VU Advanced Software Engineering **Sheet 2: (Dynamic) Symbolic Execution and Advanced Testing**

194.021, WS2024

Exercise 1: Symbolic Execution

Symbolically execute with a short-first search strategy function `main` (see below) with the two symbolic variables B and E for `b` and `e` by filling out the table below. For the first Symbolic Execution run, the values for `b` and `e` are both 1.
Give the path conditions for all runs as well as the values used for `b` and `e`. If a Symbolic Execution run encounters an error in the code, highlight this execution run.

```
int sqr_pls(int b, int e) {
  int r = b;
  if(e > 0){
    r=r*b;
    e=e-1;
    if(e > 0)
      r=r+b;
  }
  return r;
}
void main(int b, int e) {
  int r = sqr_pls(b, e);
  if(e % 2 == 0)
    assert(r >= 0);
}
```

Path ID	Parent	Prefix	Input	Path condition
1	-	[]	b = 1 e = 1	$E > 0 \wedge (E-1) \leq 0 \wedge (E-1) \% 2 = 0 \wedge R \geq 0$
2	1	$E \leq 0$	b = 1 e = 0	$E \leq 0 \wedge (E-1) \% 2 = 0 \wedge R \geq 0$
3	2	$E \leq 0 \wedge (E-1) \% 2 = 0$	b = 1 e = -1	$E \leq 0 \wedge (E-1) \% 2 = 0 \wedge R \geq 0$
4	1	$E > 0 \wedge (E-1) > 0$	b = 1 e = 3	$E > 0 \wedge (E-1) > 0 \wedge (E-1) \% 2 = 0 \wedge R \geq 0$
5	4	$E > 0 \wedge (E-1) > 0 \wedge (E-1) \% 2 \neq 0$	b = 1 e = 2	$E > 0 \wedge (E-1) > 0 \wedge (E-1) \% 2 \neq 0$
6	1	$E > 0 \wedge (E-1) \leq 0 \wedge (E-1) \% 2 \neq 0$	UNSAT	

Exercise 2: Dynamic Symbolic Execution - Method Calls

Use Dynamic Symbolic Execution (DSE) to test method `fnc` below. Run the DSE algorithm from the lecture using a long-first search strategy and provide the analysis details required in the table below. `Math.round` is an external function rounding a float number with the round-to-nearest strategy. Replace the call to this function with the concrete value as in the lecture. The prefix is composed of conditions that might be marked as assumptions using `asm(...)`. In column 'input' give the concrete inputs used for the analysis run or UNSAT if the prefix is not satisfiable. For the first Symbolic Execution run, use $a = 0$ and $t = 2$. The path condition is the concatenation of the prefix and the suffix as described in the DSE algorithm. You may not need all rows given in the table below. What does the analysis reveal about the safety of the assertion?

```
void f(int a, int t){
    assume(a >= 0 && t > 1);
    int b, r;
    if (t == 2){
      r = (a + 1) / 2;
      b = a % t;
    } else {
      r = a;
      b = math_round(1.0/t);
    }
    assert(t * r - b == a);
}
```

Path ID	Parent	Prefix	Input	Path condition
1	-			

Exercise 3: Guiding Symbolic Execution Towards Unverified Paths

In this setup, we combine abstract interpretation and symbolic execution to verify the safety of a program. In the first step of the verification, we do syntactic transformations on the input program preparing it for an abstract interpreter. In the second step, we run the abstract interpreter using the interval domain trying to verify as many parts of the program as possible. In the third step, we invoke a symbolic execution engine, which tries to verify all parts that could not be proven safe in the previous step.

```
void verify_me(int x, int y){
    int z = 100;
    if(x > 0){
        y = 2*x;
        z += 1000;
    } else if(x < -10){
        y = -x;
    } else {
        y = -5;
        z = z/2;
    }
    assert(x-y <= z);
}
```

More specifically, verify function `verify_me`. You may assume mathematical integers, that is, that no arithmetic over- or underflows can occur.

In the first step, remove the `assert` statement and insert it at different locations while keeping the semantics of `verify_me` the same. The goal of this syntactic transformation is to allow abstract interpretation to verify the assertion for some paths even if it is not able to do so for all paths. Show the transformed program `verify_me_transformed`.

Next, perform abstract interpretation on `verify_me_transformed`. You do not need to write down the whole abstract CFG; it suffices to specify the abstract states right before the assertions. Illustrate which assertions can be verified.
Finally, verify the safety of the assertions not verified with abstract interpretation by performing symbolic execution on `verify_me_transformed`. Assume the initial seed inputs to be $x = 0, y = 0$, but instead of applying a generic search strategy, guide symbolic execution towards the unverified assertions and stop once these are verified.
What can be an advantage of analyzing a program with such a combined approach in the presence of loops?

Exercise 4: Metamorphic and Differential Testing

- Given is a program π, which returns the length of a given input string. Describe and sketch how you can use metamorphic testing to test π. Describe (in text) at least **three** different metamorphic relations for input and output.
- Consider a function `min`: $\mathcal{P}(\mathbb{Z}) \mapsto \mathbb{Z}$ which returns the minimum of a non-empty (unordered) set of mathematical integers. Define both a metamorphic input relation R^I and a matching metamorphic output relation R^O for testing function `min`.
- Consider the following two testing targets and discuss the applicability of differential testing:
  - The SAT-solver "minisat"
  - The Linux kernel

Exercise 5: Fuzzing

```
int foo(int a, int b, int c) {

    if ( a > b ) {
        if ( b > c ) {
            return a - c;
        }
        return a - b;
    }

    if ( a > b && b < c ) {
        return c;
    }

    int d = b - a;
    if ( d < c ) {
        abort(); // error
    }

    return d;
}
```

You are given a list of inputs from a greybox fuzzing execution on function `foo`:

1. a = 1, b = 1, c = 0
2. a = -1, b = -1, c = -1
3. a = 3, b = 2, c = 0
4. a = -1, b = 2, c = 0

a) Compute the resulting test suite from the initial fuzzer run. Next, provide two more inputs each, such that the fuzzer ...

   1) ... stores them into the test suite.
   2) ... discards them after the execution.

b) Compute the code-coverage percentage of the initial test suite and the updated one from **a)**. Use statement coverage for this task (i.e., a statement is covered if it is executed). Can you improve the coverage of the test suite by keeping the fuzzer running? What information does a fuzzer provide in regards to code reachability?

c) Provide the cost expression depending on `c, d` for taking branch `d < c`. Use the inputs (1), (2) and (4) from above and provide the costs computed using your cost expression.

d) Compute a linear cost function depending on parameter `b`, using linear interpolation as in the lecture. Use the costs for input (2) and (4) from above.

e) Apply fuzzing with prediction starting from input (4) and your linear cost function from **d)** to try to take the branch `d < c`. If your interpolation result for the value of `b` is not an integer, use the round-to-nearest strategy. What are the resulting values of the input parameters? Did you take the branch `d < c`?

Hint: The cost expression provides a concrete positive value showing the distance to flip a condition. The linear interpolation is a naive numeric technique to approximate the actual cost using two known input-cost pairs.