual e Próxima Fila). O ciclo opera da seguinte forma:
    - Uma semente é retirada da *Fila Atual* e enviada ao LLM junto com um dentre 20 *prompts* de mutação. O modelo usa a *temperature: 1.0* com *num_predict: 2048*. O *Script Mutado* resultante é executado no interpretador Lua instrumentado com *AddressSanitizer* (ASan) e *UndefinedBehaviorSanitizer* (UBSan).
    - *Tratamento de Resultados:*
        - **Erro/Timeout:** Scripts inválidos são descartados.
        - **Falha (Bug):** Se uma falha é detectada, ela é registrada e a semente original recebe alta prioridade na fila seguinte para gerar mais variantes.
        - **Sucesso (Válido):** Scripts válidos avançam para a análise de cobertura.
    - *Evolução:* O script é reexecutado em um binário com *GCOV*. Se houver ganho de cobertura em relação à semente original, essa mutação é considerado "interessante" e é adicionado tanto à fila atual(não podendo criar outros scripts) quanto á próxima fila com alta prioridade. Quando a fila atual acabar, a próxima fila assume seu lugar, começando outro ciclo.

4.1 Fases da Pesquisa

A execução foi organizada nas seguintes fases:

- **Fase 1**: Revisão Bibliográfica sobre *fuzzing* de interpretadores e LLMs.
- **Fase 2**: Definição do Ambiente Experimental e Compilação do Interpretador Lua (versão 5.4.8) com *flags* de cobertura e sanitização.
- **Fase 3**: Geração do *Pool* Inicial de Sementes utilizando LLM e a suíte de testes oficial.
- **Fase 4**: Desenvolvimento e Execução do *Fuzzer*, operando em ciclo contínuo de mutação e verificação.
- **Fase 5**: Análise dos Resultados, focada no ganho de cobertura interna do interpretador e na quantidade de falhas únicas encontradas.

4.2 Ambiente Experimental

Os experimentos utilizam o código-fonte oficial do interpretador Lua (versão 5.4.8). O ambiente foi preparado com duas compilações distintas do interpretador:

1. **Binário Sanitizado**: Compilado com *Clang* utilizando *-fsanitize=address,undefined* para detecção precisa de erros de memória.
2. **Binário de Cobertura**: Compilado com *GCC* utilizando *–coverage* (GCOV) para mapear a execução das instruções C do interpretador e capturar a porcentagem e o número total de linhas.

O modelo *StarCoder2:instruct* é executado localmente via plataforma Ollama, garantindo reprodutibilidade e isolamento. Foi utilizado duas temperaturas diferentes, que representam a "criaivadade do modelo", sendo mais conservador para geração de sementes e mais "criativo"para as mutações.

A escolha do modelo (especificamente a variante ajustada para instruções) como motor de geração e mutação justifica-se por duas características técnicas fundamentais descritas por Lozhkov *et al*. [2024]:

1. **Desempenho em Lua:** O modelo demonstra capacidade superior em linguagens de *script* com menos recursos de treinamento disponíveis (*low-resource*). Em benchmarks comparativos, o StarCoder2 apresentou desempenho superior a concorrentes de maior porte (como DeepSeekCoder-33B) especificamente em linguagens como Lua, D e Julia.
2. **Profundidade de Contexto:** O suporte a janelas de contexto de 16k tokens permite que o modelo processe não apenas o trecho de código a ser mutado, mas também a suíte de testes e definições auxiliares injetadas no *prompt*. Isso é essencial para manter a consistência semântica em mutações profundas que exigem o entendimento de todo o escopo do script.

4.3 Engenharia de Prompts

O sucesso da metodologia depende da qualidade das instruções fornecidas ao LLM. Foram desenvolvidos dois conjuntos distintos de *prompts*: um focado na criação inicial de sementes válidas, e outro focado na mutação para testar diversos caminhos do interpretador.

