Program Testing and Analysis

—Final Exam—

Department of Computer Science
University of Stuttgart

Winter semester 2021/22, March 2, 2021

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.)

   - ☐ Weiser's static slicing algorithm is based on reachability within a control flow graph.
   - ☐ Weiser's static slicing algorithm is based on reachability within an abstract syntax tree.
   - ☐ Weiser's static slicing algorithm is based on reachability within a program execution graph.
   - ☒ Weiser's static slicing algorithm is based on reachability within a program dependence graph.
   - ☐ Weiser's static slicing algorithm is based on reachability within a call graph.

2. Which of the following statements is true? (Only one statement is true.)

   - ☐ The coverage information used by AFL to prioritize inputs is an approximation of function-level coverage.
   - ☐ The coverage information used by AFL to prioritize inputs is exact line coverage.
   - ☐ The coverage information used by AFL to prioritize inputs is exact function-level coverage.
   - ☐ The coverage information used by AFL to prioritize inputs is exact path coverage.
   - ☒ The coverage information used by AFL to prioritize inputs is an approximation of branch coverage.

3. Which of the following statements is true? (Only one statement is true.)

   - ☒ A more precise call graph contains fewer edges.
   - ☐ A more precise call graph contains more edges.
   - ☐ An imprecise call graph is missing edges that may occur during an execution.
   - ☐ A precise call graph connects each node to itself and to at least one other node.
   - ☐ An imprecise call graph is missing method nodes that may occur during an execution.

4. Which of the following statements is true? (Only one statement is true.)

   - ☐ In a concurrent program, the number of thread interleavings is constant.
   - ☐ In a concurrent program, the number of thread interleavings is quadratic in the number of threads and in the number of instructions per thread.
   - ☒ In a concurrent program, the number of thread interleavings is exponential in the number of threads and in the number of instructions per thread.
   - ☐ In a concurrent program, the number of thread interleavings is exponential in the number of threads and linear in the number of instructions per thread.
   - ☐ In a concurrent program, the number of thread interleavings is linear in the number of threads and exponential in the number of instructions per thread.

Part 2 [15 points]

Consider the following SIMP program:

```
while ¬ (!a = !b) do (if !b > 0 then b := !b - 3 else skip)
```

1. Given the initial store $s = \{a \mapsto 5, b \mapsto 7\}$, provide the first nine steps of the evaluation sequence of the small-step operational semantics. For each transition, indicate the name of the axiom or rule that you are using. If multiple rules or axioms are used for a single transition, indicate the one that is at the bottom of the corresponding proof tree.

   Abbreviations you may use:

   - B stands for `(if !b > 0 then b := !b - 3 else skip)`
   - C stands for `¬ (!a = !b)`

   Solution:

   $\langle$`while C do B`$, s\rangle$

   $\xrightarrow{while}$ $\langle$`if C then (B; while C do B) else skip`$, s\rangle$

   $\xrightarrow{if}$ $\langle$`if ¬ (5 = !b) then (B; while C do B) else skip`$, s\rangle$

   $\xrightarrow{if}$ $\langle$`if ¬ (5 = 7) then (B; while C do B) else skip`$, s\rangle$

   $\xrightarrow{if}$ $\langle$`if ¬ False then (B; while C do B) else skip`$, s\rangle$

   $\xrightarrow{if}$ $\langle$`if True then (B; while C do B) else skip`$, s\rangle$

   $\xrightarrow{if_T}$ $\langle$`B; while C do B`$, s\rangle$

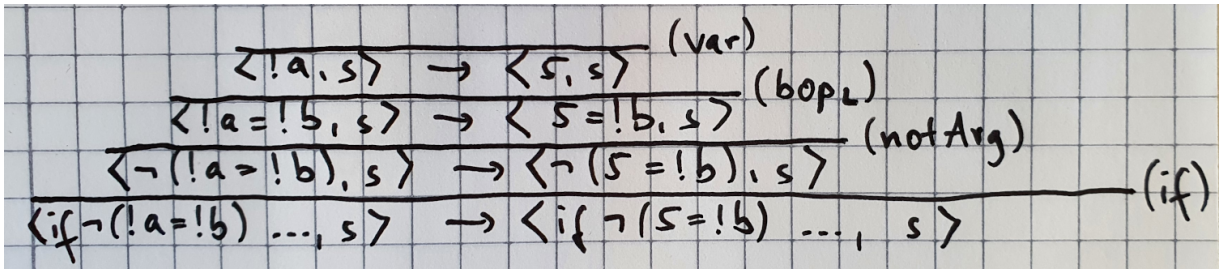
   $\xrightarrow{seq}$ $\langle$`(if 7 > 0 then b := !b - 3 else skip); while C do B`$, s\rangle$

