Program Testing and Analysis

—Final Exam—

Department of Computer Science
University of Stuttgart

Winter semester 2019/20, February 26, 2020

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)
   - ☐ Slicing computes the subset of all variables in a program that are written to the console.
   - ☒ Static slicing overapproximates the set of statements that may influence a given slicing criterion.
   - ☐ Slicing serializes a concurrent program into a single thread.
   - ☐ Dynamic slicing considers all ways in which statements could influence each other during any execution.
   - ☐ Static slicing considers only those statements that are executed with a given input.

2. Which of the following statements is true? (Only one statement is true.)
   - ☐ The path constraints given to a solver during concolic execution cover the entire execution tree.
   - ☐ The path constraints given to a solver during concolic execution describe all inputs already given the program under test.
   - ☒ The path constraints given to a solver during concolic execution describe a set of inputs that will trigger not yet executed behavior.
   - ☐ The path constraints given to a solver during concolic execution never have a solution.
   - ☐ The path constraints given to a solver during concolic execution always have a solution.

3. Which of the following statements is true? (Only one statement is true.)
   - ☐ Path profiling yields the performance improvement of one program over another.
   - ☒ Path profiling is more accurate in identifying the most frequent path than edge profiling.
   - ☐ Path profiling is as accurate in identifying the most frequent path as edge profiling, but less efficient.
   - ☐ Path profiling is as accurate in identifying the most frequent path as edge profiling, but more efficient.
   - ☐ Path profiling yields the speedup of one program over another.

4. Which of the following statements is true? (Only one statement is true.)
   - ☐ The Daikon invariant detector reports data properties that are true in some but not analyzed executions.
   - ☐ The Daikon invariant detector proofs program properties based on a given set of specifications.
   - ☐ The Daikon invariant detector summarizes API usages into finite automata.
   - ☒ The Daikon invariant detector reports data properties that are true in all analyzed executions.
   - ☐ The Daikon invariant detector identifies method calls that stand out as anomalies.

Part 2 [11 points]

Suppose the following SIMP program and an initial store $s = \{a \mapsto 7, b \mapsto 4, c \mapsto 5\}$:

```
b := !a; if !b = 5 then skip else c := 9
```

- Provide the evaluation sequence of the program using the small-step operational semantics of SIMP. For your reference, the appendix provides the axioms and rules that have been introduced in the lecture (copied from Fernandez' book).

  Solution:

  $\langle$`b := !a; if !b = 5 then skip else c := 9`, $\{a \mapsto 7, b \mapsto 4, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`b := 7; if !b = 5 then skip else c := 9`, $\{a \mapsto 7, b \mapsto 4, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`skip; if !b = 5 then skip else c := 9`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`if !b = 5 then skip else c := 9`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`if 7 = 5 then skip else c := 9`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`if False then skip else c := 9`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`c := 9`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 5\}\rangle$

  $\rightarrow \langle$`skip`, $\{a \mapsto 7, b \mapsto 7, c \mapsto 9\}\rangle$

- Is the program divergent?

  Solution: No

- Is the program blocked?

  Solution: No

- Is the program terminating?

  Solution: Yes

Part 3 [11 points]

This task is about data flow analyses in general and about reaching definitions analysis in particular. Consider a naive variant of the data flow analysis framework described in the lecture, where instead of propagating information along a control flow graph, we propagate information along statements in their lexical order of appearance in the code. This naive analysis would read the source code line by line, starting at the first line until it reaches the end of the file. Whenever the analysis finds another statement, it assumes that this statement may execute right after the previously statement seen statement.

Give an example program where performing a reaching definitions analysis with the naive data flow framework and the data flow analysis framework given in the lecture will produce different results. Clearly explain the difference and why they occur.