4.3.1 Estratégias de Geração (Bootstrapping)

Na fase inicial, o objetivo é criar um conjunto diversificado. Para tal, foram utilizados 10 *prompts*, descritos detalhadamente na Tabela 1, projetados para explorar funcionalidades

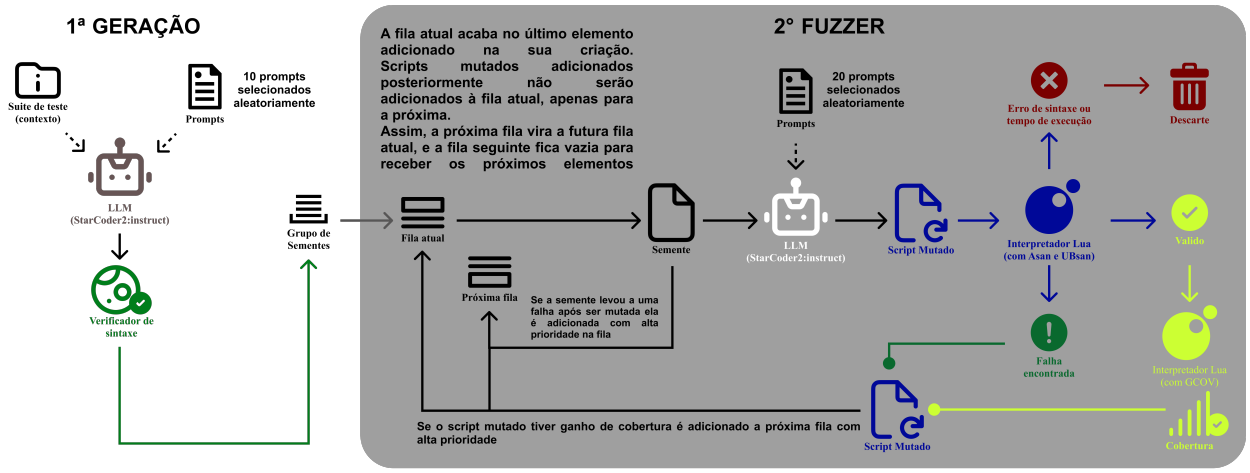

Figura 1. Fluxo de trabalho da metodologia proposta: (1) Fase de Geração de Sementes utilizando contexto da suíte de testes; (2) Ciclo de Fuzzing com orquestração de filas e análise de cobertura/sanitização.

específicas da linguagem Lua e mitigar o problema de arranque a frio (*cold start*).

4.3.2 Estratégias de Mutação (Fuzzing)

Durante o ciclo evolutivo, 20 estratégias são aplicadas para transformar sementes válidas em novos casos de teste. Estas estratégias estão sumarizadas na Tabela 2, organizadas em categorias que variam desde mutações específicas da sintaxe de Lua até refatorações semânticas e injeção de complexidade estrutural.

4.4 Validação

A validação consiste em demonstrar que a mutação guiada por LLM é capaz de: (1) gerar e mutar scripts sintaticamente válidos em alta taxa; e (2) aumentar progressivamente a cobertura do código nativo do interpretador partindo de sementes iniciais, atingindo áreas da implementação da linguagem que testes simples não alcançariam.

5 Desenvolvimento

A implementação do sistema foi realizada em Python, estruturada em dois módulos principais para lidar com o desafio de obter sementes de qualidade (o problema de "Cold Start") e para gerenciar a orquestração do *fuzzing* evolutivo.

5.1 Criação do Grupo Inicial

A eficácia de um *fuzzer* baseado em mutação depende diretamente da diversidade e validade de suas entradas iniciais. O Algoritmo 1 descreve o processo de criação e validação das sementes.

