LLM-Guided Semantic Mutation for Lua Interpreter Fuzzing: A Coverage-Driven Approach

Richard Facin Souza
Universidade Federal da Fronteira Sul
Chapecó, Brazil
richardfacinsouza@gmail.com

Andrei De Almeida Sampaio Braga
Universidade Federal da Fronteira Sul
Chapecó, Brazil
andrei.braga@uffs.edu.br

Samuel Da Silva Feitosa
Universidade Federal da Fronteira Sul
Chapecó, Brazil
samuel.feitosa@uffs.edu.br

Giancarlo Dondoni Salton
Universidade Federal da Fronteira Sul
Chapecó, Brazil
gian@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. 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.

Keywords

Fuzzing, LLM, Lua, Software Testing, Security

1 Introduction

The Lua language is known for being lightweight, efficient, and easy to integrate, being widely used in areas such as game development, embedded systems, and task automation [5, 9, 14]. Due to its popularity and applicability, ensuring the reliability and security of applications developed in Lua is a growing concern, particularly when considering the potential impact of failures in critical environments.

One of the most effective methods for identifying vulnerabilities and flaws in software is through the technique of *fuzzing*, an approach distinguished by its ability to generate unexpected or malformed inputs that exercise different execution paths in programs [3, 10]. However, traditional *fuzzing*, especially mutation-based fuzzing, faces significant limitations, such as the difficulty of generating semantically relevant mutations or exploring non-trivial paths in the code [2, 4].

Recently, Large Language Models (LLMs) have established themselves as promising tools for code generation, including the creation of test cases [1, 15]. In the context of *fuzzing*, these models exhibit substantial potential to improve input generation, enabling smarter and more targeted mutations that consider both the syntax and semantics of the language [4, 16]. This advancement creates opportunities to overcome long-standing limitations of conventional techniques and to explore new forms of test automation.

In this work, we propose and evaluate a methodology for generating semantically rich mutations in *fuzzing*, employing LLMs as a code transformation engine. Unlike traditional approaches that rely on byte-level alterations, our technique leverages the ability of LLMs to comprehend source code syntax and semantics to create complex and valid variations [4, 16].

To validate this approach, we developed a *fuzzing loop* prototype that applies LLM-guided mutation specifically to the Lua language. The choice of Lua as a case study is justified by its dynamic characteristics and its prevalent use in critical systems where security is paramount [5, 11]. The main objective is to validate the tool by analyzing three key aspects: the incremental gain in code coverage provided by the mutations compared to the original seeds, the effective bug-finding capability, and the temporal performance (execution time) of the approach.

2 Theoretical Background

This section presents the conceptual foundation necessary for understanding the proposed work. It covers the Lua programming language, Large Language Models (LLMs), and the *fuzzing* technique, focusing on the combined use of LLMs to support mutation in automated testing.

2.1 Lua Language

Lua is an interpreted, efficient, and lightweight programming language developed in Brazil at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). Since its creation in 1993, it has stood out for its syntactic simplicity, portability, and extensibility [5, 6, 9]. Designed as a *glue language*, it integrates easily with C/C++ programs and is widely used in games, embedded systems, and automation applications.

The language is dynamically typed and features a single native data structure, the *table*, which can represent arrays, records, or objects with great flexibility. It also includes concepts such as first-class functions, automatic garbage collection, and support for coroutines—features that facilitate cooperative concurrency and event-driven programming [5]. Moreover, its minimalist and modular design contributes to low memory and processing consumption, making it suitable for resource-constrained devices and real-time systems, a factor that explains its strong presence in embedded software [9, 14].

Another relevant aspect is its relatively smooth learning curve, aided by a simple syntax and accessible documentation, which encourages its adoption in both academic and industrial environments. The active community reinforce its use across different domains, from digital game development to security tools and network automation [5, 11]. Its small footprint and efficiency explain its adoption in game engines, platforms such as Roblox, and tools like Nmap, establishing Lua as a globally used language.

