AI for Software Testing and Reverse Engineering - Lab 1

INTRODUCTION

Software contains many errors that are difficult to find. Methods for locating such errors come in two flavors: static and dynamic analysis. Static analysis searches the code for patterns that indicate the presence of errors such as a use-after-free vulnerability. Dynamic analysis instead executes the code, monitors this execution, and aims to find inputs that trigger vulnerabilities such as segmentation faults. This course gives an overview of different techniques for dynamic analysis, which can be divided into three categories:

- *Black-box – the code or internal software structure is not used*
- *Grey-box – uses parts of the internal structure/code as guidance*
- *White-box – fully interprets the internal structure/code*

The first lab focusses on ***grey-box methods****. You will start by building a simple* ***black-box fuzzer*** *that continuously probes a program with random input, hoping to cause problems. You will then improve this fuzzer by guiding it using* ***grey-box*** *details such as the branch coverage, a flow analysis, or input taints. Finally, your task is to compare your fuzzer to the state-of-the-art. The programs that we will use throughout this class come from the 2020 RERS (reverse engineering of reactive systems) challenge.* Specifically, we will be using the *Sequential Problems* from challenge.

- *http://rers-challenge.org/2020/*

These are highly obfuscated pieces of software that take strings as input and return strings as output. On Brightspace, you can find a folder containing a Docker file and a shellscript. The Docker file can be used to build a Docker container containing the relevant problems, as well as a multitude of programs that you will use throughout this course.

LEARNING OUTCOMES

After completing this assignment, you will be able to:

- *Build fuzzers from scratch.*
- *Use instrumentation to gather grey-box information.*
- *Use grey-box information to build smarter fuzzers that use search.*
- *Use and understand AFL (http://lcamtuf.coredump.cx/afl/).*
- *Analyze differences between fuzzing results.*

INSTRUCTIONS

Task 0: Preparations

Read the content on Brightspace on fuzzing, tainting, branch distance, hill-climbing and the document *docker_instruction.pdf*.

Download the folder containing the Docker file and the shell script from Brightspace. Build the container following the instructions that are given in *docker_instructions.pdf.*

The RERS problems are located in the RERS directory. These are highly obfuscated pieces of software that model reactive systems that read symbols (“iA”, “iB”, …) from stdin and print output to stdout (“oZ”, “oY”, …). Different input sequences trigger the execution of different code branches. There are two kinds of RERS challenges: Reachability and LTL. The goal of the first RERS challenge is to discover which of the error branches at the top of the file are reachable. The goal of the LTL challenge is to answer a set of given linear temporal logic (LTL) queries about the code. For the purpose of this lab, we focus on the Reachability problems.

We will work with the Java files and instrument them using JavaParser (https://javaparser.org/). We have already written the code instrumentation for you. The instructions for running the instrumentation can be found on the following repository:

- https://github.com/apanichella/JavaInstrumentation

For this assignment, there is no need for you to do the instrumentation yourself. You only need to write code for your solution using the instrumented files. Of course, if you are curious about how we have done the instrumentation, you can always have a look at the Java files that we have used to instrument the problem files.

A simple random fuzzer is already implemented. Compiling and executing the instrumented Java file should start this fuzzer. Your job is to make this fuzzer smarter.

Task 1: branch distance

For this task, you will have to implement the logic that can compute the sum of branch distances for a given input trace. You should only need to modify *FuzzingLab.java* for this assignment. To save you some time and to make it easier for you to implement your solution, we have instrumented the RERS problems such that each time that we have encountered a new branch, the method *encounteredNewBranch* is called every time a *myIf* is called. The *myIf* statements are executed immediately before every if-branch. You will need to implement the logic for the *encounteredNewBranch* method i.e., you would need to compute branch distances and use it to guide the fuzzer. For each branch, the condition, the value of the condition, and the line number where the branch is located in the original file is given to the method. It is up to you to use this information in any way you see fit to make your fuzzer smarter.

The condition is represented as a (near) binary tree of *MyVar* variables. Every *MyVar* contains either a primitive value, a unary operator (“!”) and a reference to another *MyVar*, or a binary operator (“&”, “|”, “<=”, “>”, …) and two such references. Print some of the branches you encounter to understand how this represents an if-branch.

