Advanced Software Testing and Reverse Engineering Lab 2

INTRODUCTION

Automated generation of software inputs or tests comes in two main flavors: symbolic and population based.

Symbolic generation of software tests relies on the ability of modern SMT solvers to quickly solve satisfiability problems for propositional logic with arithmetic constraints. By constructing such formulas for code-paths, symbolic execution engines construct the inputs required to reach a new piece of code deterministically. They then use some form of backtracking in order to try to reach full branch coverage.

Population-based methods use a guided random process for constructing new inputs/tests that try to trigger new code branches. They recombine previously tried inputs/tests guided by a heuristic that measures the proximity to conditions that trigger new branches.

In this Lab, you will construct and experiment with both these approaches, understand how they work, learn how to use them, and discover their strengths and weaknesses.

LEARNING OUTCOMES

After completing this assignment, you will be able to:

- *Build genetic algorithms for generating new test inputs.*
- *Use and understand modern population-based software testing tools such as EvoSuite (http://www.evosuite.org/).*
- *Build path constraints and forward these to an SMT solver (such as Z3 https://github.com/Z3Prover/z3) to trigger new code branches.*
- *Use and understand modern symbolic execution engines such as KLEE (https://klee.github.io/)*
- *Symbolically execute programs, analyze the generated paths and performance.*

INSTRUCTIONS

This assignment continues with the code you wrote for Lab assignment 1. You can work on any problem from the RERS challenge.

Task 1: symbolic/concolic execution

The goal of this subtask is to give you in-depth experience with the inner workings of Symbolic and/or concolic execution. Symbolic execution works by tainting the memory, monitoring for branches, constructing a path constraint, and forwarding this to an SMTsolver. You build on top of the code for Lab assignment 1.

We will use the Z3 Java API. Study the example for solving a sudoku provided on Brightspace. You have to first construct the path constraint, which basically is a conjunction of all assignments, computations, and conditional jumps (if statements) covered by a trace. Compute the path constraint first for an example input trace, take care that:

- New symbolic variables should be introduced for each new MyBool, MyInt, and MyString object.
- A single static assignment should be used to distinguish each consecutive change of the same variable (this includes assignments and computations).
- The symbolic variables of the input MySting objects should be easily distinguishable from the others.

Forward the path constraint to the SAT solver using the API and extract the assignment that satisfies the path constraint. Provide the trace as input to the program and check the result.

Provide a search wrapper around the path constraint code (depth-first, depth-first, up to you…) and compute a convergence graph. Compare it to the graphs produced by your genetic algorithm.

Task 2: KLEE vs AFL

KLEE is a state-of-the-art symbolic execution engine based on LLVM. Follow the instructions on GitHub to install KLEE and adapt RERS Reachability problems for use by KLEE.

Your task is to compare the performance of KLEE and AFL to all Reachability problems. Give them the same amount of runtime and analyse the obtained reachability targets. Your focus is to highlight and explain the differences between fuzzing and concolic execution. Include the following elements:

1. Descriptions of AFL and KLEE. How do they work? What are they good at?
2. Experiments of both AFL and KLEE on the RERS 2019 Reachability problems.
3. An analysis of the results.

The Driller paper (see Resources) contains good examples of result presentations. Use these for inspiration, although some will be too detailed to include in your report, be selective!

RESOURCES

Slides from Lectures 3,4

Study:

1. Automatic test case generation: what if test code quality matters? https://dl.acm.org/citation.cfm?doid=2931037.2931057
2. A detailed investigation of the effectiveness of whole test suite generation https://link.springer.com/article/10.1007%2Fs10664-015-9424-2
3. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study https://dl.acm.org/citation.cfm?doid=2699688
4. Disposable testing: avoiding maintenance of generated unit tests by throwing them away https://dl.acm.org/citation.cfm?id=3098414
5. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Cristian Cadar, Daniel Dunbar, Dawson Engler
6. SAGE: whitebox fuzzing for security testing. Godefroid, Patrice, Michael Y. Levin, and David Molnar.
7. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. Stephens, Nick and Grosen, John and Salls, Christopher and Dutcher, Andrew and Wang, Ruoyu and Corbetta, Jacopo and Shoshitaishvili, Yan and Kruegel, Christopher and Vigna, Giovanni
8. Symbolic execution for software testing: three decades later. Cadar, Cristian, and Koushik Sen.