The flexibility of Lua can be observed in Listing 1, which demonstrates three fundamental characteristics of the language: the use of tables as a universal data structure, syntactic sugar for object orientation, and the use of metatables and coroutines to extend the language's semantics.

```lua
-- 1. Tables and Object Syntax
local person = {
name = "Maria",
age = 30,
speak = function(self)
print("Hi, my name is " .. self.name)
end
}
-- The ':' operator implicitly passes 'person' as
    the 'self' argument
person:speak()

-- 2. Metatables
local vector = {x = 1, y = 2}
local mt = {
__add = function(a, b) -- Operator overloading for
     +
return {x = a.x + b.x, y = a.y + b.y}
end
}
setmetatable(vector, mt)

-- 3. Coroutines
function range(a, b)
return coroutine.wrap(function()
for i = a, b do
coroutine.yield(i) -- Pauses and returns value
end
end)
end

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

Listing 1: Example of fundamental structures of the Lua language.

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) represent a disruptive advancement in natural language processing, grounded in the Transformer architecture and trained on massive volumes of textual data [12]. The evolution of these models follows scaling laws, where increasing the number of parameters and the volume of training data results in the emergence of capabilities such as logical reasoning, planning, and in-context learning—abilities that were not explicitly trained [12].

In the domain of software engineering, the application of LLMs has become a paradigm shift. Wang and Chen [15] highlight that, due to the structural similarity between programming languages and natural languages, LLMs trained on vast code repositories (such as GitHub) have developed an unprecedented capacity to understand, generate, and complete source code. Unlike traditional tools based on static analysis or templates, LLMs possess contextual adaptability that allows them to handle semantic nuances and generate solutions to complex problems based on natural language descriptions [1].

The impact of these tools extends throughout the software development lifecycle. According to Deckker and Sumanasekara [1], the integration of LLMs substantially increases developer productivity, assisting not only in writing boilerplate code but also in debugging, documentation, and rapid prototyping. However, this adoption brings challenges, such as the possibility of hallucinations (generation of incorrect yet plausible code) and the introduction of security vulnerabilities, necessitating human oversight and rigorous validation [1, 15].

Recently, the development of code-specialized LLMs (Code LLMs) has reached a new state of the art with open models such as StarCoder2 [8]. Trained on The Stack v2 dataset, this model distinguishes itself through a focus on data quality and transparency. Empirical evaluations demonstrate that specialized architectures, such as that of StarCoder2, can match or exceed the performance of significantly larger proprietary models, especially in low-resource languages—such as Lua—and in tasks requiring extended context windows for understanding global dependencies in large files [8].

2.3 Fuzzing

Fuzzing is a software testing technique that consists of subjecting a program to unexpected, invalid, or malformed inputs with the aim of revealing faults, vulnerabilities, or incorrect behaviors [3, 10]. Originally introduced by Miller et al. in the 1990s while testing UNIX utilities with random inputs, the technique has become one of the most effective for automated error detection and is now widely used both in academia and industry [10].

The basic fuzzing process involves three main elements: *(i)* the program under test, known as the *Program Under Test (PUT)*; *(ii)* a set of inputs, called *seeds*, which serve as a starting point; and *(iii)* the *fuzzer*, the tool responsible for generating new inputs and monitoring program execution. During execution, the fuzzer seeks to trigger failures such as crashes, memory violations, buffer overflows, and input validation errors.

2.3.1 Types of Fuzzing. According to Manès et al. [10], fuzzing can be classified into three main categories:

- **Black-box**: treats the program as a "black box," with no internal knowledge of its implementation. Inputs are generated randomly or through simple mutations, and failures are observed only through output or external behavior. This approach is simple and scalable but has low effectiveness.
- **White-box**: has full access to the source code and applies symbolic analysis and data flow techniques to guide input

generation. It achieves high coverage rates but suffers from state explosion problems, becoming expensive and difficult to apply to large programs.
- **Grey-box**: combines the two previous approaches, using partial information, such as code coverage metrics, to guide mutations. Tools like the *American Fuzzy Lop (AFL)* have made this approach the most popular, as it balances effectiveness and cost.

2.3.2 Input Generation Approaches. The quality and effectiveness of fuzzing depend heavily on how inputs are generated. There are two main strategies [3, 4]:

