COSC 6351: Software Engineering
Project: Fuzzing `bc` with Mutations

Group Members:

Soodeh Atefi
Email: satefi@central.uh.edu

Md Rafiqul Islam Rabin
PeopleSoft ID: 1797648
Email: mrabin@central.uh.edu

Date: February 08 , 2020.

1 Hypothesis

- We create a mutation fuzzer with 7 different mutation techniques: 3 from random mutation-based fuzzing technique and 4 from AFL mutation-based fuzzer.
- We create more input candidates by applying those 7 mutation-based fuzzing technique on the seed inputs of `bc` tool.
- We run those newly generated input candidates against `bc` tool to detect number of valid inputs, bugs and coverage information.
- We compare the performance of random mutation-based fuzzing operators with AFL mutation-based fuzzing operators.
- We apply multiple mutations to compare against the performance of single mutation.
- We compare the performance of mutation on arithmetic operators against the mutation on numbers.

2 Steps for Evaluation

- We implement the mutation fuzzer in Python with 7 mutation operators. The three random mutation operators are (M1) Deleting random character, (M2) Inserting random character, and (M3) Flipping random character. And four AFL mutation operators are AFL techniques: (M4) Simple arithmetics, (M5) Walking bit flips, (M6) Known integers, and (M7) Walking byte flips.
- Generating random inputs is not good starting points for target based fuzzing like fuzzing `bc` tool as doing these create maximum invalid inputs. Therefore, we select a set of valid inputs from `bc` tool to start with. After that we run our fuzzer against the seed inputs of `bc` tool to have more newly generated mutated test inputs.
- An input is valid if it passes by `bc` tool and the input is buggy if `bc` tool fails (i.e. crash immediately, or hang for long time, or give different output for different run) on it. Using gcov tool we can also have the information of coverage (i.e. statement, function, branch). We generate 1000 new inputs for each different mutation and compute these metrics.
- We analyze the metrics (i.e. valid inputs, buggy inputs, coverage info) of random mutation and AFL mutation of previous step 3, separately. Say we can also compute the average cosine distance of generated test inputs.
- Instead of applying a single mutation at a time, we can perform multiple mutation (i.e. apply all 7 mutations) for excessive mutated inputs.

We generate 1000 new inputs with multiple mutation and compute the metrics.

- As `bc` tool is a calculator program, we can mutate arithmetic operators and numbers separately for another comparison. We generate 1000 new inputs with mutating arithmetic operators and numbers separately and compute the metrics.

3 Mutation Techniques

We have created a mutation fuzzer with 7 different mutation techniques: 3 from random mutation-based fuzzing technique (Insert, Delete, and Flip) and 2 from AFL mutation-based fuzzer (Bit flip and Byte flip) and 2 newly generated mutators (Arithmetic mutation and Operator mutation). Beside those, we also performed multiple fuzzing with those mutators (i.e. Multiple Random, Multiple AFL, and Multiple (Random + AFL)). We have applied our mutation fuzzer on the seed inputs of `bc` tool.

The results of running newly generated inputs on 100 programs shows in 1.

Mutation Approach	**# Mutated Programs**	**# Invalid Programs**	**# Statement Coverage (Total = 1863 lines)**
Random (Insert)	100	78	1127
Random (Delete)	100	71	1126
Random (Flip)	100	81	1155
AFL (Bit flip)	100	95	1094
AFL (Byte flip)	100	97	1092
Multiple Random	100	95	1167
Multiple AFL	100	98	1125
Multiple (Random + AFL)	100	100	1124
Operator Only	100	100	1097
Number Only	100	45	1084

Table 1: Number of mutated programs, invalid programs and statement coverage for different mutation operations.

Our mutation fuzzer consists of seven different mutation techniques. Those are:

3.1 Random Mutators

- **Insert**: In this technique, we add a random character at random position.
- **Delete**: In this technique, we delete a character from random position.

- **Flip**: In this technique, we flip a random character at random position.

3.2 AFL

- We have implemented **'walking bit flips'** which consists Flipping a single bit, Flipping two bits in a row, and Flipping four bits in a row. The number of flipped bits differs from 1 to 4 (randomly). For a random input, we converted input to ASCII value and then converted to binary. We considered 8 bits and number of flipped bits randomly have been chosen in the range of 1 to 5.
- In **'walking byte flips'** we did the same approach but we chose 8 number of flipped bits (randomly).

3.3 Multiple Random

We Applied all of our 3 random mutation techniques (Insert, Delete, and Flip) one by one on our seed programs (called "Multiple Random").

3.4 Multiple AFL

We Applied all of our 2 AFL mutation techniques (Walking bit flips and Walking byte flips) one by one on our seed programs (called "Multiple AFL").

3.5 Multiple (Random + AFL)

Here, we apply mutation multiple times on our seed programs with all random and AFL mutations.

3.6 Operator Mutation

In this mutation techniques, we focused on mutating only arithmetic operator (+, -, *, /). For example, given a program, we iterator over inputs, and if we found an operator, we replace it with other operators or delete it from program. Note that, for each operator, we perform this mutation just once in program.

3.7 Number Mutation

Same as arithmetic operator mutation techniques, but here we perform mutation on numbers [0-9] only.

4 Observations

We generate 100 mutated programs with 17 input seeds of `bc` tool.

- Among random mutations, Delete has lowest invalid programs and lowest statement coverage. On the other hand, Flip is highest in generating invalid programs and statement coverage.
- For AFL, byte flip generates more invalid programs and cover more statements than bit flip.
- AFL has highest invalid programs but lowest statement coverage compared to Random mutation techniques.
- Mutation on arithmetic operators "Operator Only" has highest invalid programs and statement coverage compared to mutation on numbers "Number Only".
- "Multiple (Random + AFL)", and mutation on arithmetic operators "Operator Only" have the highest number of invalid programs among other techniques. However, mutation on numbers "Number Only" has lowest number of invalid programs.
- "Multiple Random" mutation technique has the highest number of statement coverage among other techniques. And, mutation on numbers "Number Only" has lowest number of statement coverage.

5 References

- Fuzzing Book: https://www.fuzzingbook.org/html/Fuzzer.html
- AFL: https://github.com/google/AFL
- `bc` tool: https://ftp.gnu.org/gnu/bc