Running KLEE on GNU coreutils

Valentin Huber

Institute of Applied Information Technology
Zürich University of Applied Sciences ZHAW
contact@valentinhuber.me

January 24, 2024

Contents

1 Introduction **1**

2 Background **2**
2.1 A Primer on Symbolic Execution . . . 2
2.2 Symbolic Execution in Practice 2

3 Reproducing the Original Paper **3**
3.1 Project Setup 3
3.1.1 Building coreutils 6.10 on a Current Version of Ubuntu . . 3
3.1.2 Using an Old Version of Ubuntu 3
3.1.3 LLVM 3
3.1.4 Running KLEE 3
3.2 Analyzing a Single Run 3
3.2.1 Gathered Metrics 3
3.2.2 Comparison to the Original Paper 3

4 Comparing Runs **3**
4.1 Metrics Distribution 3
4.2 Influence of Testing Timeout 3

5 Testing More Recent Versions of coreutils **3**
5.1 Differences in Testing Setup 3
5.2 Findings 3

6 Discussion **3**
6.1 Research Questions 3
6.2 Produced Artifacts 3
6.3 Future Work 3

Bibliography **3**

1 Introduction

KLEE [1] is an open source, symbolic execution based, advanced fuzzing platform. It was introduced in the seminal paper titled "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs" in 2008. In their article, Cadar *et al.* present their work and evaluate it on a diverse set of programs. The most prominent of those is the GNU coreutils suite, in which ten fatal errors were found.

Ever since then, KLEE has not only matured as a fuzzer, it has also been used extensively as a platform for other researchers to build on top of, as I have discovered in [3]. As an introduction to the practical side of fuzzing, I attempted to answer the following questions about KLEE:

1. Reproducing the original paper (see Section 3)
   (a) Can the current version of KLEE be run on coreutils version 6.10 (as tested in the original paper)?
   (b) Can the same metrics as measured in the original paper still be measured?
   (c) How do the measured metrics compare to what was published 15 years ago?
2. Examining the statistical distribution of results over different fuzzing times (see Section 4)
   (a) How does the non-determinism in KLEE influence the variance in the results between different test runs?
   (b) How do different testing timeouts influence results?
3. Testing more recent versions of coreutils (see Section 5)
   (a) What needs to change in the test setup to test more recent versions of coreutils?
   (b) How do the results from testing different versions of coreutils differ?

All experiments were run on a virtualized server with the following specs: AMD EPYC 7713 64C 225W 2.0GHz Processor, 1 TiB RAM, 2x 25GiB/s Ethernet.

2 Background

What follows is a short explanation of the application of symbolic execution in fuzzing. For more extensive background, I refer to some of my previous work [3], [4].

Remember that KLEE is an open-source, symbolic execution based fuzzer. It takes LLVM bytecode from the program under test (PUT) as its input, runs its analysis on it. KLEE then outputs some statistics about the run, inputs to the PUT that crash it, and inputs that, when executed, cover all branches KLEE has examined during its analysis.

2.1 A Primer on Symbolic Execution

KLEE is a fuzzer based on symbolic execution. This means that instead of executing a PUT with a concrete value, it instead runs through the instructions and maps relationships between data in memory (such as variables and user input) to mathematical formulas. So an instruction like %result = **add i32** %a, %b would be mapped to the logical relationship $\phi_k = \phi_i + \phi_j$. Conditional jumps are mapped to conditions on these variables for both outcomes of the condition, so the instruction %isPositive = **icmp sgt i32** %result, 0 would be represented with $\phi_k > 0$ and $\phi_k \leq 0$ respectively.

The set of all conditions along a certain path through the PUT are called the *path condition*. It can be passed to s satisfiability modulo theories (SMT) solver (KLEE uses STP [5] as a default), which returns values for all user inputs, such that the PUT is forced down the exact path represented by the path condition.

This is the major advantage of symbolic execution based fuzzing, as compared to ordinary fuzzing. By not using concrete values, but instead logical representations of user input, it essentially runs through the PUT with all possible user inputs *at the same time*. So if the solver returns that no inputs satisfy the passed formula, we have proven that such inputs simply do not exist. To be able to do this, it accepts the huge overhead of translating the code to formulas and then solving them.