- **Mutation-based fuzzing**: starts from valid seeds, which are modified randomly or systematically. Typical operations include bit flipping, insertion of special bytes, duplication or removal of entire blocks, and arithmetic alterations of values. This approach is efficient when high-quality seeds are available.
- **Generation-based fuzzing**: creates inputs from scratch using a formal grammar or structural models of the input format accepted by the program. This technique is suitable for systems that require complex or highly structured inputs, such as compilers or language interpreters.

Some recent works combine both strategies, using grammars to generate initial inputs and then applying mutations to explore additional variations [4, 16].

2.4 Fuzzing with LLMs

The incorporation of LLMs into the fuzzing process represents an evolution over traditional approaches. Unlike simple mutations such as bit or byte alterations, LLMs can apply semantically relevant transformations to programming scripts, increasing the chances of discovering deeper faults [1, 4]. The main strategies include:

- **Direct input generation**: the LLM creates entire scripts from descriptions or instructions;
- **LLM-guided mutation**: the model receives a valid script and modifies it while preserving its syntax and adding richer variations.

A fundamental challenge in mutation-based fuzzing is the *Cold Start Problem*,the requirement for a robust, diverse, and semantically valid initial set of seed inputs. Without high-quality seeds, traditional fuzzers waste vast computational cycles generating malformed inputs that are immediately rejected by the target parser. By leveraging the Transformer architecture's capacity for in-context learning, LLMs can ingest existing language specifications or small test suites to synthesize entirely new, grammatically correct seed pools from scratch, effectively solving the cold start bottleneck before the evolutionary mutation loop begins.

This integration expands test coverage and facilitates adaptation to new language versions, since LLMs do not depend on manually defined grammars. Thus, LLM-assisted fuzzing presents itself as a promising path to increase the effectiveness of vulnerability detection in languages such as Lua.

3 Related Work

Several recent studies have explored the integration of Large Language Models into fuzzing techniques, proposing solutions that enhance the fault detection capabilities of compilers and interpreters.

Fuzz4All [16] presents a universal LLM-driven fuzzer capable of generating diverse tests for multiple languages. Its main contribution is the use of autoprompting and an iterative generation and mutation loop, which resulted in significant improvements in coverage and bug discovery in several compilers, such as GCC and Clang.

The work Rust-Twins [17] applies LLM-based mutation in the context of the Rust language. To overcome the syntactic constraints of the language, the method employs prompt-guided specific mutations and double-macro generation techniques for differential testing. The results showed increased coverage and the identification of previously unknown vulnerabilities in the *rustc* compiler.

In the context of JavaScript, CovRL-Fuzz [2] combines LLM-guided mutations with reinforcement learning and coverage metrics. The mutation-by-mask technique demonstrated effectiveness in creating valid test cases and detecting security vulnerabilities, outperforming traditional fuzzers.

FlowFusion [7] proposes the fusion of data flows to generate new test cases from the official PHP test suite. This approach, combined with mutations and recombinations, enabled the discovery of hundreds of bugs in the PHP language interpreter.

Finally, MetaMut [13] presents a framework for the automatic creation of mutation operators using LLMs. Instead of merely generating inputs, the system synthesizes reusable mutators, which were successfully applied to robust compilers such as GCC and Clang, resulting in broad coverage and the discovery of critical faults.

Although these works demonstrate the feasibility and relevance of using LLMs in fuzzing, most focus on statically compiled languages such as C/C++ and Rust. The present research differentiates itself by focusing on the Lua language, a dynamic scripting environment rarely explored in the fuzzing literature. Our contribution is not limited to applying existing techniques but rather to developing a specialized prompt engineering methodology to exploit Lua's particularities (such as tables, metatables, and coroutines) and to assess whether semantically rich, LLM-guided mutations can indeed enhance the effectiveness of vulnerability identification in dynamic language interpreters.

4 Methodology

The proposed methodology is divided into two distinct and complementary phases: the initial generation of a seed pool and the continuous evolutionary fuzzing cycle. The complete system architecture and data flow among components are presented in Figure 1.

The process, as illustrated, follows the logical sequence outlined below:

