Explorando o Uso de LLMs para Fuzzing de Código Lua: Metodologia e Primeiras Etapas

Richard Facin Souza[1], **Samuel da Silva Feitosa**[1]

[1]Universidade Federal da Fronteira Sul

richard.souza@estudante.uffs.edu.br, samuel.feitosa@uffs.edu.br

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, aiming to explore complex execution paths more effectively. We will validate this methodology by comparing its ability to discover vulnerabilities and increase code coverage against traditional fuzzing techniques, expecting to offer a significant contribution to automated software testing and the security of systems that use Lua.*

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, visando explorar caminhos de execução complexos de forma mais eficaz. Validaremos esta metodologia comparando sua capacidade de descobrir vulnerabilidades e aumentar a cobertura de código em relação a técnicas de fuzzing tradicionais, esperando oferecer uma contribuição significativa para o teste de software automatizado e a segurança de sistemas que utilizam Lua.*

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 vulnerabilities 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, que têm maior potencial de descobrir vulnerabilidades profundas [Huang et al. 2024, Xia et al. 2024].

Para validar essa abordagem, desenvolvemos uma idei 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 é avaliar experimentalmente se essa integração resulta em um aumento mensurável na cobertura de código e na taxa de descoberta de erros quando comparada a um *fuzzer* de mutação tradicional. Com isso, esperamos oferecer uma contribuição metodológica para a área de testes automatizados, demonstrando como a especialização de prompts pode otimizar o *fuzzing* para linguagens de script específicas.

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, Ierusalimschy 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. In-

clui 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.

2.2. Modelos de Linguagem de Grande Escala (LLMs) para Fuzzing

Modelos de Linguagem de Grande Escala (LLMs) são redes neurais profundas, geralmente baseadas na arquitetura Transformer, treinadas com volumes massivos de dados textuais e de código-fonte [Naveed et al. 2024]. Essa escala permite que desenvolvam capacidades emergentes, como a compreensão da sintaxe, semântica e estruturas lógicas de linguagens de programação [Wang and Chen 2023, Deckker and Sumanasekara 2025].

No contexto de fuzzing, o potencial dos LLMs reside em sua capacidade de atuar como ”motores de mutação”, superando as limitações dos métodos tradicionais. Suas principais características aplicadas a este trabalho são:

- **Aprendizado em Contexto (In-Context Learning):** LLMs podem gerar variações de um script de entrada com base em instruções em linguagem natural (*prompts*), sem a necessidade de retreinamento. Isso permite a criação de mutações direcionadas que exploram lógicas de programa complexas de forma controlada [Xia et al. 2024].
- **Geração de Código Sintaticamente Válido:** Treinados com milhões de exemplos de código, os LLMs têm uma alta probabilidade de gerar mutações que respeitam a gramática da linguagem alvo. Isso reduz o número de casos de teste inválidos, que seriam descartados prematuramente pelo interpretador, otimizando o tempo de fuzzing [Ou et al. 2024].
- **Exploração de Casos de Borda:** A diversidade de padrões de código presentes nos dados de treinamento permite que os LLMs gerem mutações que simulam cenários de borda ou usos incomuns de APIs, que dificilmente seriam criados por mutadores baseados em regras ou aleatoriedade [Huang et al. 2024].

Apesar de seu potencial, os LLMs apresentam desafios, como a possibilidade de gerar código plausível mas funcionalmente incorreto (”alucinação”) e a forte dependência da qualidade da engenharia de prompts [Naveed et al. 2024]. Nossa metodologia busca mitigar esses riscos através de um ciclo de validação contínua e do desenvolvimento de prompts especializados para a linguagem Lua.

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.

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.

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

O objetivo é desenvolver e avaliar um protótipo de *fuzzer* para scripts em Lua baseado em mutação orientada por LLMs. O processo será dividido em fases sequenciais, a serem realizadas entre julho a dezembro de 2025, que incluirão a revisão de literatura e a análise experimental dos resultados.

4.1. Visão Geral

A abordagem adota um ciclo iterativo de geração e execução de testes, conforme ilustrado na Figura 1. Nesse ciclo, um conjunto de *seeds* (scripts Lua válidos) é utilizado como ponto de partida para as mutações realizadas pelo LLM. Cada script mutado é então executado no interpretador Lua, com monitoramento de falhas e retroalimentação do ciclo.

