CS639

PROGRAM ANALYSIS, VERIFICATION AND TESTING

ASSIGNMENT 1: COMPLETE A FUZZING LOOP

NAME: NEELU LALCHANDANI

ROLL NUMBER: 231110031

**Objective:**

Mutate the inputs in a way that `maximizes` coverage within a small time budget.

**Implementation:**

The following functions were implemented in Python to achieve the objective:

* **- def compareCoverage(curr\_metric, total\_metric)**
* **- def mutate(input\_data)**
* **- def updateTotalCoverage(curr\_metric, total\_metric)**

Let us look at these one by one:  
  
1**. compareCoverage(self, curr\_metric, total\_metric):** This method compares the coverage metrics between the current execution (curr\_metric) and the total coverage metrics (total\_metric) gathered so far. It is used to determine if the current execution resulted in an improvement in coverage.

* curr\_metric: This is a list that represents the coverage information for the current execution. It contains elements that describe which statements or branches of the code were executed during the current run.
* total\_metric: This is a list that accumulates the coverage information from all previous executions. The method compares the lengths of curr\_metric and total\_metric. If curr\_metric is longer (i.e., it covers more statements or branches) than total\_metric, it returns True, indicating that the coverage has improved. Otherwise, it returns False, indicating that there has been no improvement in coverage.

2. **updateTotalCoverage(self, curr\_metric, total\_metric):** This method is responsible for updating the total coverage metric (total\_metric) with the coverage information from the current execution (curr\_metric).

* curr\_metric: This is the coverage information obtained from the current execution, represented as a list.
* total\_metric: This is the cumulative coverage information gathered from all previous executions, also represented as a list.

The method performs the following steps:

It extends total\_metric with the elements from curr\_metric. This means that the coverage information from the current execution is added to the cumulative coverage information.

After merging the coverage information, it uses the list(set(...)) pattern to remove duplicates and ensure that the coverage information is unique.

Finally, it sorts the resulting list to make it easier to work with and maintain a consistent order.

3. **mutate(input\_data) :** It introduces various mutation types to diversify the input data and explore different code paths in the fuzzing process.

keys = list(input\_data.data.keys())

for i in keys:

    x = random.choice([10, 20, 30])

    random\_addition = random.randint(-x, x)

    random\_xor = random.randint(-x, x)

    input\_data.data[i] += random\_addition

    input\_data.data[i] ^= random\_xor

For each key in the input\_data.data dictionary, this code snippet applies two types of mutations:

* **random\_addition:** It adds a random integer between -x and x to the existing value associated with the key. This introduces variation in the input data.
* **random\_xor**: It performs a bitwise XOR operation with a random integer between -x and x on the existing value. This also introduces variation.

mutation\_type = random.choice(['0', '1', '2', '3', '4','5'])

if mutation\_type in ['0','1']:

    input\_data.data[i] = flip\_random\_bits(input\_data.data[i],random.randint(2, 5))

elif mutation\_type in ['2']:

    input\_data.data[i] = inject\_bits(input\_data.data[i], 0b01, random.randint(0, 8))

elif mutation\_type in ['3','4','5']:

    input\_data.data[i] = apply\_random\_bitmask(input\_data.data[i])

Here, mutation\_type is randomly selected from the list ['0', '1', '2', '3', '4', '5'], which represents different mutation operations. Depending on the selected mutation\_type, one of the following mutation operations is applied:

* If mutation\_type is '0' or '1', it calls the flip\_random\_bits function to flip a random number of bits (between 2 and 5) in the existing value.
* If mutation\_type is '2', it calls the inject\_bits function to inject the bit pattern '01' at a random bit position (between 0 and 8) in the existing value.
* If mutation\_type is '3', '4', or '5', it calls the apply\_random\_bitmask function to apply a random bitmask operation to the existing value.

These mutation operations introduce randomness and variation into the input data, which is crucial for effective fuzzing. The combination of addition, XOR, bit flipping, bit injection, and bitmask operations provides a diverse set of mutations to explore different code paths in the target program.

**Assumptions**

**Datatype:** The datatype of inputs in input\_data.data dictionary is assumed to be integer.

**Limitations:**

1. **Limited Mutation Types:** Although the mutation function includes several mutation types (addition, XOR, bit flipping, bit injection, bitmask), it may not cover all possible mutation scenarios. Depending on the nature of the target program, there may be specific mutation types or operations that are not addressed.

2. **Fixed Mutation Magnitudes**: The magnitude of mutations (e.g., the range of random values added or XORed) is fixed (10, 20, or 30) and not adaptively adjusted based on the context or the nature of the variables. This fixed magnitude may not be suitable for all scenarios.

3. **No Handling of Edge Cases:** There is no specific handling of edge cases or boundary values, which are often important to test for vulnerabilities or unexpected behavior in programs.

4. **Deterministic Randomness:** The random values generated for mutations are based on Python's built-in random module, which uses a deterministic pseudo-random number generator. This means that mutation sequences are repeatable, which may not accurately reflect the variability of real-world inputs.