Solution:

The following program will be analyzed differently by the naive variant and a regular data flow analysis:

```
1 var x = 10;
2 while (x < 20)
3   x = x - 1;
```

Because a regular data flow analysis is based on the control flow graph, it sees that the statement at line 3 may be executed repeatedly in the loop. In particular, this means that a value written into `x` in one iteration of the loop may be read in the next iteration of the loop. The regular reaching definitions analysis will find the following solution to the data flow equations (where s refers to the line numbers):

s	$RD_{entry}(s)$	$RD_{exit}(s)$
1	$\{(x,?)\}$	$\{(x,1)\}$
2	$\{(x,1),(x,3)\}$	$\{(x,1),(x,3)\}$
3	$\{(x,1),(x,3)\}$	$\{(x,3)\}$

In contrast, the naive reaching definitions analysis will find the following solution to the data flow equations:

s	$RD_{entry}(s)$	$RD_{exit}(s)$
1	$\{(x,?)\}$	$\{(x,1)\}$
2	$\{(x,1)\}$	$\{(x,1)\}$
3	$\{(x,1)\}$	$\{(x,3)\}$

The naive approach is missing the fact the control may also flow "up" from line 3 to line 2. As a result, the reaching definitions computed at lines 2 and 3 are incomplete.

Part 4 [10 points]

This task is about dynamic information flow analysis. Consider the following JavaScript code to analyze:

```
1  var x = getSecret();
2  var y = getInternal();
3  var z = 23;
4  if (x === 42) {
5    z = 24;
6  } else {
7    z = 25;
8  }
9  z = y + z;
10 if (y === 5) {
11   leak(z);
12 }
```

There are three security classes: *secret*, *internal*, and *public*, which are ordered into a lattice such that *secret* is higher than *internal*, and *internal* is higher than *public*. By default, all values are labeled as *public*. Values returned by `getSecret()` are labeled as *secret* and values returned by `getInternal()` are labeled as *internal*. The function `leak()` is an untrusted sink, which should only be reached by *public* information.

Consider a dynamic information flow analysis that considers both explicit and implicit flows. Suppose an execution where `getInternal()` returns 5 and `getSecret()` returns 42.

1. What are the security labels of variables and expressions during the execution? Use the following template to provide your answer.

   Solution:

Line	Variable or expression	Security label of variable or expression (after executing the line)	Security stack
1	`x`	*secret*	(empty)
2	`y`	*internal*	(empty)
3	`z`	*public*	(empty)
4	`x === 42`	*secret*	*secret*
5	`z`	*secret*	*secret*
9	`z`	*secret*	(empty)
10	`y === 5`	*internal*	*internal*

2. Does the analysis detect a policy violation? If yes, what is the reported level of the leaked information? Explain your answer.

   Solution:

   Yes, there is a policy violation. The value `z` that reaches the sink is labeled as *secret*.

3. What does an attacker who knows the source code and who receives the value given to `leak()` at line 11 learn about the value stored in `x`?

   Solution:

   The attackers learns whether `x` contains 42. The reason is that the leaking value of `z` depends on whether `x === 42`.

4. Suppose to analyze the same execution with an information flow analysis that considers only explicit flows. Would it also report a policy violation? If yes, what is the reported level of the leaked information? Explain your answer.

   Solution:

   Yes, there also is violation of the information flow policy. However, in contrast the above, the leaked value `z` is labeled as *internal* only. The reason is that line 9 has an explicit flow of the *internal* value in `y` to `z`, whereas the analyses misses the implicit flow from the *secret* value in `x` into `z`.

Part 5 [12 points]

The lecture has introduced the DeadlockFuzzer tool by Joshi et al., which performs "active testing" to detect deadlocks in a two-stage approach. Another kind of concurrency bug are data races. This task is about finding data races using active testing, and you should describe how to adapt the active testing idea to data races. Please explain how an active data race detection tool would work. We provide the following questions to help you structure your answer. Describe your ideas as precise as possible (within the given space and time) and try to use an example whenever appropriate.

1. Active testing works in two stages, a static analysis followed by a dynamic analysis. What would these two analyses do for an active data race detection tool?

   Solution:

   The first stage is a static analysis that finds potential data races. There are many ways of implementing such an analysis. One approach would be to statically overapproximate the set of shared memory locations by considering all static and non-static fields as shared. Furthermore, the analysis could analyze all thread creation operations and the code executed in the created threads to determine code that could potentially execute concurrently. Based on the identified threads, the analysis could assume that all pairs of accesses of shared memory are potentially concurrent, unless both certainly happen in the main thread. The output of the first stage is a set of pairs of code locations that are potentially involved in a data race, i.e., the code locations are reads and writes of memory, and at least one element of each pair is a write.

   The second stage of the approach is a dynamic analysis that actively modifies the schedule to try to trigger the potential data races (see third question for details).

2. Give an example of a data race that a simple static analysis (i.e., the first stage) might find, but that actually cannot be triggered. Explain your solution.