   $\xrightarrow{seq}$ $\langle$`(if True then b := !b - 3 else skip); while C do B`$, s\rangle$

   $\xrightarrow{seq}$ $\langle$`b := !b - 3; while C do B`$, s\rangle$

2. For the first use of the "if" rule above, provide a proof tree that shows why you can use this rule.

   Solution:

3. Which of the following properties does the program execution have or not have. Briefly explain your answer.

   - [x] Divergent
   - [ ] Terminating
   - [ ] Blocked

   Explanation:

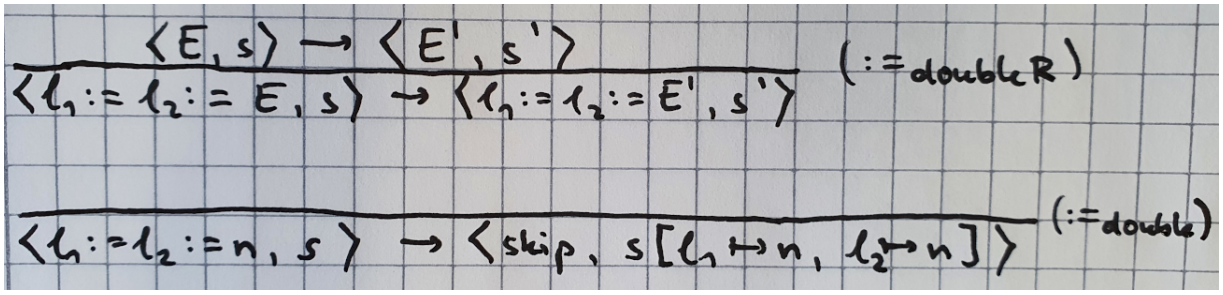
   Divergent, because the evaluation sequence is infinite.

4. Suppose we add a new language feature to SIMP, which allows for assigning to two variables in a single command. For example, `c := d := 23` will write the value 23 into both variables `c` and `d`. Similar to single-variable assignments, the right-hand side of the new multi-variable assignment can be an arbitrary SIMP integer expression.

   Extend the transition rules given in the appendix to accommodate the new language feature. You can adapt any of the existing rules and axioms, and also add new rules or axioms. Give any added or changes rule or axiom:

   Solution:

   We add one new axiom and one new rule:



Part 3 [15 points]

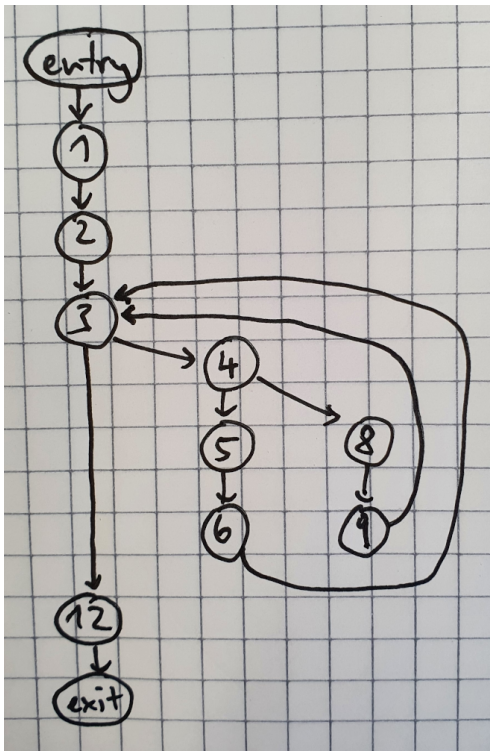
Consider the following JavaScript code:

```
1  a = 4;
2  b = 7;
3  while (c1) {
4    if (c2) {
5      x = a + b;
6      y = a / b;
7    } else {
8      z = a + b;
9      x = a * b;
10   }
11 }
12 a = 3;
```

The following is about performing a very busy expressions analysis on the above code. As the domain of the analysis, consider only non-trivial expressions, i.e., expressions that go beyond a constant or a single variable.

1. Provide a control flow graph for the code. Each node in the graph should correspond to one statement. Include entry and exit nodes. Use the line numbers to label the nodes.