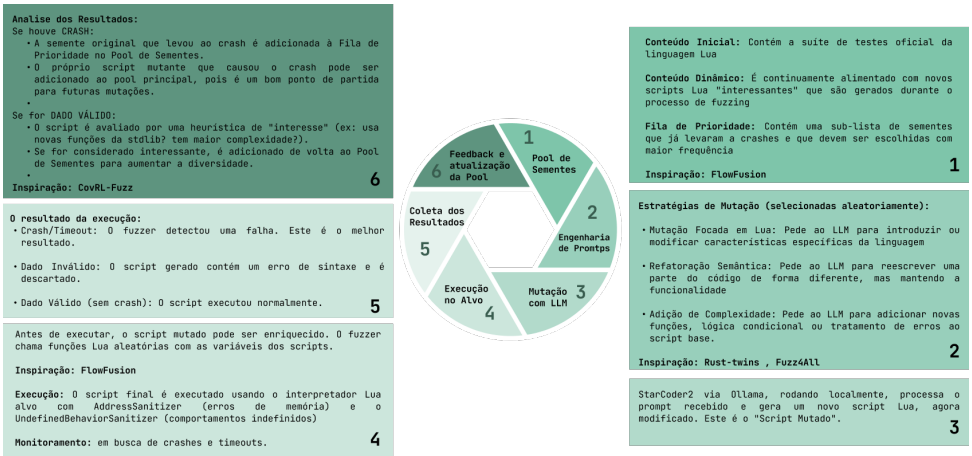

Figura 1. Ciclo de fuzzing.

O ciclo funciona em etapas sucessivas e contínuas:

1. **Seleção da semente**: um script Lua válido é escolhido de um repositório inicial de programas de referência e inspirado no FlowFusion irá conter a suite de teste oficial da linguagem. Essas sementes funcionam como ponto de partida para o processo de mutação. O Pool de sementes é continuamente alimentando por scripts mutados ”interessantes”. E uma sub-lista para sementes que levaram a falha para serem escolhidas com mais frequência.
2. **Engenharia de Prompts**: inspirado pelo Rust-twins e no Fuzz4All haverá em torno de 10 *prompts* separados por mutação focada em Lua, refatoração semântica e adição de complexidade, que será escolhido aleatoriamente para mutar o script.

3. **Mutação via LLM**: o script é enviado ao modelo *starcoder2*, executado no Ollama, junto ao *prompt* que orienta o modelo a gerar uma variante sintática ou semanticamente distinta.
4. **Execução no interpretador Lua**: motivado pelo FlowFusion antes de executar, a semente é enriquecida com funções Lua aleatórias utilizando as variáveis já existentes e assim o script mutado é executado no ambiente de testes. Nessa etapa, os monitores Asan (Address Sanitizer) e o UBSan (Undefined Behavior Sanitizer) irão reportar falhas, erros de memória e desvios de comportamento.
5. **Coleta e análise de resultados**: os resultados da execução são registrados, classificando os casos em falhas válidas, falhas descartadas (erros sintáticos) e execuções corretas.
6. **Retroalimentação do ciclo**: se houve falha, a semente original que levou à falha é adicionada à Fila de Prioridade no Pool de Sementes, e o próprio script mutante que causou a falha pode ser adicionado ao pool principal, pois é um bom ponto de partida para futuras mutações. Já se executou corretamente, o script é avaliado e, se for considerado ”interessante”(e.g., aumentou a cobertura), é adicionado de volta ao Pool de Sementes para aumentar a diversidade, inspirado no CovRL-Fuzz.

Esse fluxo cíclico garante que o processo de *fuzzing* não se limita a mutações simples e repetitivas. Ao contrário, ele explora de maneira incremental novos comportamentos do interpretador Lua, ampliando as chances de detectar vulnerabilidades ou inconsistências que não seriam encontradas por técnicas tradicionais.

4.2. Fases

A execução da pesquisa foi organizada nas seguintes fases:

- **Fase 1**: Revisão Bibliográfica - levantamento teórico sobre fuzzing, LLMs e a linguagem Lua, além da análise de trabalhos correlatos.
- **Fase 2**: Definição do Ambiente Experimental - preparação do ambiente de testes, incluindo a seleção da versão do interpretador Lua, os mecanismos de monitoramento de falhas ASan e UBSan, as métricas de cobertura utilizando o *LuaCov* e a configuração do modelo de linguagem escolhido.
- **Fase 3**: Desenvolvimento do *fuzzer* - construção do protótipo capaz de operar em um ciclo automático. O sistema envia scripts Lua ao LLM com *prompts* projetados para guiar as mutações de forma sintática e semanticamente relevante.
- **Fase 4**: Geração e Execução dos Testes - aplicação do fuzzer sobre um conjunto de programas Lua de referência. Para fins comparativos, também será executado um *fuzzer* de *baseline* com mutações não guiadas por LLM.
- **Fase 5**: Análise dos Resultados - comparação das métricas coletadas, como número de falhas encontradas, diversidade dos testes e cobertura de código atingida. O objetivo é validar se o uso do LLM amplia a eficácia da abordagem.

4.3. Ambiente Experimental

Os experimentos serão conduzidos em um ambiente controlado, utilizando a versão Lua 5.4 como interpretador alvo. O monitoramento de falhas será realizado por meio da análise de códigos de erro, sinais de interrupção (como *segmentation faults*) e o uso das ferramentas Asan (Address Sanitizer) e UBSan (Undefined Behavior Sanitizer), de sanitização. O modelo de linguagem utilizado será o *starcoder2*, executado localmente através da plataforma Ollama para garantir a reprodutibilidade dos resultados.

4.4. Métricas de Avaliação

A validação da hipótese será realizada por meio da análise quantitativa das seguintes métricas, alinhadas com a literatura de *fuzzing* [Godefroid 2020, Manès et al. 2021]:

- **Número de Falhas Únicas Descobertas:** Quantidade de *crashes* ou comportamentos anômalos distintos identificados por cada abordagem.
- **Cobertura de Código:** Medida utilizando a ferramenta **luacov**, analisando a porcentagem de linhas e ramos (*branches*) do código-fonte das *seeds* e de bibliotecas padrão que foram executados. Será observada a taxa de crescimento da cobertura ao longo do tempo.
- **Taxa de Geração de Entradas Válidas:** Percentual de scripts mutados que são sintaticamente válidos e executáveis pelo interpretador Lua. Espera-se que a abordagem com LLM tenha uma ótima taxa, otimizando o tempo de teste.

4.5. Validação

A validação da metodologia será realizada por meio da comparação sistemática entre os resultados do *fuzzer* orientado por LLM e o *fuzzer* de *baseline*, utilizando as métricas definidas anteriormente. Espera-se demonstrar que a incorporação de um modelo de linguagem amplia a diversidade e a relevância das mutações, resultando em uma maior capacidade de identificar vulnerabilidades em scripts Lua.

5. Considerações Finais e Trabalhos Futuros

Neste trabalho foi apresentada uma proposta de *fuzzer* baseado em mutação para a linguagem Lua, apoiado por Modelos de Linguagem de Grande Escala. A fundamentação teórica mostrou como a combinação entre *fuzzing* e LLMs surge como uma abordagem promissora para superar limitações de técnicas tradicionais, permitindo mutações semanticamente mais ricas e potencialmente mais eficazes na detecção de falhas.

A metodologia descrita estabeleceu um ciclo iterativo, em que códigos em Lua são selecionados, mutados por meio de um modelo de linguagem, executados no interpretador e analisados quanto à ocorrência de falhas. Esse processo contínuo visa ampliar a diversidade e a relevância das entradas de teste, explorando características próprias da linguagem Lua. O diferencial em relação a trabalhos correlatos está justamente no foco dessa linguagem pouco explorada em pesquisas de teste e na adoção de processos relevantes levantados.

Como trabalhos futuros, destacam-se as seguintes direções:

- **Exploração de técnicas de *prompt engineering*:** investigar diferentes estratégias de construção de prompts para guiar mutações específicas em elementos característicos da linguagem Lua, como tabelas, metatables e corrotinas. A intenção é avaliar como diferentes estilos de prompt podem influenciar a qualidade e a relevância semântica das mutações geradas.
- **Definição de métricas de avaliação mais abrangentes:** além da cobertura de código, propor métricas que contemplem aspectos de segurança, desempenho e manutenção do código mutado. Essas métricas podem contribuir para uma análise mais completa da eficácia do fuzzer, indo além da simples detecção de falhas superficiais.

- **Generalização para outras linguagens de script:** estender a abordagem para linguagens com características semelhantes, como JavaScript e Python, investigando os desafios de adaptação e avaliando o potencial de reutilização do método em diferentes contextos. Essa expansão pode demonstrar a versatilidade e o alcance da proposta.
- **Integração em pipelines de desenvolvimento e teste:** estudar formas de incorporar o *fuzzer* a ambientes de desenvolvimento reais, como ferramentas de integração contínua (CI/CD). Essa integração permitiria que mutações e testes fossem executados automaticamente durante o ciclo de desenvolvimento, ampliando o impacto prático da abordagem.

Dessa forma, espera-se que este trabalho contribua tanto para a área de *fuzzing* quanto para a aplicação prática de LLMs em testes automatizados. O avanço nessa linha de pesquisa pode levar ao desenvolvimento de ferramentas mais eficazes, capazes de fortalecer a confiabilidade e a segurança de softwares em diferentes domínios.

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 reinforcement 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.

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*.

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.

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.

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.

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.

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.