Spring 2023

CS453 Automated Software Testing

Team Project

Differential Testing of Lua Implementations by Structure-based Fuzzing

HyunSik Yoon (20150526)
Jaeho Choi (20223684)
Jiseok Kim (20160165)
Junyoung Choi (20180798)

1 Introduction

Lua is a programming language created in 1993 at Pontifical Catholic University of Rio de Janeiro, Brazil. It is widely used in game development and embedded systems due to its lightweight nature, simple syntax, and high scalability. Lua, with its broad extensibility, has seen the development of various compilers, including GopherLua, which utilizes Golang, and LuaJIT, which incorporates just-in-time functionality, in addition to the officially supported compiler and interpreter. We were interested in whether multiple compilers for the same language can truly behave identically, and we explored methods for checking compatibility.

2 Problem

To check the compatibility of various Lua interpreters and compilers, we planned to create Lua scripts using a fuzzer and conduct differential testing. For this purpose, we needed a fuzzer which is capable of generating Lua scripts. And we discovered the existence of an alternative called fuzzing-lua. However, fuzzing-lua was a fuzzing tool built on the basis of libFuzzer, which had limited functionality for our needs in conducting differential testing. Therefore, we planned to develop more suitable fuzzers which are structure-aware fuzzer and grammar-based fuzzer.

3 How to solve problem

Differential Testing

Our entire differential testing process was designed in two stages. First, input scripts for targets are generated using fuzzing techniques. We explored several techniques in parallel by implementing multiple fuzzers. Details of the fuzzer implementation are described in the following section. Second, the results of the targets' compilation/interpretation for the generated scripts are collected. This process is accomplished by executing the basic compilation/interpretation command of targets. An automated script runs all inputs on each of the four targets. In this project, we determined the equivalence of the results based on success/failure of command which is the easiest to identify.

Using libFuzzer[5]: Parser as Fuzzing Target

fuzzing-lua[1] is designed to test the Zua, Lua lexer written in Zig. The project is based on libfuzzer and fuzzes the built-in lexer of Lua library. Two methodologies were taken for mutating fuzzing-lua. First method was to filter only the grammatically correct output of the fuzzing-lua. But it was later discarded. The results produced by bitwise fuzzing were too slow to generate synthetically correct Lua code and produced only 2-3 valid corpus per minute.
Next method applied to fuzzing-lua was changing the fuzzing target. Because the coverage guide in libfuzzer operates on language C, we targeted the code that runs the Lua virtual machine in code C.

Using libFuzzer: Structure Aware Fuzzing

The libFuzzer and its derivative, fuzzing-lua, are primarily based on coverage-guided mutation-based fuzzing. Coverage-guided mutation-based fuzzing has the advantage of not being limited by input types or grammar, making it convenient and widely applicable. However, it may not be suitable for fuzzing targets with specific grammar or constraints. Therefore, we decided to explore the additional feature of libFuzzer called structure-aware fuzzing.

Structure-based fuzzing in libFuzzer is still a mutation-based approach, but instead of fuzzing inputs randomly, it first analyzes the given structure or grammar and then mutates the analyzed results accordingly. This allows for fuzzing that aligns with the specified structure.

AFL++ Grammar based fuzzing

Furthermore, we investigated the utilization of AFL++'s Grammar Mutator[2]. This mutator, an additional feature of AFL++ inspired by Nautilus[3], allows for grammar-based fuzzing. It takes the grammar of the input program in JSON format and employs it to guide the fuzzing process. While there was a pre-existing Lua grammar file, it was designed for Lua version 5.3. Hence, we reconstructed the grammar to be compatible with version 5.1. (See Lua.json)

Using the Lua 5.1 grammar we created, we generated a set of 100 seeds. After that, we instrumented the official Lua 5.1 compiler by the AFL++ compiler and employed it as the target for the fuzzing process. Employing AFL++'s parallel fuzzing capability, we ran 8 fuzzers for a duration of 20 minutes. To reduce the size of the generated corpus, we applied coverage-based techniques (afl-cmin, afl-tmin) to minimize the number and size of the corpus files. As a result, we obtained a corpus consisting of 1850 files. For a comprehensive description of our experiment process, please refer to the AFL++ fuzzing in depth document.

4 How to evaluate

Target compilers

The sample compilers for diff testing are the following projects.

- Lua Official compiler
- LuaJit: Just-In Time Compiler for Lua
- gopher Lua: Lua Compiler with Golang

Evaluation Method

Each fuzzer performs fuzzing for the same limited time to generate an input corpus. The inputs generated from each fuzzer are compiled by each sample compiler. Generated Inputs evaluated with following metrics

- The tester records whether each sample compiler successfully compiled Input or not. Then, compilation results of each sample compiler to each Input in the Input set generated by each Fuzzers are used for diff testing.

- Code coverage for target compilers

5 Result

Result of Differential Testing

Fuzzer	Seed Count	Diff count	Diff rate
fuzzing-lua	862	42	4.87%
target modified fuzzing-lua (lexer -> parser)	2472	126	5.10%
fuzzer with parser target and custom mutator	1241	-	-
AFL++ Grammar Based Fuzzing	1850	356	19.24%

* Due to the infinite loop of inputs, differential checking on custom mutator generated corpus was failed

Comparing the Quality of Differential Testing Input Sets

To assess the effectiveness of each fuzzing approach in generating high-quality differential testing input sets, we measured the extent to which each input set could explore the official Lua-based compiler.

For this purpose, we utilized gcov to measure the code coverage of the official Lua compiler for each input set. The results of our measurements revealed that the input set (A) generated by the libfuzzer approach, targeting the lexer, achieved a coverage of 36.44% of the total lines. The input set (B) generated by the libfuzzer approach, targeting the parser, achieved 50.04% coverage, and the input set (C) generated by AFL++ Grammar Mutator achieved 62.21% coverage of the total lines. You can see the detailed results here.

Based on these findings, we can conclude that the quality of the input sets follows the order of A < B < C, indicating that the input sets from B and C outperform the input set from A.

6 Conclusion

Differential testing

Differential testing is a traditional method used to test the differences between multiple targets that provide the same functionality, and it performed even better than we expected. Contrary to concerns about what to do if the fuzzing results were the same, we were able to observe different outputs from each Lua compiler and interpreter. However, it is regrettable that we were unable to analyze aspects such as the program's state, as we focused primarily on analyzing the compile or not results.

Coverage

Our experiment compared the quality of differential testing input sets generated by different fuzzing approaches for exploring the official Lua-based compiler. The results demonstrated that the input sets produced by the libfuzzer approach, targeting the parser, and the input set generated by AFL++ Grammar Mutator achieved higher code coverage compared to the input set produced by the libfuzzer approach targeting the lexer, indicating their superior effectiveness in generating high-quality input sets for differential testing.

Validity of our method

1. Parser as fuzzing target

   The outputs generated from targeting the parser produced more significant gcov results compared to the outputs generated from targeting the existing lexer. Although it had lower coverage compared to the grammar-based fuzzer in afl++ that had already been developed, it achieved higher coverage than the coverage achieved by the existing fuzzing-lua.

2. AFL++ Grammar based fuzzing

   AFL++ delivered the highest level of gcov results among the fuzzers we developed, as expected. It confirmed that structure-aware fuzzing is an effective approach for testing compilers, as higher coverage is achieved when the structure is well-designed. This aligns with our initial expectations regarding the benefits of structure-aware fuzzing.

3. Structure aware fuzzing

   Expectations for this method were high, and it was confirmed that a large number of significant inputs were generated. But due to the infinite loop of inputs, differential checking on custom mutator generated corpus failed. Also it recorded the lowest coverage among our attempts.

7 Future Work

· Improve mutation function
· Improve diff testing criteria: execution state checking
· Optimization: Improve code speed
· Add fault localisation: make more accurate evaluation possible

8 GitHub Repository

https://github.com/AchessYoon/lua-structured-fuzz-reduce

9 Final Presentation

https://docs.google.com/presentation/d/1X5X02uSfCESHsL0HsUtUDXjO5TV8kCuBI36bUhIWJCA/edit#slide=id.g226d6840ac2_2_7

10 References

[1] fuzzing-lua
[2] AFL++ Grammar Mutator
[3] Nautilus
[4] ANTLR
[5] libFuzzer
[6] metalua
[7] structure aware fuzzing with libFuzzer