PROJECT REPORT

COSC6386
Mangesh Soundarya
Aasa Nithin
Vitalii Zhukov

GitHub link: https://github.com/vvzhukov/COSC6386_project

INTRODUCTION

Fuzzing is an important concept in Software Testing which provides unexpected, random, invalid inputs a computer program to detect vulnerabilities that are missed out in manual code inspection. But on the downside the traditional fuzzing techniques fail to exercise all the possible behaviors when the input size is extremely large.

Concolic tracing does try to benefit in some ways, but it fails when there are no sample inputs available. To overcome all these shortcomings, symbolic fuzzing can be used. In symbolic fuzzing we understand the behavior of the program without executing it. Using symbolic fuzzing we can obtain the inputs for a specific block of code or while obtaining a particular output.

The objective behind this project is to develop a symbolic fuzzer to understand its working, effectiveness, shortcomings and application. The project aim is to develop a tool based on a symbolic fuzzer that generates all possible paths to a given depth, collects the information about the constraints and possible inputs for the given program and provides solutions for them.

We made use of the advanced symbolic fuzzer available at the https://www.fuzzingbook.org/ in the fuzzing book as the base of our tool along with standard libraries importLib and argparse.

In our project, we were able to successfully reap the benefits of symbolic fuzzer , we were able to obtain all the possible paths and constraints for a given program. We were able to trace the location of the code that generated that constraint. Our solution is also not tracing the unsatisfiable paths and normally process the code bases with includes it.

However the simple symbolic fuzzer wasn’t beneficial when our programs had loops and recursions. It also fails during variable reassignment. Hence, we used the Advanced Symbolic Fuzzer which overcame the drawbacks of simple symbolic fuzzer.

TOOL

The symbolic fuzzer makes use of control flow graph from the starting point then generate all the possible paths to given depth using max depth using get_all_paths. It also collects the constraints that were encountered while traversing, prints them and shows possible solutions. The unsatisfied paths are not traced,

Usage

```
python3 ./checker.py –-file example2 –-iter_max 20
```

Output

```
File: example2
Set maximum iterations to 20
Set maximum tries to 10
Set maximum depth to 10
-------------------------------------------------------------
Function: abs_value . Found total paths:  40
------------------------------
Path# 0
Constraints: ['z3.And(x == _x_0)', '(_x_0 < 0)', '_v_0 == -_x_0']
Solution: {'x': -1/2}
------------------------------
Path# 1
Constraints: ['z3.And(x == _x_0)', 'z3.Not(_x_0 < 0)', '_v_0 == _x_0']
Solution: {'x': 0}
------------------------------
…
```

Here ‘File’ is the inspected file, maximum iterations, tries and depth are the parameters for the advanced fuzzer, ‘Function’ is the name of currently processed function. ‘Constraints’ and ‘Solutions’ are showing collected constraints and solutions for each path respectively.
By default maximum iterations, maximum tries and maximum depth are set to 10.

FEATURES

We can use individual python files or a single python file with consolidated functions to generate the above mentioned outputs using inspect library (get members, is function).

Generating all paths using get_all_paths() to recursively retrieve all paths in the function.

Extracting all constraints using extract_constraints() in such a way that it can be executable directly using Z3 as it’s our default solver.

Some python program variables need variable reassignments and its achieved using rename_variables() this can be tracked using pnode which can be found in the output generated by tool.

We have given args for max_tries, max_iter,max_depth where we can explicitly mention how many tries, iteration and depth (from parent to child) we want to execute.

LIMITATIONS AND FUTURE WORK

- Trace UNSAT cores and highlight them in the output
- Create additional tools to run the reviewed functions with the generated variables
- Tool itself was not tested thoroughly
- Properly process files with both: .py extension and without it
- Try different constraint solvers
- Check if it is possible to 'seed' the z3 solver in order to get different outputs

SUMMARY

Our understanding from this is, symbolic execution could not determine and detect all the possible failures under certain circumstances when execution was stopped at a lower depth. In addition they are computation extensive and they provide a reasonable gradient of exploration. But they can be used when we want to understand all characteristics of the program and also to test for specific inputs to be present in the program.