(1) **1st Seed Generation:** This stage aims to mitigate the cold start problem. The **StarCoder2:instruct** (15B parameter variant via Ollama) model is fed with files from the official Lua 5.4.8 test suite (comprising 33 files and approximately 14,500 lines of code), which serve as context for few-shot learning. The files were dynamically chunked to fit within

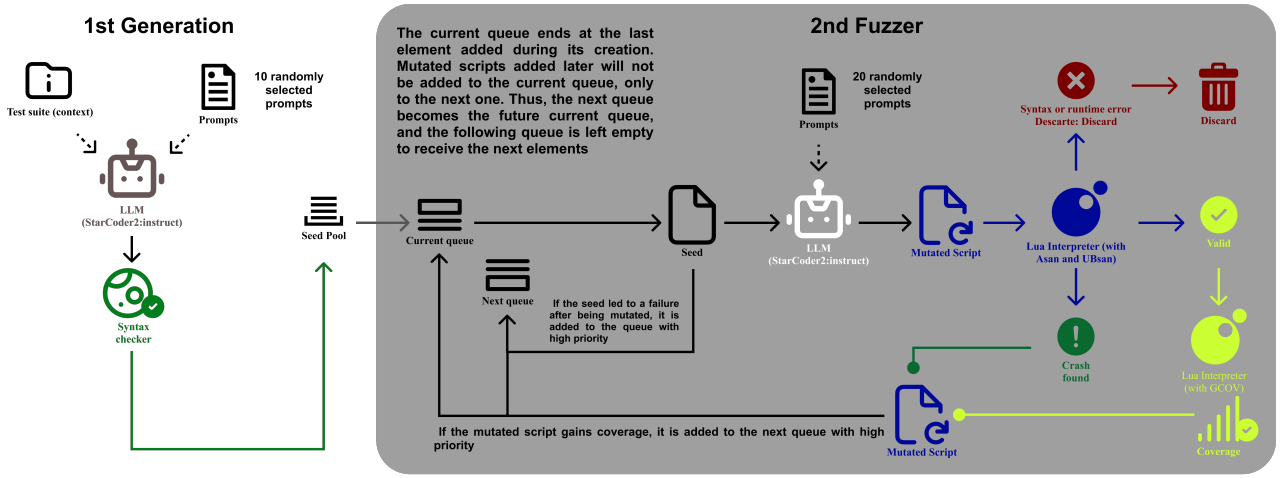

Figure 1: Workflow of the proposed methodology: (1) Seed Generation Phase using the test suite context; (2) Fuzzing Cycle with queue orchestration and coverage/sanitization analysis.

the model's context window. The generator uses a temperature of 0.8, Top-P of 0.95, and a repetition penalty of 1.1, being somewhat conservative to ensure valid initial seeds, with a prediction length of 400 tokens for shorter files. The system randomly selects one of ten prompts designed to encourage new test cases. The generated code is immediately checked by the Lua parser. Only syntactically valid scripts are stored, forming the initial Seed Pool.

(2) **2nd Fuzzer:** This phase constitutes the core of the tool and is managed by a dual-queue system (Current Queue and Next Queue). The cycle operates as follows:
- A seed is taken from the *Current Queue* and sent to the LLM along with one of twenty mutation prompts. The model uses a temperature of 1.0 with a prediction length of 2048 tokens. The resulting *Mutated Script* is executed in the Lua interpreter instrumented with *AddressSanitizer (ASan)* and *UndefinedBehaviorSanitizer (UBSan)*.
- *Result Handling:*
  - **Error/Timeout:** Invalid scripts are discarded.
  - **Failure (Bug):** If a failure is detected, it is logged, and the original seed receives high priority in the subsequent queue to generate more variants.
  - **Success (Valid):** Valid scripts proceed to coverage analysis.
- *Evolution:* The script is re-executed on a binary instrumented with GCOV. If there is a coverage gain relative to the original seed, this mutation is deemed "interesting" and is added both to the current queue (preventing it from generating other scripts) and to the next queue with high priority. When the current queue is exhausted, the next queue takes its place, starting another cycle.

4.1 Research Phases

The execution was organized into the following phases:

- **Phase 1**: Bibliographic review on interpreter fuzzing and LLMs.
- **Phase 2**: Definition of the experimental environment and compilation of the Lua interpreter (version 5.4.8) with coverage and sanitization flags.
- **Phase 3**: Generation of the initial Seed Pool using the LLM and the official test suite.
- **Phase 4**: Development and execution of the Fuzzer, operating in a continuous mutation and verification cycle.
- **Phase 5**: Analysis of results, focusing on the internal coverage gain of the interpreter and the number of unique faults found.

4.2 Experimental Environment

The experiments utilize the official source code of the Lua interpreter (version 5.4.8). The environment was prepared with two distinct builds of the interpreter:

(1) **Sanitized Binary**: Compiled with Clang using *-fsanitize=address,undefined* for precise detection of memory errors.

(2) **Coverage Binary**: Compiled with GCC using *–coverage (GCOV)* to map the execution of the interpreter's C instructions and capture both the percentage and total number of lines covered.

The *StarCoder2:instruct* model is executed locally via the Ollama platform, ensuring reproducibility and isolation.

The choice of this model (specifically the instruction-tuned variant) as the generation and mutation engine is justified by two fundamental technical characteristics described by Lozhkov et al. [8]:

(1) **Performance in Lua:** The model demonstrates superior capability in scripting languages with limited training data

available (low-resource languages). In comparative benchmarks, StarCoder2 outperformed larger competitors (such as DeepSeekCoder-33B) specifically in languages like Lua, D, and Julia.

(2) **Context Depth:** Support for 16k-token context windows allows the model to process not only the code snippet to be mutated but also the test suite and auxiliary definitions injected into the prompt. This is essential for maintaining semantic consistency in deep mutations that require understanding the entire scope of the script.

4.3 Prompt Engineering

The success of the methodology depends on the quality of the instructions provided to the LLM. Two distinct sets of prompts were developed: one focused on the initial creation of valid seeds, and another focused on mutation to exercise various interpreter paths.

4.3.1 Generation Strategies (Bootstrapping). In the initial phase, the goal is to create a diverse set. Ten prompts were used, as numbered below, designed to explore specific features of the Lua language:

(1) **GEN_NESTED_TABLES**: Generate deep nested tables with mixed keys (integers, strings, and other tables) and iterate over them using pairs and ipairs.
(2) **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.
(3) **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.