Algoritmo 1 Gerador de Sementes

Entrada: Tempo Máx. (T_{max}), Suíte Lua (Ctx)
Saída: Pool de Sementes ($Pool$)

função GENERATOR(T_{max}, Ctx)
  enquanto $Tempo < T_{max}$ **faça**
    $Prompt \leftarrow$ CRIARPROMPT(Ctx)
    $CodBruto \leftarrow$ LLM.GERAR("$starcoder2$", $Prompt$)
    $CodLimpo \leftarrow$ LIMPAR($CodBruto$)
    se SINTAXE(CodLimpo) == OK **então**
      se $CodLimpo$ **é** UNICO **então**
        $Pool$.ADD($CodLimpo$)
      fim se
    fim se
  fim enquanto
fim função

5.2 Orquestração e o Loop de Fuzzing

O módulo orquestrador (*MainFuzzer*) gerencia o ciclo de vida do teste mantendo duas filas de sementes: a fila vigente e a próxima fila

Uma distinção fundamental implementada na arquitetura do orquestrador é a separação entre erros de *script* e erros de implementação. Erros de sintaxe ou de tempo de execução (*runtime errors*) gerados pela máquina virtual Lua são considerados comportamentos esperados e seguros. O foco da ferramenta é identificar falhas na "camada C" do interpretador (o binário compilado), onde residem vulnerabilidades críticas de memória.

Para isso, a estratégia de execução adota o princípio do sanitizador primeiro onde o código é executado em um binário compilado com *AddressSanitizer* (ASan) e *UndefinedBehaviorSanitizer* (UBSan). Estes sanitizadores instrumentam o código C para interceptar acessos inválidos de memória que passariam despercebidos ou causariam apenas um *segmentation fault* genérico. Caso não haja falha crítica nesta camada nativa, o sistema executa o binário de cobertura (GCOV) para saber se pelo menos houve um ganho de cobertura, economizando recursos computacionais. A lógica de decisão é apresentada no Algoritmo 2.

Tabela 1. Estratégias de Geração de Sementes (Bootstrapping).

Nome do Prompt	Descrição
GEN_NESTED_TABLES	Generate deep nested tables with mixed keys (integers, strings, and other tables) and iterate over them using pairs and ipairs.
GEN_STRING_MANIP	Write a program that heavily uses the string library (e.g., gsub, match, format) to manipulate large text buffers and test pattern matching.
GEN_COMPLEX_LOOPS	Create a script with nested control structures, combining numeric for, generic for, while, and repeat-until loops with break and goto statements.
GEN_FUNCTIONS	Generate code demonstrating functions with variable arguments (...), multiple return values, and anonymous functions passed as arguments.
GEN_COROUTINES	Write a Lua script implementing a producer-consumer pattern or custom iterator using coroutine.create, yield, and resume.
GEN_METATABLES	Create an object-oriented example using metatables to implement inheritance, and overload operators like __add, __index, and __tostring.
GEN_CLOSURES	Generate a script that uses closures and upvalues heavily to maintain state across function calls, testing scope visibility and variable lifetime.
GEN_TABLE_LIB	Write a program that exercises the table library, performing sorting (table.sort with custom comparators), insertion, and concatenation of large arrays.
GEN_MATH_BITWISE	Generate a script performing complex arithmetic sequences and bitwise operations (&, |, ~, «) to test number coercion and integer/float boundaries.
GEN_ERROR_HANDLING	Create a script that defines functions which intentionally raise errors, wrapping calls in pcall and xpcall to test exception handling and stack trace generation.