2.2 Symbolic Execution in Practice

Symbolic execution in fuzzing has several major challenges to overcome. I have previously discussed them in detail [3], but would like to give a short summary here:

- Environment interactions (such as file system interactions) in general are opaque to the fuzzer and cannot be mapped to logical formulas. KLEE deals with this by solving the path constraint before the instruction in question and then uses concrete values in the call. This abandons the claim on completeness symbolic execution typically has, but is often the only feasible way to still continue analysis.

- The second major challenge in symbolic execution is what is known as *path explosion*. Because the number of program states grows exponentially with the number of instructions, for all but the most simple programs it is not feasible to calculate the entire state space. KLEE deals with this by reducing the search space to actually executable instructions, using advanced data structures, and examining paths through the PUT consecutively, with a user-defined timeout. To maximize the state space and code coverage as quickly as possible, it alternates between two strategies for choosing the next input to evaluate: KLEE either chooses the input that promises to increase the coverage the most and a random input to prevent the execution from getting stuck in a certain subtree of the PUT.

- KLEE needs to model the entire memory of a process. This is straight-forward as long as variables are used directly but becomes a challenge when pointers are involved. This is especially true if the value of these pointers depends on user input, since this would require KLEE to model all possible addresses having all possible values, which instantly explodes the memory consumption and number of states to examine and is thus infeasible. KLEE deals with this by representing such pointer operation as array accesses where the accessed object is copied as often as necessary to model all possible results, including error states.

- As programs become more complex, the path constraints become increasingly long and solving them contributes more and more to the fuzzer's runtime. KLEE applies some advanced optimizations, like query splitting and more

general optimizations, or a cache of previous results, which often solve supersets the query they are a solution to. Finally, KLEE defines a timeout, after which the solver is interrupted and analysis is continued at an other branch.

3 Reproducing the Original Paper

I'm basing my experiment setup on the original paper [2], the FAQs in the project's documentation [6] and the tutorial on testing coreutils version 6.11 [7].

3.1 Project Setup

3.1.1 Building coreutils 6.10 on a Current Version of Ubuntu

3.1.2 Using an Old Version of Ubuntu

3.1.3 LLVM

3.1.4 Running KLEE

3.2 Analyzing a Single Run

3.2.1 Gathered Metrics

3.2.2 Comparison to the Original Paper

4 Comparing Runs

4.1 Metrics Distribution

4.2 Influence of Testing Timeout

5 Testing More Recent Versions of coreutils

5.1 Differences in Testing Setup

5.2 Findings

6 Discussion

6.1 Research Questions

6.2 Produced Artifacts

6.3 Future Work

Bibliography

[1] "KLEE symbolic execution engine." (2024), [Online]. Available: `https://klee.github.io` (visited on Jan. 24, 2024).

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[3] V. Huber, "Challenges and mitigation strategies in symbolic execution based fuzzing through the lens of survey papers," Dec. 2023. [Online]. Available: `https://github.com/riesentoaster/review-symbolic-execution-in-fuzzing/releases/download/v1.0/Huber-Valentin-Challenges-and-Mitigation-Strategies-in-Symbolic-Execution-Based-Fuzzing-Through-the-Lens-of-Survey-Papers.pdf`.

[4] S. Flum and V. Huber, "Ghidrion: A ghidra plugin to support symbolic execution," Bachelor's Thesis, Zürich University of Applied Science — Institute of Applied Information Technology, Jun. 2023. [Online]. Available: `https://valentinhuber.me/assets/ghidrion.pdf`.

[5] "The simple theorem prover." (2024), [Online]. Available: `http://stp.github.io` (visited on Jan. 24, 2024).

[6] "OSDI'08 coreutils experiments." (2024), [Online]. Available: `https://klee.github.io/docs/coreutils-experiments/` (visited on Jan. 24, 2024).

[7] "Tutorial on how to use KLEE to test GNU coreutils." (2024), [Online]. Available: `https://klee.github.io/tutorials/testing-coreutils/` (visited on Jan. 24, 2024).