(4) **GEN_FUNCTIONS**: Generate code demonstrating functions with variable arguments (...), multiple return values, and anonymous functions passed as arguments.
(5) **GEN_COROUTINES**: Write a Lua script implementing a producer-consumer pattern or custom iterator using coroutine.create, yield, and resume.
(6) **GEN_METATABLES**: Create an object-oriented example using metatables to implement inheritance, and overload operators like __add, __index, and __tostring.
(7) **GEN_CLOSURES**: Generate a script that uses closures and upvalues heavily to maintain state across function calls, testing scope visibility and variable lifetime.
(8) **GEN_TABLE_LIB**: Write a program that exercises the table library, performing sorting (table.sort with custom comparators), insertion, and concatenation of large arrays.
(9) **GEN_MATH_BITWISE**: Generate a script performing complex arithmetic sequences and bitwise operations (&, |, ~, <<) to test number coercion and integer/float boundaries.
(10) **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.

4.3.2 Mutation Strategies (Fuzzing). During the evolutionary cycle, twenty strategies are applied to transform valid seeds into test cases. They are divided into five categories:

Lua-Specific.

(1) **MULTI_RETURN_MUTATION**: Modify functions with multiple returns or variable arguments (...) by adding/removing values or changing order.
(2) **MULTI_RET_IN_SINGLE_EXPR**: Replace a single expression with a function call returning multiple values in a context where only one value is expected.
(3) **METATABLE_MUTATION**: Modify metatables and metamethods (e.g., __index, __newindex) by adding or removing fields to test fallback behaviors.
(4) **COROUTINE_MUTATION**: Generate functions using coroutines combined with closures and parameter passing to stress the runtime.

Syntax and Structure.

(1) **ARG_LIST_MUTATION**: Modify argument lists in function calls or table constructors by inserting extra values or removing some.
(2) **DECL_ORDER_MUTATION**: Reorder local or function declarations, or interleave declarations and expressions to test initialization dependencies.
(3) **LITERAL_MUTATION**: Modify literals (strings with escapes, numbers, nil) to test parsing edge cases.
(4) **FORMATTING_MUTATION**: Insert or remove comments, spaces, and irregular indentation to test parser tolerance.
(5) **IDENTIFIER_EDGECASE**: Use atypical identifiers (long names, shadowing, unicode) to stress the lexer.
(6) **MIXED_DECLARATIONS**: Combine different declaration styles (multiple assignments, mixed blocks) to test parsing robustness.

Semantic Refactoring.

(1) **COMPLEX_SCOPES**: Mix scope styles (global/local), adding closures and variable shadowing to provoke binding conflicts.
(2) **ALTERNATIVE_CONTROL**: Transform loops (e.g., numeric for to while) maintaining logic but altering control structure.
(3) **BITWISE_AND_ARITH_OPERATOR**: Combine arithmetic and bitwise operations to exercise type coercion and conversions.
(4) **SHADOW_BUILTIN**: Overwrite standard functions (e.g., print) with local versions and invoke both to test shadowing.

Complexity and Runtime.

(1) **COMPLEX_TABLE**: Transform simple tables into structures with metatables, functions as values, or deep nesting.
(2) **BLOCK_NESTING**: Wrap code in do...end blocks, closures, or nested ifs to test deep parsing.
(3) **ERROR_INJECTION**: Insert deliberate errors inside pcall/xpcall blocks to test error handling recovery.
(4) **GC_AND_METAGC**: Force garbage collection cycles using tables with __gc metamethods or userdata simulation.
(5) **DYNAMIC_CHUNK_LOAD**: Create code chunks as strings and load them via load/loadfile to exercise dynamic parsing.

(6) **MODULES_REQUIRE_MUTATION**: Manipulate package.path and simulate module conflicts to test the loading system.

4.4 Evaluation Metrics

The validation of the proposed approach employs metrics focused on the robustness of the interpreter:

- **Native Interpreter Code Coverage:** This metric assesses the percentage of lines and branches of the interpreter's C code that were executed. The objective is to verify whether LLM-generated mutations can exercise deep parts of the Lua virtual machine.
- **Bug Detection:** Number of unique faults (crashes, memory leaks, or undefined behaviors) reported by the sanitizers.
- **Execution Time:** Temporal efficiency of the generation-execution-analysis cycle.

4.5 Validation

The validation consists of demonstrating that LLM-guided mutation is capable of: (1) generating and mutating syntactically valid scripts at a high rate; and (2) progressively increasing the coverage of the interpreter's native code starting from initial seeds, reaching areas of the language implementation that simple tests would not achieve.

5 Development

The system implementation was carried out in Python, structured into two main modules to address the challenge of obtaining quality seeds (the "cold start" problem) and to manage the orchestration of evolutionary fuzzing.

5.1 Creation of the Initial Seed Pool

The effectiveness of a mutation-based fuzzer depends directly on the diversity and validity of its initial inputs. Algorithm 1 describes the process of creating and validating seeds.