Algoritmo 2 Fuzzer Principal

Entrada: Sementes ($Pool$)
Saída: Logs de Falhas

função FUZZER
  $FilaAtual \leftarrow$ CARREGAR($Pool$)
  $FilaProx \leftarrow []$
  enquanto $Tempo < Limite$ **faça**
    se $FilaAtual$ vazio **então**
      $FilaAtual \leftarrow FilaProx$; $FilaProx \leftarrow []$
    fim se
    $Pai \leftarrow FilaAtual$.POP
    $Mutante \leftarrow$ LLM.MUTAR(Pai)
    se SINTAXE($Mutante$) $\neq$ OK **então**
      REGISTRARERROSINTAXE()
      continue
    fim se
    $ResSan \leftarrow$ EXECSANITIZER($Mutante$)
    se $ResSan ==$ BUG **então**
      LOGBUG(Mutante, "CRASH")
      $FilaAtual$.PRIORIZAR(Pai)
    senão
      $ResCov \leftarrow$ EXECGCOV($Mutante$)
      se $ResCov > Pai.Cobertura$ **então**
        $FilaProx$.ADD($Mutante$)
      fim se
    fim se
  fim enquanto
fim função

6 Resultados e Discussão

Nesta seção, apresentamos os dados obtidos durante a execução experimental do protótipo. A avaliação foi dividida em duas etapas: a eficácia da geração inicial de sementes e o desempenho do ciclo de *fuzzing* evolutivo guiado por LLM.

Todos os experimentos foram conduzidos em um ambiente equipado com um processador Intel Xeon W-1370P (3.60GHz), 32 GB de memória RAM e uma GPU NVIDIA GeForce RTX 3060. A execução foi configurada sobre o sistema operacional Ubuntu 24.04. O modelo *StarCoder2:instruct* foi executado localmente utilizando a aceleração de hardware da GPU via plataforma Ollama.

6.1 Fase de Geração de Sementes

A etapa de preparação das sementes iniciais teve duração fixa de 72 horas. O objetivo foi mitigar o problema de (*cold start*) criando scripts sintaticamente válidos baseados na suíte de testes oficial da linguagem Lua.

Ao final do período, o gerador produziu um total de 1.915 sementes válidas e únicas. Estas sementes passaram pelos filtros de limpeza (remoção de markdown) e validação de sintaxe (parser do Lua), constituindo o *Seed Pool* inicial para a fase de orquestração.

6.2 Campanha de Fuzzing: Desempenho e Eficiência

A campanha de *fuzzing* principal foi executada por um período contínuo de 24 horas seguindo o padrão de outros trabalhos relacionados. A Tabela 3 resume as estatísticas gerais da execução.

Um dado crítico observado foi a taxa de execução (*throughput*) de aproximadamente 0,06 execuções por segundo. A análise dos tempos revela o gargalo do sistema: enquanto a execução dos scripts pelo interpretador Lua consumiu apenas cerca de 3 minutos (179s) do tempo total, a inferência do LLM na GPU RTX 3060 para gerar mutações consumiu mais de 23,9 horas (86.119s).

Isso demonstra que, embora o hardware utilizado seja robusto, o custo computacional da inferência de modelos de grande escala torna o processo de geração ordens de grandeza mais lento que a execução do teste. Isso sugere que arquiteturas híbridas (usando LLM apenas para sementes iniciais e mutadores tradicionais para volume) podem ser mais eficientes para campanhas de longa duração.

Tabela 2. Estratégias de Mutação utilizadas no ciclo evolutivo.

