CS243 Lab 6

Spring 2022

Due: May 26, 2022 at 11:59 pm

Directions:

- Submit this assignment on Canvas.
- This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.

Problem 1. Path Sensitive Analysis With Satisfiability Modulo Theories

In this problem you will use an SMT solver to find test cases exhibiting a bug in the following C function:

```
int func(int x, int[] data, int N) {
    int v, z;
    if (0 <= x && x < N) {
        if (x >= 3) {
            x = 2 * x - 5;
        }
        v = data[x]; // line 7
        if (v >= 0 && v < N / 2) {
            z = data[2 * v]; // line 9
        } else if (v >= N/2 && v < N){
            v = (x + 2 * v) / 3;
            z = data[v]; // line 12
        } else{
            z = data[data[0]]; // line 14
        }
        return z;
    } else {
        return 0;
    }
}
```

NOTE: `data` is an array of length `N`.

We are interested in checking whether the program can 'crash' due to array out-of-bounds accesses. First you will translate this function into an SMT-formula. Then you will run an SMT-solver and interpret its output.

You can use either Z3 (https://rise4fun.com/Z3) or CVC5 (https://cvc5.github.io/), which are both high-performing SMT-solvers.

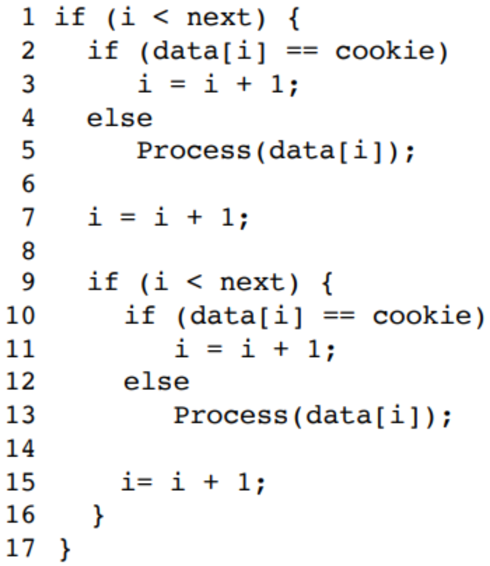
There is a introductory guide at http://www.rise4fun.com/Z3/tutorial/guide, which also serves as a language reference sufficient for our purposes. The tutorial is also applicable to CVC5. The full language specification is available at http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

Your submission will consist of the input to the SMT-solver, the output from running it (sat or unsat, and, if sat, the produced model), along with comments explaining your interpretation of the result. Follow these steps to complete this problem:

1. **Rewrite the program in SSA form.** The first step is to write the function in Static Single Assignment form by assigning all definitions a unique suffix. You need to generate phi-nodes for each branch. At each join point in the CFG you will need

to introduce new definitions for the variables that are defined on either path. At this point, the program is still imperative code, but each variable is defined exactly once.

See the following example which transforms the imperative code on the left into SSA form on the right:

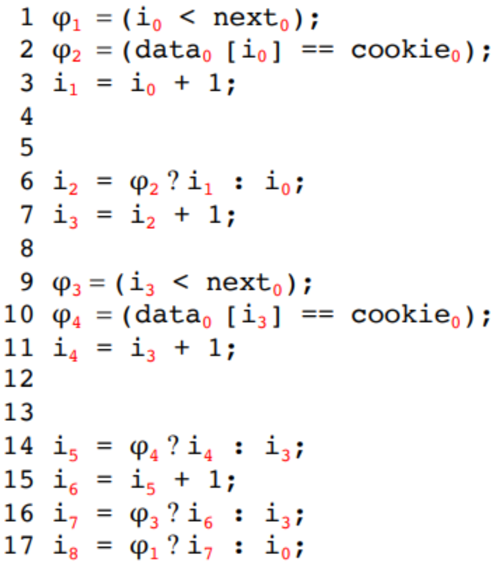


```
if (i < next) {
   if (data[i] == cookie)
      i = i + 1;
   else
      Process(data[i]);

   i = i + 1;

   if (i < next) {
      if (data[i] == cookie)
         i = i + 1;
      else
         Process(data[i]);

      i= i + 1;
   }
}
```



```
φ1 = (i0 < next0);
φ2 = (data0 [i0] == cookie0);
i1 = i0 + 1;


i2 = φ2 ? i1 : i0;
i3 = i2 + 1;

φ3 = (i3 < next0);
φ4 = (data0 [i3] == cookie0);
i4 = i3 + 1;


i5 = φ4 ? i4 : i3;
i6 = i5 + 1;
i7 = φ3 ? i6 : i3;
i8 = φ1 ? i7 : i0;
```

