Program Analysis, University of Stuttgart, Winter 2023/24

Exercise 3: Test Generation: Fuzzing, Symbolic and Concolic Execution

Deadline for uploading solutions via Ilias:
December 14, 2023, 11:59pm Stuttgart time

Task 1 Fuzzing [19 points]

Suppose we generate inputs for the following piece of code using the American Fuzzy Lop (AFL) fuzzer (note: `A`, `B`, etc. denote individual statements, whose details are irrelevant):

```
A
B

if (C) {
  D
} else {
  E
  while (F) {
    G
    H
  }
  I
}
J
```

Subtask 1.1 Control Flow Graph [6 points]

Draw a control flow graph of the code, where nodes are basic blocks. Your graph should have entry and exit nodes.

Subtask 1.2 Coverage Information [10 points]

Assume AFL has already generated five inputs, which trigger executions that cover the following lines:

Input	Covered lines	Coverage as measured by AFL
Input 1	1, 2, 4, 5, 14	
Input 2	1, 2, 4, 6, 7, 8, 12, 14	
Input 3	1, 2, 4, 6, 7, 8, 9, 10, 8, 12, 14	
Input 4	1, 2, 4, 6, 7, 8, 9, 10, 8, 9, 10, 8, 12, 14	
Input 5	1, 2, 4, 6, 7, 8, 9, 10, 8, 9, 10, 8, 9, 10, 8, 12, 14	

Indicate which coverage information AFL tracks for these inputs by providing the covered edges. Use the last column of the table to provide your answers and provide a short explanation of your notation below. (Here and in the following question, assume that the "edge hit count" refinement discussed in the lecture is not considered, but only whether edges are covered or not.)

Explanation of notation:

Subtask 1.3 Inputs for further mutation [3 points]

Which of the inputs will be retained for further mutation? Explain your answer.

Task 2 Symbolic Execution [33 points]

For this task, you are given the following Python program.

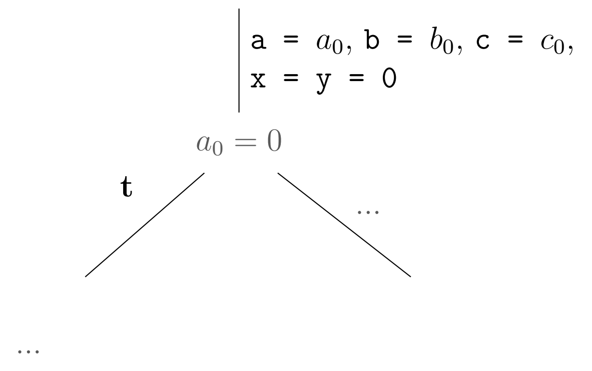
```python
def foo(a, b, c):
  x = y = 0

  if a == 0:
    x += 2
    if b > 0:
      y += 1
      while y + b < 0:
        y += 2
    else:
      y = x + 3
  else:
    y = 9
    if b < 9 or c > 0:
      x = 3

  assert x + y != 9
```

Subtask 2.1 Program Execution Tree [12 points]

Complete the program execution tree (PET) started below, as demonstrated in the lecture. That is, nodes are conditionals and edges correspond to sequences of non-conditional statements. Mark *true* branches with a **t** and *false* branches with an **f**. Also write the assignments that were performed next to each edge. The assertion does not have to be part of the tree. Please use a small square □ to indicate when an execution ends (i.e., for leaf nodes of the PET). In case the tree is infinite, please indicate any repeating patterns.



Subtask 2.2 Path Conditions [9 points]

Each of the leaves in the program execution tree above corresponds to a unique path through the program. For each such unique path, there is a path condition, i.e., a logical formula that must be satisfied for the program to take this path. Collect the path condition for each leaf in your program execution tree. Write down all those path conditions below by listing them from left to right in the tree. Use syntax from mathematics for logical connectives (i.e., $\wedge$, $\neg$, etc.) to make clear that these are logical formulas. You are allowed but not required to simplify the formulas.

1. ____

2. ____

3. ____

4. ____

5. ____

Subtask 2.3 Solve for Inputs [9 points]

An SMT solver (and you as a human) can solve path conditions, i.e., try to find an assignment of a_0, b_0, and c_0 that satisfies the formula. Write down one possible solution (there may be infinitely many) per path condition, i.e., write down values for a_0, b_0, and c_0. If no values of a_0, b_0, and c_0 can satisfy the formula, write down **UNSAT**.

1. ____

2. ____

3. ____

4. ____

5. ____

Subtask 2.4 Assertion [3 points]

For which path does the assertion in line 17 fail, i.e., `x + y != 9` evaluates to false? Write down the path condition:

____

What are the values of `x` and `y` at the assertion in that case? `x` = _______, `y` = _______.

Task 3 Concolic Testing [38 points]

In this task, you will perform concolic testing on the following Python program. Assume all variables are integers.

```
 1 def bar(x, y):
 2   # Program state?
 3   y = x * 2 - 1
 4   # Program state?
 5   if x > 0:
 6     x = baz(x, y + 1)
 7   else:
 8     x = baz(x, y - 1)
 9   # Program state?
10   if y < 2:
11     x = 0
12   else:
13     x = x + 5
14   # Program state?
15   while x < y:
16     x = x + 2
17   y = y - 2
18   # Program state?
19   assert x + y == 0
20
21 def baz(a, b):
22   return a * b - 1
```

Subtask 3.1 First Concolic Execution [12 points]

We start executing the program by calling function `bar` with seed inputs `x` $= 2$ and `y` $= 1$. Complete the table below with the values of the variables `x` and `y` for the concrete and symbolic execution of the program. Write the concrete values down at each program line given in the first column. After the first branch, also write down the path condition under which the program has executed along this path.

Line	Concrete execution	Symbolic execution	Path condition
2	`x` $= 2$, `y` $= 1$	`x` $= x_0$, `y` $= y_0$	-
4			
9			
14			
18			

Subtask 3.2 Generating New Inputs [3 points]

Since concolic execution is a test generation technique, our next goal is to generate a new set of inputs that lead the program down a different path than in the previous execution. For that, take the path condition from the previous execution and negate the conjunct that corresponds to the branch at line 10 in the program.

This results in the following path condition: ________________________________

Solve this path condition for x_0 and y_0 to get test inputs for the program that take a new path (there are infinitely many correct solutions): $x_0 =$ _______, $y_0 =$ _______

Subtask 3.3 Second Concolic Execution [15 points]

We now execute the program a second time, taking the new concrete values for `x` and `y` that you generated in the previous subtask as concrete inputs. Please fill the table below again with the concrete and symbolic state of the program and the path condition.

Line	Concrete execution	Symbolic execution	Path condition
2			-
4			
9			
14			
18			

At the end of this second execution, does the assertion in line 19 fail? _________

Subtask 3.4 Generate More Inputs [8 points]

Concolic execution can successively generate more inputs for `x` and `y` until ultimately each path through the program has been taken once.

Please list two more solutions for `x` and `y` that each cover a new path through the program. For each pair of inputs, also list the final path condition when the program executes on these inputs.

1. `x` = _______, `y` = _______, path condition: ________________________________

2. `x` = _______, `y` = _______, path condition: ________________________________

Task 4 Different Test Generation Approaches [10 points]

In this task, you have to demonstrate your understanding of the four test generation approaches: Randoop, American Fuzzy Lop (AFL), symbolic execution, and concolic execution.
Read the following sentences and indicate whether they are true (T) or false (F).

		T	F
(a)	Randoop is a tool for generating unit tests automatically by using feedback-directed random test generation.	☐	☐
(b)	American Fuzzy Lop (AFL) is primarily used for static analysis of code.	☐	☐
(c)	Symbolic execution analyzes a program to determine what inputs cause each part of a program to execute.	☐	☐
(d)	Concolic execution combines concrete and symbolic execution to improve path coverage.	☐	☐
(e)	Randoop can only generate tests for programs written in Java.	☐	☐
(f)	AFL iteratively mutates and optimizes fuzzing inputs based on code coverage.	☐	☐
(g)	Symbolic execution is not useful for finding security vulnerabilities.	☐	☐
(h)	Concolic execution is not effective in dealing with complex mathematical operations.	☐	☐
(i)	AFL actively steers the execution of the code-under-test toward null pointer dereferences.	☐	☐
(j)	Randoop always requires a formal specification of the software to generate tests.	☐	☐
(k)	AFL can automatically generate unit tests for functions.	☐	☐
(l)	Symbolic execution reasons about the behavior of a program by executing it with symbolic values	☐	☐
(m)	Symbolic execution always generates test cases for every possible path in a program.	☐	☐
(n)	Concolic execution is also known as dynamic symbolic execution.	☐	☐
(o)	Symbolic execution can require more computational resources than concolic execution.	☐	☐