   Solution:

```
1  // Thread 1:
2  localVar1 = computeValue();
3  if (localVar1 == 42) {
4    someObject.sharedField = 5;
5  }
6
7  // Thread 2:
8  someObject.sharedField = 23;
```

   The simple static analysis described above would determine that the two pieces of code under "Thread 1" and "Thread 2" could potentially execute concurrently. It would also determine that the two write accesses to `sharedField` both write a potentially shared memory location. Hence, it would report a potential data race between the write at line 4 and the write at line 8.

   However, in practice this data race may be infeasible if the value returned by `computeValue` is always different from 42. Since the simple static analysis sketched above does not reason about the feasibility of paths, it cannot determine whether the condition at line 3 may become true, and conservatively assumes that it may be true.

3. Describe how the second stage of your active race detection tool would work.

   Solution:

   The second stage of actively testing for data races takes the set of potential races and tries to trigger them at runtime. To this end, the approach influences the program execution so that it checks before each memory access whether the access is part of a potential data race. If yes, then the approach delays the thread's execution and gives priority to other threads that may reach the other memory access of the potential race. The second stage could be implemented either by instrumenting the program, e.g., by inserting statements right before every memory access, or by modified the runtime environment, e.g., by modifying the Java virtual machine to intercept memory accesses.

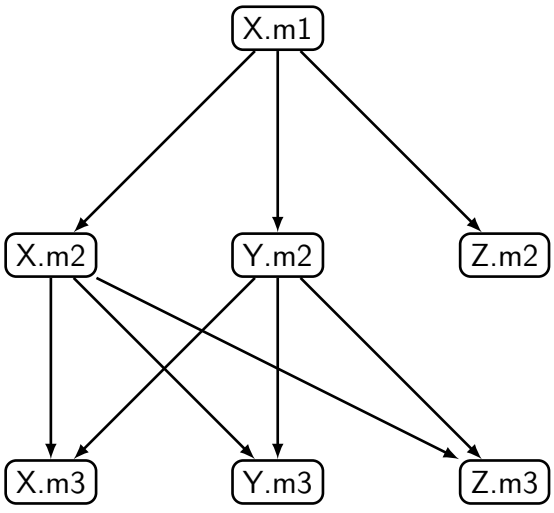
Part 6 [12 points]

This task is about static call graph analysis. You are given the following Java code, where `X.m1` is the entry point into the code:

```
1  class X {
2      void m1() {
3          X x = new Y();
4          Y y = new Y();
5          Z z = new Z();
6          y = z;
7          x.m2(y);
8      }
9      void m2(X a) {
10         a.m3();
11     }
12     void m3() {}
13 }
14
15 class Y extends X {
16     void m2(X b) {
17         b.m3();
18     }
19     void m3() {}
20 }
21
22 class Z extends Y{
23     void m2(X c) {}
24     void m3() {}
25 }
```

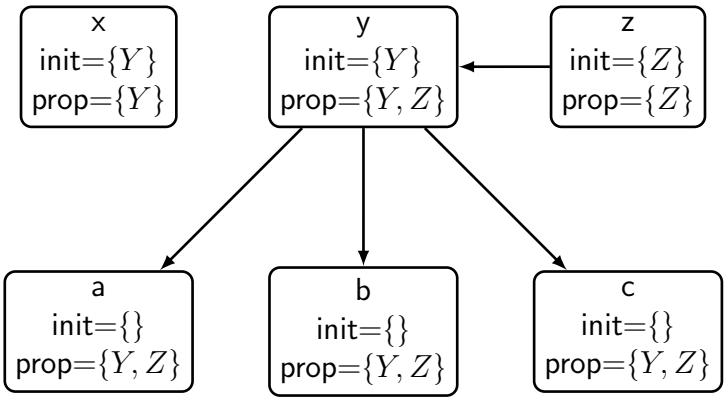
1. Provide the call graph computed by the CHA (class hierarchy analysis) algorithm. Use the following graph template:

   Solution:



2. Using the call graph provided by CHA as a starting point, we now perform the VTA (variable type analysis) algorithm. Give the type propagation graph computed by VTA. For each node, show the initial types and the types after propagation. Use the following graph template.

   Solution:



3. Based on the types computed by VTA, give the call graph that VTA eventually produces, including all nodes and edges (but no more) reachable from the entry point `X.m1`.

   Solution: