

SYMBOLIC EXECUTION

FINAL PROJECT REPORT

COURSE: COSC 6386 PROGRAM ANALYSIS AND TESTING

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF HOUSTON



SUBMITTED BY:

Ankur Khanna (1841253)

Adarsh Kalyampudi (1898447)

Gautham Varadarajan Shanmugaraj (1890686)

INDEX

- I. Introduction to Fuzzing
- II. Symbolic Fuzzing
- III. Using the Fuzzer
- IV. Shortcomings
- V. Future Work
- VI. Conclusion
- VII. References

Github Repository:

https://github.com/ankurkhanna0405/COSC6386_Project.git

I. Introduction - Fuzzing

- Fuzzing or Fuzz Testing is an automated quality assurance technique that is used to find errors in code and loopholes that might cause a security breach.
- The process works by randomly bombarding the test subject with different permutations of random data, known as fuzz, in an effort to make the software reveal vulnerabilities.
- The software tool that is used to detect possible causes for any such vulnerabilities is called a fuzzer.
- **Advantage:** Even though Fuzz testing might seem like a simple process, it has a high benefit-to-cost ratio and can often uncover serious flaws usually ignored during the development and testing of software.
- **Disadvantage:** Fuzz testing cannot give feedback about the overall security and/or effectiveness of the program. So it is usually combined with other testing methods such as black box testing, beta testing, and other debugging methods.

II. Symbolic Fuzzing

- Traditional fuzzing methods have a major drawback, they cannot figure out all possible behaviours of a system when the input space is very high or is complex.
- Some programs might have one or more paths of execution of the program that can only be traversed with very specific inputs due to constraints, this could be a very small portion of the entire input space but still is essential to be tested.
- Traditional fuzzing methods depend on randomness to generate the required inputs. So, paths that have a low probability of occurring have very low chance of getting tested through these methods.
- When the input space is big, depending on randomness to generate and reproduce all possible inputs is not an ideal method.
- Fuzzing methods are mainly used in vulnerability analysis of software systems and programs.

III. Using the Tool

- Before we use the tool, we should first download the files from the github repository:
https://github.com/ankurkhanna0405/COSC6386_Project.git
- Next, we should place the .py file that we want to test inside the examples folder.
- The execution command is, python ./AdvancedSymbolicFuzzer.py filename.py
- We then pass the file name as input to the main code file, SymbolicFuzzer.py, when executing it. This makes sure the file name is passed to the fuzzer functions.
- The output we get are the set of possible inputs that could be passed, to the program that we want to test, without giving any errors that could occur because of input.
- Each time we run the program, for the Simple Fuzzer, we get a different set of possible inputs, in random. These inputs pertain to one particular path.
- The working of simple symbolic fuzzer is such that it chooses one particular path, extracts all constraints in that path, and then solved using z3.
- It explores a certain path to completion before attempting another.
- When using the advanced fuzzer, we get random possible inputs to each of the possible paths of execution.
- It explores the graph in a step-wise manner, expanding every possible execution one step at a time. This way we achieve complete coverage.

```
*****-----*****
Fuzzer output for Check Triangle
Path 0
Constraint found is: ['(a == b)', '(a == c)', '(b == c)']
Z3 solver solution for the above constraint is: {'a': 2, 'b': 2, 'c': 2}
Path 1
Constraint found is: ['(a == b)', '(a == c)', 'z3.Not(b == c)']
Z3 solver solution for the above constraint is: {}
Path 2
Constraint found is: ['(a == b)', 'z3.Not(a == c)']
Z3 solver solution for the above constraint is: {'a': 3, 'b': 3, 'c': 4}
Path 3
Constraint found is: ['z3.Not(a == b)', '(b != c)', '(a == c)']
Z3 solver solution for the above constraint is: {'a': 6, 'b': 5, 'c': 6}
Path 4
Constraint found is: ['z3.Not(a == b)', '(b != c)', 'z3.Not(a == c)']
Z3 solver solution for the above constraint is: {'a': 7, 'b': 8, 'c': 9}
Path 5
Constraint found is: ['z3.Not(a == b)', 'z3.Not(b != c)']
Z3 solver solution for the above constraint is: {'a': 10, 'b': 11, 'c': 11}
```

Fig.
Output for one of our
examples.

VIII. Shortcomings

- In the case of Simple Symbolic Fuzzers, it cannot deal with variable reassignments and other complex input data types.
- It also has no concept of loops and fails to account for them.
- In the case of advanced fuzzers, they always execute till a fixed height, that is established beforehand, irrespective of bugs and errors. That is, symbolic execution is wide but shallow.
- Symbolic execution is computation intensive. That is, they are able to reproduce a bigger input space and have better coverage, but they don't check for magic bytes.
- Our tool does not trace the unsatisfied paths. Given the time, after different methods and various attempts at It, we were not able to come up with a working solution find unsatisfied paths and their scores.

VI. Future Work

- One way to eliminate shallow depth is by using concolic execution, this would allow us to go deeper than pure symbolic execution. This could be one possible addition to our tool.
- Other fuzzing methods could also be added to the tool so there are a wide variety of fuzzers that could be tried for complex programs or in cases where symbolic fuzzing is not feasible. For example, Search based fuzzing could be used when random fuzzing doesn't give enough required information and symbolic fuzzing takes resources than can be spared.
- We were unable to come up with a solution to trace the unsatisfied paths (UNSAT cores), so that could also be finessed in the future.
- A GUI can be developed to make the usage of our tool much simpler for people not used to execution python programs. This way we could just drag and drop the .py files in to he gui and it would give our output along with the path marked on the CFG.

VII. Conclusion

- We have learned from our experience with our symbolic fuzzers project, is that symbolic execution is one way of analysing a program to determine what inputs cause which part of the program to execute. It is well suited in certain scenarios where the program relies on certain values present in the code. But when such values are present, its use case lessens.

VIII. References

1. Fuzzingbook – Symbolic Fuzzer:
<https://www.fuzzingbook.org/html/SymbolicFuzzer.html>
2. Wikipedia – Fuzzing: <https://en.wikipedia.org/wiki/Fuzzing>

X-----X-----X