Categoria	Estratégia	Descrição
Específicas de Lua	MULTI_RETURN_MUTATION	Modify functions with multiple returns or variable arguments (...) by adding/removing values.
	MULTI_RET_SINGLE	Replace a single expression with a function call returning multiple values in a single-value context.
	METATABLE_MUTATION	Modify metatables/metamethods (e.g., __index) adding/removing fields to test fallback behaviors.
	COROUTINE_MUTATION	Generate functions using coroutines combined with closures and parameter passing.
Sintaxe e Estrutura	ARG_LIST_MUTATION	Modify argument lists in function calls or constructors by inserting/-removing values.
	DECL_ORDER_MUTATION	Reorder local/function declarations or interleave declarations and expressions.
	LITERAL_MUTATION	Modify literals (strings with escapes, numbers, nil) to test parsing edge cases.
	FORMATTING_MUTATION	Insert or remove comments, spaces, and irregular indentation to test parser tolerance.
	IDENTIFIER_EDGECASE	Use atypical identifiers (long names, shadowing, unicode) to stress the lexer.
	MIXED_DECLARATIONS	Combine different declaration styles (multiple assignments, mixed blocks).
Refatoração Semântica	COMPLEX_SCOPES	Mix scope styles (global/local), adding closures and shadowing to provoke binding conflicts.
	ALTERNATIVE_CONTROL	Transform loops (e.g., numeric for to while) maintaining logic but altering control structure.
	BITWISE_ARITH	Combine arithmetic and bitwise operations to exercise type coercion.
	SHADOW_BUILTIN	Overwrite standard functions (e.g., print) with local versions to test shadowing.
Complexidade e Runtime	COMPLEX_TABLE	Transform simple tables into structures with metatables, functions as values, or deep nesting.
	BLOCK_NESTING	Wrap code in do...end blocks, closures, or nested ifs to test deep parsing.
	ERROR_INJECTION	Insert deliberate errors inside pcall/xpcall blocks to test error handling recovery.
	GC_AND_METAGC	Force garbage collection cycles using tables with __gc metamethods.
	DYNAMIC_CHUNK	Create code chunks as strings and load them via load/loadfile.
	MODULES_REQ	Manipulate package.path and simulate module conflicts to test loading.

Tabela 3. Estatísticas da Campanha de Fuzzing (24 horas).

Métrica	Valor
Total de Mutações Geradas	5.115
Sementes na Fila Final	2.124
Execuções por Segundo	$\approx 0,06$
Tempo Total em Inferência (LLM)	86.119 s ($\approx 23,9$ h)
Tempo Total em Execução (Lua)	179,59 s (≈ 3 min)

6.3 Análise das Mutações e Erros

A eficácia do LLM em gerar códigos que o interpretador aceita processar é um indicador chave da qualidade do *fuzzer*. Os dados abaixo detalham o destino das 5.115 mutações geradas:

- **Mutações com Erro de Sintaxe:** 2.037 (39,8%) - Scripts rejeitados imediatamente pelo parser antes da execução.
- **Mutações Sintaticamente Válidas:** 3.078 (60,2%) - Scripts que avançaram para a etapa de execução.

Uma análise qualitativa dos erros de sintaxe revelou que grande parte dessas falhas originou-se de alucinações do modelo de linguagem. Foram frequentemente observados pequenos erros de formatação, como a escrita incorreta ou incompleta de palavras reservadas e o uso inconsistente de pontuação ou fechamento de blocos. Essas alucinações indicam que, embora o modelo capture a semântica geral, ele ainda é suscetível a lapsos de precisão sintática quando operando sem restrições gramaticais rígidas.

Dentre as mutações válidas executadas:

- **Execução com Sucesso:** 2.730 casos.
- **Runtime Errors:** 348 casos (6,8% do total). Estes representam scripts onde o LLM gerou operações logicamente inválidas para a linguagem (ex: operações com tipos incompatíveis), que foram tratadas corretamente pelo interpretador sem causar *crash*.
- **Bugs Críticos (Crashes/Sanitizer):** 0 casos.

Apesar de quase 40% de erros de sintaxe, uma taxa de aproveitamento de 60% é considerável para geração de código sem restrições gramaticais estritas, indicando que o modelo capturou satisfatoriamente a estrutura da linguagem Lua através do contexto fornecido.

6.4 Evolução da Cobertura de Código

O objetivo principal da validação foi verificar se as mutações guiadas por LLM poderiam explorar novos caminhos no código nativo (C) do interpretador de forma incremental. A Figura 2 ilustra o comportamento da cobertura global acumulada ao longo das 5.115 iterações.

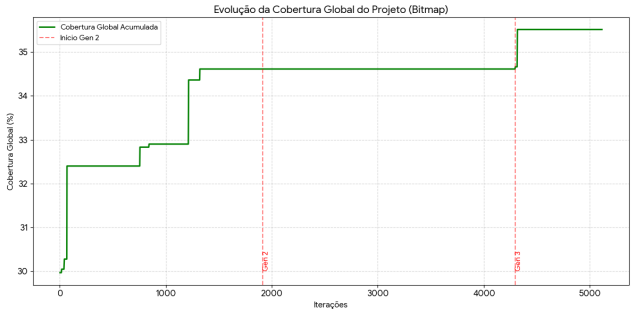

Figura 2. Evolução da cobertura global do projeto (Bitmap) ao longo do tempo. As linhas verticais tracejadas indicam o momento de troca das filas de sementes (início de novas gerações).

Os dados quantitativos finais foram:

- **Cobertura Inicial (Sementes):** 32,83%
- **Cobertura Máxima Alcançada:** 35,51%
- **Melhoria Absoluta:** +2,69%

A análise visual do gráfico revela um comportamento de crescimento em "degraus". Observam-se longos períodos de platô, onde as mutações demoram para descobrir novos caminhos, seguidos por saltos abruptos de cobertura.

Crucialmente, as linhas verticais vermelhas na Figura 2 marcam o esgotamento da fila atual e o início do processamento da próxima geração, começando pelas melhores mutações da fase anterior. Nota-se que, especialmente na transição para a *Gen 3* (após a iteração 4.000), houve um rompimento significativo de um longo período de estagnação. Isso valida a hipótese do *fuzzing* evolutivo: ao priorizar e reintroduzir scripts que tocaram novas áreas do código, o LLM conseguiu, em iterações subsequentes, aprofundar a exploração naquelas áreas específicas, atingindo lógicas condicionais que as sementes originais não alcançavam.

6.5 Discussão sobre Ausência de Bugs

Nenhum bug crítico (*crash* ou violação de memória detectada pelo ASan) foi encontrado nas 24 horas de execução. Este resultado pode ser atribuído a dois fatores principais:

1. **Maturidade do Alvo:** O interpretador Lua é um software extremamente estável, testado industrialmente há décadas. A descoberta de falhas triviais é rara.
2. **Volume de Testes Limitado:** Devido ao alto tempo de inferência do LLM, o volume de testes (5.115 execuções) foi insuficiente para saturar o espaço de estados do programa, comparado a fuzzers tradicionais que realizam milhões de execuções no mesmo período.

Contudo, o objetivo de validar a metodologia foi atingido: o sistema gerou entradas válidas, executou o ciclo de instrumentação e aumentou a cobertura de código de forma autônoma utilizando apenas mutações semânticas.

7 Conclusão

Este trabalho apresentou o desenvolvimento e a avaliação de uma metodologia de *fuzzing* guiado por Modelos de Linguagem de Grande Escala (LLMs), aplicada especificamente ao interpretador da linguagem Lua. O objetivo central foi investigar se a capacidade de compreensão semântica dos LLMs poderia ser utilizada para superar as limitações dos mutadores aleatórios tradicionais, gerando casos de teste capazes de explorar novos caminhos no código nativo do interpretador.

Os resultados experimentais demonstraram que o modelo *StarCoder2-Instruct* é um motor de mutação eficaz do ponto de vista qualitativo. A estratégia de *Cold Start*, utilizando a suíte de testes oficial como contexto, provou-se viável, gerando um *pool* inicial de 1.910 sementes válidas sem a necessidade de coleta manual de repositórios externos. Durante o ciclo de *fuzzing*, o modelo manteve uma taxa de geração de código sintaticamente válido superior a 60%, um índice significativo para uma abordagem sem restrições gramaticais rígidas.

A validação por cobertura de código confirmou a hipótese de que mutações guiadas por LLM podem exercitar caminhos profundos no software alvo. Observou-se um ganho absoluto de 2,69% na cobertura do código C do interpretador Lua em relação às sementes originais. Considerando a maturidade e a estabilidade do projeto Lua, esse incremento indica que o LLM foi capaz de construir estruturas lógicas não triviais que a suíte de testes padrão não abrangia.

Entretanto, a análise de desempenho revelou um desafio crítico para a adoção puramente baseada em LLMs: o custo computacional de inferência. Com uma taxa média de apenas 0,06 execuções por segundo (em hardware com GPU RTX 3060), o tempo de geração de entradas domina completamente o ciclo de teste, tornando-o ordens de grandeza mais lento que *fuzzers* tradicionais (como AFL++), que podem atingir milhares de execuções por segundo. A ausência de *crashes* críticos detectados pode ser atribuída a este baixo volume de testes, insuficiente para saturar o espaço de estados de um software robusto em apenas 24 horas.

Conclui-se, portanto, que os LLMs introduzem uma nova dimensão ao *fuzzing*: a alta qualidade semântica individual das mutações, em contrapartida ao baixo volume de execução. Para trabalhos futuros, sugere-se a investigação de arquiteturas híbridas, onde o LLM atua apenas periodicamente para romper barreiras de cobertura complexas, enquanto mutadores tradicionais de alto desempenho encarregam-se da exploração massiva ao redor dessas novas sementes. Além disso, a quantização de modelos ou o uso de LLMs menores (3B ou 7B parâmetros) podem oferecer um compromisso melhor entre inteligência semântica e vazão de testes.

Referências

Deckker, D. and Sumanasekara, S. (2025). The role of chatgpt in software development and code generation: A review of opportunities, challenges, and future directions. *TechRxiv Preprints*. Preprint, disponível em: `https://www.researchgate.net/publication/391807454`.

Eom, J., Jeong, S., and Kwon, T. (2024). Covrl: Fuzzing javascript engines with coverage-guided reinforce-

ment learning for llm-based mutation. *arXiv preprint arXiv:2402.12222v1*.

Godefroid, P. (2020). Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76. DOI: 10.1145/3376122.

Huang, Y., Zuo, Z., Sun, Z., Yu, L., and Zhang, T. (2024). Large language models based fuzzing techniques: A survey. *arXiv preprint arXiv:2401.08753*.

Ierusalimschy, R. (2016). *Programming in Lua*. Feisty Duck, 4th edition.

Ierusalimschy, R. (2020). *Lua 5.4 Reference Manual*. PUC-Rio. Documentação oficial da linguagem Lua.

Jiang, Y., Zhang, C., Ruan, B., Liu, J., Rigger, M., Yap, R. H. C., and Liang, Z. (2025). Fuzzing the php interpreter via dataflow fusion. *arXiv preprint arXiv:2410.21713v2*.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., *et al.* (2024). Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Lua.org (2025). About lua. Acesso em May. 2025.

Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2021). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331. DOI: 10.1109/TSE.2019.2946563.

Marbux (2021). Where lua is used. Lista arquivada de casos de uso da linguagem Lua.

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., and Mian, A. (2024). A comprehensive overview of large language models. *Elsevier Preprint*. Preprint, disponível em: `https://arxiv.org/abs/2307.06435v10`.

Ou, X., Li, C., Jiang, Y., and Xu, C. (2024). The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, pages 298–312, La Jolla, CA, USA. ACM. DOI: 10.1145/3622781.3674171.

Siberoloji (2021). Basics of lua programming for nmap nse. Introdução prática ao uso de Lua com Nmap.

Wang, J. and Chen, Y. (2023). A review on code generation with llms: Application and evaluation. In *Proceedings of the 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–290, Shanghai, China. IEEE. DOI: 10.1109/MedAI59581.2023.00044.

Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. (2024). Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. IEEE/ACM. DOI: 10.1109/ICSE48619.2024.00050.

Yang, W., Gao, C., Liu, X., Li, Y., and Xue, Y. (2024). Rust-twins: Automatic rust compiler testing through program mutation and dual macros generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. DOI: 10.1145/3691620.3695059.