Algorithm 1 Seed Generator

Require: Max Time (T_{max}), Lua Suite (Ctx)
Ensure: Seed Pool ($Pool$)
function GENERATOR(T_{max}, Ctx)
  while $Time < T_{max}$ **do**
    $Prompt \leftarrow$ CREATEPROMPT(Ctx)
    $RawCode \leftarrow$ LLM.GENERATE("$starcoder2$", $Prompt$)
    $CleanCode \leftarrow$ CLEAN($RawCode$)
    if SYNTAX(CleanCode) == OK **then**
      if $CleanCode$ **is** UNIQUE **then**
        $Pool$.ADD($CleanCode$)
      end if
    end if
  end while
end function

5.2 Orchestration and the Fuzzing Loop

The orchestrator module (*MainFuzzer*) manages the test lifecycle by maintaining two seed queues: the current queue and the next queue.

A fundamental distinction implemented in the orchestrator's architecture is the separation between script errors and implementation errors. Syntax errors or runtime errors generated by the Lua virtual machine are considered expected and safe behaviors. The tool's focus is to identify faults in the interpreter's "C layer" (the compiled binary), where critical memory vulnerabilities reside.

To this end, the execution strategy adopts a sanitizer-first principle, where the code is executed on a binary compiled with *AddressSanitizer (ASan)* and *UndefinedBehaviorSanitizer (UBSan)*. These sanitizers instrument the C code to intercept invalid memory accesses that would otherwise go unnoticed or cause only a generic segmentation fault. If no critical fault occurs in this native layer, the system executes the coverage binary (GCOV) to determine whether at least a coverage gain was achieved, thereby saving computational resources. The decision logic is presented in Algorithm 2.

Algorithm 2 Main Fuzzer

Require: Seeds ($Pool$)
Ensure: Failure Logs
function FUZZER
  $CurrentQueue \leftarrow$ LOAD($Pool$)
  $NextQueue \leftarrow$ []
  while $Time < Limit$ **do**
    if $CurrentQueue$ empty **then**
      $CurrentQueue \leftarrow NextQueue$; $NextQueue \leftarrow$ []
    end if
    $Parent \leftarrow CurrentQueue$.POP
    $Mutant \leftarrow$ LLM.MUTATE($Parent$)
    if SYNTAX($Mutant$) $\neq$ OK **then**
      LOGSYNTAXERROR()
      continue
    end if
    $ResSan \leftarrow$ EXECSANITIZER($Mutant$)
    if $ResSan$ == BUG **then**
      LOGBUG(Mutant, "CRASH")
      $CurrentQueue$.PRIORITIZE($Parent$)
    else
      $ResCov \leftarrow$ EXECGCOV($Mutant$)
      if $ResCov > Parent.Coverage$ **then**
        $NextQueue$.ADD($Mutant$)
      end if
    end if
  end while
end function

6 Results and Discussion

In this section, we present the data obtained during the experimental execution of the prototype. The evaluation was divided into two stages: the effectiveness of the initial seed generation and the performance of the LLM-guided evolutionary fuzzing cycle.

All experiments were conducted in an environment equipped with an Intel Xeon W-1370P processor (3.60GHz), 32 GB of RAM, and an NVIDIA GeForce RTX 3060 GPU. The execution was configured on the Ubuntu 24.04 operating system. The *StarCoder2:instruct* model was run locally using GPU hardware acceleration via the Ollama platform.

6.1 Seed Generation Phase

The initial seed preparation stage had a fixed duration of 72 hours. The objective was to mitigate the cold start problem by creating syntactically valid scripts based on the official Lua language test suite.

At the end of the period, the generator produced a total of 1,915 valid and unique seeds. These seeds passed through cleaning filters (markdown removal) and syntax validation (Lua parser), constituting the initial Seed Pool for the orchestration phase.

6.2 Fuzzing Campaign: Performance and Efficiency

The main fuzzing campaign was executed for a continuous period of 24 hours. Table 1 summarizes the general execution statistics.

Table 1: Fuzzing Campaign Statistics (24 hours).

Metric	Value
Total Mutations Generated	5,115
Seeds in Final Queue	2,124
Executions per Second	≈ 0.06
Total Inference Time (LLM)	86,119 s ($\approx$ 23.9 h)
Total Execution Time (Lua)	179.59 s ($\approx$ 3 min)

A critical observation was the execution throughput of approximately 0.06 executions per second. Analysis of the times reveals the system bottleneck: while script execution by the Lua interpreter consumed only about 3 minutes (179 s) of total time, LLM inference on the RTX 3060 GPU to generate mutations consumed over 23.9 hours (86,119 s).

This demonstrates that, although the hardware used is robust, the computational cost of large-scale model inference makes the generation process orders of magnitude slower than test execution. This suggests that hybrid architectures (using LLMs only for initial seeds and traditional mutators for volume) may be more efficient for long-duration campaigns.

6.3 Analysis of Mutations and Errors

The effectiveness of the LLM in generating code that the interpreter can process is a key indicator of the fuzzer's quality. The data below detail the fate of the 5,115 generated mutations:

- **Mutations with Syntax Errors:** 2,037 (39.8%) — Scripts rejected immediately by the parser before execution.
- **Syntactically Valid Mutations:** 3,078 (60.2%) — Scripts that advanced to the execution stage.

A qualitative analysis of the syntax errors revealed that a large portion of these failures originated from model hallucinations. Small formatting errors were frequently observed, such as incorrect or incomplete spelling of reserved words and inconsistent use of punctuation or block closures. These hallucinations indicate that, although the model captures the general semantics, it remains susceptible to lapses in syntactic precision when operating without rigid grammatical constraints.

Among the executed valid mutations:

- **Successful Execution:** 2,730 cases.
- **Runtime Errors:** 348 cases (6.8% of total). These represent scripts where the LLM generated logically invalid operations for the language (e.g., operations with incompatible types), which were correctly handled by the interpreter without causing a crash.
- **Critical Bugs (Crashes/Sanitizer):** 0 cases.

Despite nearly 40% syntax errors, a success rate of 60% is considerable for code generation without strict grammatical constraints. To eliminate these syntax wasted cycles in future iterations, the architecture could integrate *Grammar-Constrained Decoding* (GCD). By modifying the LLM's decoding loop to strictly enforce Lua's context-free grammar during token sampling, syntax hallucinations could be reduced to near zero. Furthermore, while our study prioritized open-source models for reproducibility, substituting the local model with larger proprietary frontier models (e.g., GPT-4 or Claude 3.5) would likely improve semantic reasoning and lower syntax error rates, though at a significantly higher latency and financial inference cost.

6.4 Code Coverage Evolution

The main validation objective was to verify whether LLM-guided mutations could explore new paths in the interpreter's native (C) code incrementally. Figure 2 illustrates the behavior of accumulated global coverage over the 5,115 iterations.

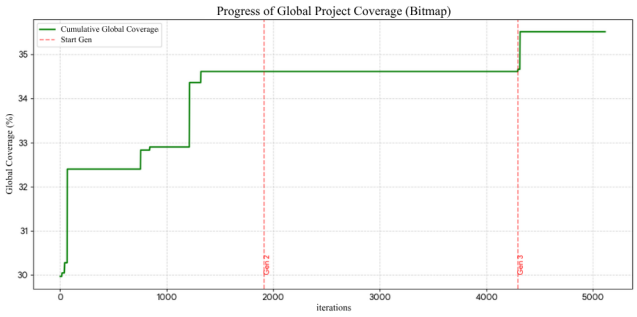

Figure 2: Evolution of global project coverage (bitmap) over time. The dashed vertical lines indicate seed queue swap moments (start of new generations).

The final quantitative data were:

- **Initial Coverage (Seeds):** 32.83%
- **Maximum Coverage Achieved:** 35.51%
- **Absolute Improvement:** +2.68%

It is important to clarify a visual artifact in Figure 2: while the text reports an initial seed coverage of 32.83%, the plotted curve in the chart initiates near the 30% baseline. This difference occurs because the chart plots the incremental discovery of coverage paths dynamically registered during the *execution* of the queues, whereas the 32.83% static baseline represents the cumulative coverage precalculated by executing the entire initial seed pool as a single batch prior to the start of the timed mutation loop.

Visual analysis of the graph reveals a stepwise growth behavior. Long plateau periods are observed, where mutations struggle to discover new paths, followed by abrupt coverage jumps.

Crucially, the red vertical lines in Figure 2 mark the exhaustion of the current queue and the start of processing the next generation, beginning with the best mutations from the previous phase. Notably, especially in the transition to Generation 3 (after iteration 4,000), a significant breakthrough from a long stagnation period occurred. This validates the evolutionary fuzzing hypothesis: by prioritizing and reintroducing scripts that touched new areas of the code, the LLM was able, in subsequent iterations, to deepen exploration in those specific areas, reaching conditional logic that the original seeds did not achieve.

6.5 Discussion on the Absence of Bugs

No critical bugs (crash or memory violation detected by ASan) were found during the 24 hours of execution. This result can be attributed to two main factors:

(1) **Target Maturity:** The Lua interpreter is an extremely stable software, industrially tested for decades. The discovery of trivial faults is rare.
(2) **Limited Test Volume:** Due to the high LLM inference time, the test volume (5,115 executions) was insufficient to saturate the program's state space, compared to traditional fuzzers that perform millions of executions in the same period.

Nevertheless, the objective of validating the methodology was achieved: the system generated valid inputs, executed the instrumentation cycle, and increased code coverage autonomously using only semantic mutations.

7 Conclusion

This work presented the development and evaluation of a fuzzing methodology guided by Large Language Models (LLMs), applied specifically to the Lua language interpreter. The central objective was to investigate whether the semantic comprehension capabilities of LLMs could be used to overcome the limitations of traditional random mutators, generating test cases capable of exploring new paths in the interpreter's native code.

The experimental results demonstrated that the StarCoder2:instruct model is an effective mutation engine from a qualitative perspective. The Cold Start strategy, using the official test suite as context, proved viable, generating an initial pool of 1,915 valid seeds without the need for manual collection from external repositories. During the fuzzing cycle, the model maintained a syntactically valid code generation rate exceeding 60%, a significant rate for an approach without strict grammatical constraints.

The code coverage validation confirmed the hypothesis that LLM-guided mutations can exercise deep paths in the target software. An absolute gain of 2.68% was observed in the coverage of the Lua interpreter's C code relative to the original seeds. Considering the maturity and stability of the Lua project, this increment indicates that the LLM was able to construct non-trivial logical structures that the standard test suite did not cover.

However, performance analysis revealed a critical challenge for purely LLM-based adoption: the computational cost of inference. With an average rate of only 0.06 executions per second (on hardware with an RTX 3060 GPU), input generation time completely dominates the test cycle, making it orders of magnitude slower than traditional fuzzers (such as AFL++), which can achieve thousands of executions per second. The absence of detected critical crashes can be attributed to this low test volume, insufficient to saturate the state space of robust software within only 24 hours.

Therefore, we conclude that LLMs introduce a new dimension to fuzzing: the high individual semantic quality of mutations, as opposed to the low execution volume. For future work, we suggest investigating hybrid architectures, where the LLM acts only periodically to break through complex coverage barriers, while high-performance traditional mutators handle massive exploration around these new seeds. Additionally, model quantization or the use of smaller LLMs (3B or 7B parameters) may offer a better trade-off between semantic intelligence and test throughput.

Artifact Availability

The artifacts associated with this paper, including the source code for the LLM-guided Lua fuzzer, generated seed pools, and the evaluation scripts, are publicly available on Zenodo. The exact version of the artifact used to produce the results discussed in this work can be accessed via the following DOI: https://doi.org/10.5281/zenodo.21730607.

The artifact is distributed under the MIT License, which allows for free reuse, modification, and distribution of the code.

References

[1] Dinesh Deckker and Subhashini Sumanasekara. 2025. The Role of ChatGPT in Software Development and Code Generation: A Review of Opportunities, Challenges, and Future Directions. *TechRxiv Preprints* (May 2025). Preprint, disponível em: https://www.researchgate.net/publication/391807454.
[2] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. CovRL: Fuzzing JavaScript Engines with Coverage-Guided Reinforcement Learning for LLM-based Mutation. *arXiv preprint arXiv:2402.12222v1* (Feb 2024). https://arxiv.org/abs/2402.12222v1
[3] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (2020), 70–76. doi:10.1145/3376122
[4] Yu Huang, Zhiqiang Zuo, Zeyu Sun, Liangcheng Yu, and Tianwei Zhang. 2024. Large Language Models Based Fuzzing Techniques: A Survey. *arXiv preprint arXiv:2401.08753* (2024). https://arxiv.org/abs/2401.08753
[5] Roberto Ierusalimschy. 2016. *Programming in Lua* (4th ed.). Feisty Duck.
[6] Roberto Ierusalimschy. 2020. *Lua 5.4 Reference Manual*. PUC-Rio. https://www.lua.org/manual/5.4/manual.html Documentação oficial da linguagem Lua.
[7] Yuancheng Jiang, Chuqi Zhang, Bonan Ruan, Jiahao Liu, Manuel Rigger, Roland H. C. Yap, and Zhenkai Liang. 2025. Fuzzing the PHP Interpreter via Dataflow Fusion. *arXiv preprint arXiv:2410.21713v2* (Feb 2025). https://arxiv.org/abs/2410.21713v2
[8] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024). https://arxiv.org/abs/2402.19173
[9] Lua.org. 2025. About Lua. https://www.lua.org/about.html Acesso em May. 2025.
[10] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563
[11] Marbux. 2021. Where Lua is Used. https://web.archive.org/web/20210825030848/https://sites.google.com/site/marbux/home/where-lua-is-used Lista arquivada de casos de uso da linguagem Lua.
[12] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2024. A Comprehensive Overview of Large Language Models. *Elsevier Preprint* (October 2024). Preprint, disponível em: https://arxiv.org/abs/2307.06435v10.
[13] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 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)*. ACM, La Jolla, CA, USA, 298–312. doi:10.1145/3622781.3674171
[14] Siberoloji. 2021. Basics of Lua Programming for Nmap NSE. https://www.siberoloji.com/basics-of-lua-programming-for-nmap-nse/ Introdução prática ao uso de Lua com Nmap.

[15] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *Proceedings of the 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, Shanghai, China, 284–290. doi:10.1109/MedAI59581.2023.00044

[16] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. IEEE/ACM. doi:10.1109/ICSE48619.2024.00050

[17] Wenzhang Yang, Cuifeng Gao, Xiaoyuan Liu, Yuekang Li, and Yinxing Xue. 2024. Rust-twins: Automatic Rust Compiler Testing through Program Mutation and Dual Macros Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. doi:10.1145/3691620.3695059