CS243 Homework 7
Winter 2018

Due: March 14, 2018 at 4:30 pm

Directions:

- Complete the following problems and hand them in at the beginning of class with your name and SUID at the top.
- Remember to complete the corresponding Gradiance quizzes by the start of class on the due date. There are no late days for Gradiance.
- If you need to use one or more late days, hand in your assignment by 4:30pm Thursday or Friday in Gates 407. Slide it under the door if it is locked.
- 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. Pointer Analysis
Perform pointer analysis on the following Java snippet, and answer the following questions.

```java
public class Foo {
    public Foo a;
    public Foo b;
    public static main(String [] args) {
        Foo x = new Foo(); // h1
        Foo y = new Foo(); // h2
        bar(x, y);
        y.a = x;
        bar(y, x);
    }
    public static void bar(Foo p, Foo q) {
        if(p.a != q.b) return;
        p.b = new Foo(); // h3
        baz(p, q);
        qux(p, q);
    }
    public static void baz(Foo c, Foo d) {
        c.a = d;
    }
    public static void qux(Foo p, Foo q) {
    }
}
```
if(c != d.b) bar(c, d);
}
public static void qux(Foo m, Foo n) {
  n.b = m;
  if(m.a != n) baz(m, n);
}

1. What are the hP tuples inferred from this code in a context insensitive analysis? To get you started here’s one: hP(h1, a, h2).

2. Draw the call-graph of the original program and the call-graph after running the cloning-based algorithm for context-sensitive analysis. Do not make clones for recursive calls.

3. How many copies of each procedure are created (for bar, baz and qux)?

4. What are the hP tuples inferred from this code in a context sensitive analysis?

Problem 2. Binary Decision Diagrams
Draw the optimal (minimum number of nodes) BDD for the following expression:

$$\exists(x_2, (x_1 \land x_0) \lor (x_1 \land x_3))$$

1. What is the variable order you chose to create this BDD?

2. What is the minimum number of nodes?

Problem 3. 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:

```c
int func(int x, int[] data, int N) {
  int v, z;
  if (0 <= x && x < N) {
    if (x <= N/2) {
      x = 2 * x;
    }
    v = data[x]; // line 7
    if (v >= 0 && v < N) {
      z = data[v]; // line 9
    } else {
      z = 0;
    }
    if (z < v) {
      data[z] = 0; // line 14
    }
  }
```
return data[data[z]]; // line 16
} else {
  return data[0]; // line 18
}

Assume we care about ‘crashes’ due to array out-of-bounds accesses (even if they don’t cause the program to segfault), and that data has length N. First you will translate this function into a formula. Then you will run a solver and interpret its output. We will use the Z3 tool available at https://rise4fun.com/z3/?menu=0. There is a introductory guide at rise4fun.com/Z3/tutorial/guide which also serves as a language reference sufficient for our purposes. 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 Z3 (copy and pasted from the web editor), 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. Refer to Review 1 for details on SSA.

2. **Translation to SMT.** The second step is to translate into an SMT formula. 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 Z3, 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)
```

Z3 responds with:

```
sat
(model
 (define-fun y () (_ BitVec 32)
 #x00000000)
 (define-fun x () (_ BitVec 32)
 #x00000001)
 )
```
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 \((\text{Array } (_, \text{BitVec } 32) (_, \text{BitVec } 32))\). Array dereferences like \( \text{data}[i] \) become \((\text{select data } i)\) when translating a read, and \((\text{store data } i x)\) when translating a write. Note that \((\text{store data } i x)\) 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:

```plaintext
if (c) {
  b1:
  x1 = ...;
} else {
  b2:
  x2 = ...;
}

x3 = phi(x1 from b1, x2 from b2);
```

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

3. **Bounds checks.** The final step is to add an assertion to check each of the three particular 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 \((\text{push})(\text{assert } C)(\text{check-sat})(\text{pop})\) 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 \((\text{get-model})\) after \((\text{check-sat})\) 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 \( \text{data } = \ldots; \) \( \text{func}(\ldots) \) 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. **Submission.** Print the Z3 input and output, attach it to your homework with the written answers to part 4, and hand it in before class. If you do not have access to a printer, you can email a txt file with the Z3 input and output to the staff mailing list.