The value is the actual value of the if-branch. The line number is the line number of the if-statement in the original Java file. You can use this to compute branch coverage.

Write code that computes a branch distance for each encountered if-branch from the condition, i.e., the distance from making the branch change its value.

Answer the following questions in **max 1 A4**, provide some details of the computation:

- How many unique branches were you able to visit on the problems you tested?
- Using which set of input traces did you manage to visit the greatest number of branches?
- List five input traces that achieved the lowest sum of branch distances.

Task 2: search

Now that you are able to compute the branch distance for a given input trace, you can use this (grey-box) information to guide the fuzzer in the generation of the input traces. How to do this is entirely up to you, but we advise building a simple hill-climber:

1. Compute the sum of branch distances for a current trace
2. Try X random permutations on an input trace and compute their sum of branch distances
3. Select a permutation that lowers the sum of branch distances
4. If none exists, select a random permutation
5. Update the current trace to the selected permutation
6. Reset the system, execute the current trace, and iterate

Implement this simple search strategy and try to reach all code branches for the RERS Reachability problems. Compared to the random fuzzer from Task 1, how many more unique branches were you able to visit using your search strategy? Give a comparison by runing both versions of the fuzzer on some of the Reachability problems. Which unique error codes were you able to reach using each version of the fuzzer? Which version managed to reach more unique error codes? For the comparisons, consider plotting the convergence graph (number of unique branches over time or number of unique error codes over time) for each version of the fuzzer. Write it down on **max 1 A4**.

Task 3: AFL

For this task, you are asked to apply AFL to the RERS Reachability problems. AFL is a state-of-the-art fuzzer that uses genetic algorithms and branch coverage for guidance. Instructions on how to set up AFL for the RERS problems are available on GitHub:

- https://github.com/apanichella/JavaInstrumentation/blob/main/docs/fuzzing_rers.md

Compare the performance of AFL with both the random fuzzer and your smart fuzzer. The findings/crashes directory contains all crashes AFL found, in this case there are the reachability errors. By running these through the normal (uninstrumented) code, you can obtain all the found reachability statements. In your comparison answer the following questions:

- Does AFL reach more unique reachability statements (error codes) than your own smart fuzzer? What about the number of unique branches?
- Investigate the traces that were used by AFL to find the error codes. Did your own fuzzer also generate the same traces?

For the comparison, you can again plot the convergence graph. Write down all your findings in your report in **max 1 A4**.

RESOURCES

The lab is run in a Docker container, the Docker container can be build using the Docker file that is provided on Brightspace.

For the first lab, you will need the RERS problems, see: *http://rers-challenge.org/2020/*. These will be downloaded when the Docker container is being built. Copy the Reachability problems into the *JavaInstrumentation* folder to make instrumentation easier.

You should read and understand the fuzzingbook chapter on mutation-based fuzzing:

https://www.fuzzingbook.org/html/MutationFuzzer.html

PRODUCTS

A small report of max three A4 pages, excluding visualization, answering the questions from the 3 tasks. The report can be extended with at most two A4 pages if visualizations are used.

An archive (tar/zip) containing the code for computing the results.

Make sure you have also provided some instructions on how to run your code.

ASSESSMENT CRITERIA

You can either pass or fail this assignment. We expect everyone to pass. You have to complete all lab assignments. Submissions of which the report text does not fit into three A4 pages (excluding visualizations) will not be evaluated; the submission will automatically receive a fail.

Code that does not compile/run will not be evaluated; the submission will automatically receive a fail.

Your work will be evaluated based on its completeness (having done all tasks), the correctness of the implementation, and demonstration that you understand the results in the analysis.

When deemed insufficient, you will receive feedback and will be given a one-week grace period to fix any shortcomings.

SUPERVISION AND HELP
There will be lab sessions every Wednesday, where the teachers and TAs will be available to answer any questions you may have. The preferred way to ask questions is through Mattermost.
SUBMISSION AND FEEDBACK
The submission is through Brightspace. You will receive feedback within one week after the deadline.