2. **Translation to SMT.** The second step is to translate into an SMT formula. Start your formula with the following lines:

```
(set-logic ALL)
(set-option :produce-models true)
```

If you are using CVC5, also add the following line:

```
(set-option :incremental true)
```

An assignment x3 = e becomes an assertion `(assert (= x3 E))` in SMT, where E is the translation of e. You need to translate int operations at the C level into bit vector operations at the SMT level. Do not use Int in CVC5, as these model mathematical integers. Assume 32-bit 2's complement representation for int. For example, the translation of x = y + 1 is:

```
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(assert (= x (bvadd y #x00000001)))
(check-sat)
(get-model)
```

CVC5 responds with:

```
sat
(model
(define-fun x () (_ BitVec 32) (_ bv1 32))
(define-fun y () (_ BitVec 32) (_ bv0 32))
)

)
```

Here the program is satisfiable with model x = 1, y = 0. Note that the variables x and y are given as functions of no arguments (which must be constants because there are no side effects), and the constants themselves are hexadecimal.

To translate arrays, you will use variables of the sort `(Array (_ BitVec 32) (_ BitVec 32))`. Array dereferences like `data[i]` become `(select data i)` when translating a read, and `(store data i x)` when translating a write. Note that `(store data i x)` returns a new array, whose i's element is now equal to x, and does not modify the original array.

To translate phi-nodes, you must use a logical expression that captures the condition under which the phi node is evaluated. For example, given the following code in SSA form:

```
if (c) {
b1:
    x1 = ...;
} else {
b2:
    x2 = ...;
}
x3 = phi(x1 from b1, x2 from b2);
```

the translation of x3 is `(ite c x1 x2)`. `ite` is short for if-then-else, and evaluates to the second or third argument based on the first.

We provide starter code in bounds-check.smt and an executable example in example.smt

3. **Bounds checks.** The final step is to add an assertion to check each of the array accesses. You need to check that the signed value of the index is in bounds. Further, not all accesses are accessible on all paths, so you need to guard the assertion for a particular access. The assertion should express the execution reaches this access, and it is out of bounds. Note that this will possibly constrain some path variables if the access is nested inside an if statement, for example. You should use the sequence `(push)(assert C)(check-sat)(pop)` for each access, where C is the check for that access. Push/pop allows us to add C to our set of assertions, check satisfiability, and then remove it to add a different C. If you add all the assertions together you will find a path which crash all points simultaneously rather than just at least one.

4. **Interpretation.** Once you find one or more bug, add (`get-model`) after (`check-sat`) within the push/pop sequence for each satisfiable assertion, to print the model found for that bug.

   In writing, interpret the results. Does the result indicate a crash can occur on some concrete path? If not, does this mean there can be no crash for this access? If there is a crash, translate the model into a concrete input represented by a call data = ...; func(...) which causes a crash at the corresponding access. Hint: what happens if the program has more than one bug, and how should the model be interpreted in that case?

5. **Find the smallest value of N that can cause out-of-bounds accesses in line 7.** Do this by asserting the value of N in addition to asserting C described in part 3. Start from N=1,and increase the value of N by 1 until you find a satisfying solution.

6. **Submission.** Provide the Z3/CVC5 inputs, outputs, and your submission.