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

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

I. Introduction - Fuzzing

- Fuzzing or Fuzz Testing is a quality assurance technique that is used to discover coding errors and security loopholes in software.
- It requires bombarding the test subject with large quantities of random data, known as fuzz, in an effort to cause it to crash.
- A software tool known as a fuzzer may be used to detect possible causes for any vulnerabilities discovered.
- Barton Miller, a professor at the University of Wisconsin, invented fuzz testing in 1989.
- **Advantage:** Fuzz testing might be easy, but it has a high benefit-to-cost ratio and can often uncover serious flaws that are ignored during the development and testing of software.
- **Disadvantage:** It cannot, however, provide a full picture of a program's overall security, quality, or effectiveness, and it works best when combined with comprehensive black box testing, beta testing, and other validated debugging methods.

II. Symbolic Fuzzing

- Traditional fuzzing methods have a major drawback, they cannot figure out all possible behaviours that a system may have, particularly when the input space is very high.
- Frequently, a particular path of execution of the program can only be traversed through with very specific inputs, this may constitute a very small fraction of the total input space but still is essential to be tested.
- Traditional fuzzing methods depend on chance 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 space to be explored is large, however, relying on randomness to produce values that we want is a bad idea.
- It is used extensively in vulnerability analysis of software, especially binary 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 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 a particular path, and extract the constraints in that path, which is 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 more step-wise manner, expanding every possible execution one step at a time. This way we achieve complete coverage.

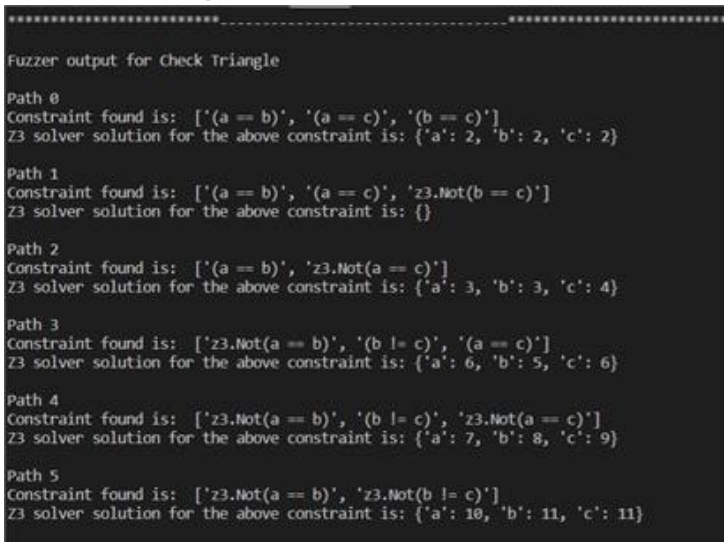

Fig.
Output for one of our examples.

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 stop execution at a predetermined depth without throwing an error. That is, symbolic execution is wide but shallow.
- Symbolic execution is computation intensive. Which means that specification based fuzzers are often able to generate a much larger set of inputs, and consecutively more coverage on programs that do not check for magic bytes, such that they provide a reasonable gradient of exploration.
- 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 would be an acceptable middle ground when random fuzzing does not provide sufficient results, but symbolic fuzzing is too heavyweight.
- We were unable to come up with a solution to trace the unsatsfied 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