   Solution:



2. Compute the gen and $kill$ sets for each statement. As above, use line numbers to identify statements. Use the following template to provide your solution:

   Solution:

Statement s	$gen(s)$	$kill(s)$
1	$\varnothing$	{a+b, a/b, a*b}
2	$\varnothing$	{a+b, a/b, a*b}
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
5	{a+b}	$\varnothing$
6	{a/b}	$\varnothing$
8	{a+b}	$\varnothing$
9	{a*b}	$\varnothing$
12	$\varnothing$	{a+b, a/b, a*b}

3. Solve the dataflow equations and enter the results into the following template:

   Solution:

Statement s	$VB_{entry}(s)$	$VB_{exit}(s)$
1	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
4	{a+b}	{a+b}
5	{a+b, a/b}	{a/b}
6	{a/b}	$\varnothing$
8	{a+b, a*b}	{a*b}
9	{a*b}	$\varnothing$
12	$\varnothing$	$\varnothing$

4. Given the very busy expressions computed above, what optimization(s) might a compiler apply to the above program?

   Solution:

   The compiler might hoist the evaluation of `a+b` to the beginning of the loop, i.e., before the if-statement. Alternatively, the compiler might move the evaluation of `a/b` and `a*b` to the beginning of lines 5 and 8, respectively.

Part 4 [10 points]

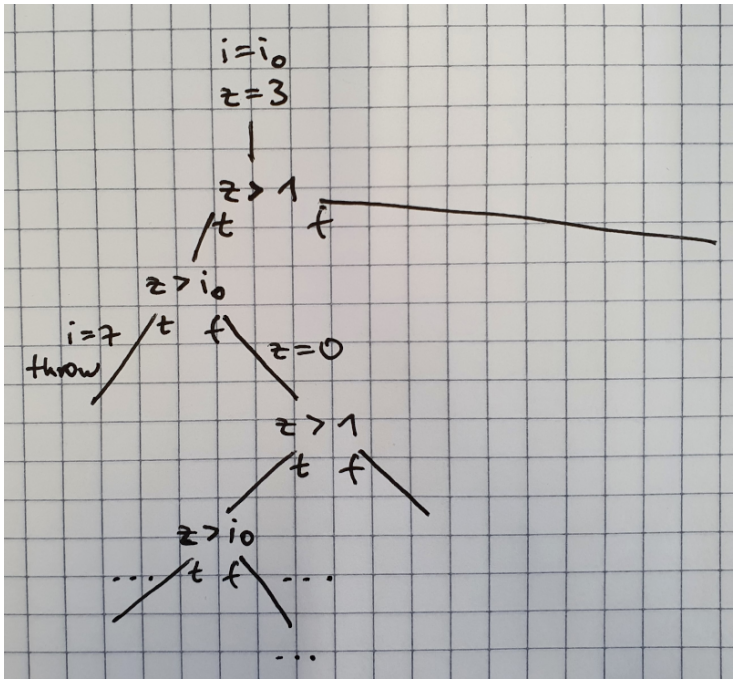
Consider the following JavaScript function:

```
1  function f(i) {
2    var z = 3;
3    while (z > 1) {
4      if (z > i) {
5        i = 7;
6        throw "Error";
7      }
8      z = 0;
9    }
10 }
```

Suppose to use concolic testing to analyze the function, where `i` is considered to be a symbolic variable.

1. Draw the execution tree of the program. If the tree is infinitely large, use "..." to represent repeating parts of the tree.

   Solution:



2. Suppose that concolic testing starts with the following concrete input `i = 5`. Illustrate the execution using the following table.

   Solution:

Line	*After* executing the line		
	State of concrete execution	State of symbolic execution	Path condition
2	$i = 5, z = 3$	$i = i_0$	true
3	$i = 5, z = 3$	$i = i_0$	true
4	$i = 5, z = 3$	$i = i_0$	$i_0 \geqslant 3$
8	$i = 5, z = 0$	$i = i_0$	$i_0 \geqslant 3$

3. What is the formula that concolic testing gives to the SMT solver after the first execution?

   Solution:

   $i_0 < 3$

4. Give a solution for this formula and describe what will happen if the program gets executed with the new input.

   Solution:

   $i_0 = 2$. The program will throw an error at line 6.

Part 5 [10 points]

Consider the following JavaScript code:

```
1 var gotIt = false;
2 var paddedPasswd = "xx" + passwd;
3 var knownPasswd = null;
4 if (paddedPasswd === "xxtopSecret") {
5   gotIt = true;
6   knownPasswd = passwd;
7 }
8 addToLogFile(gotIt);
```

The following is about performing a dynamic information flow analysis on this code. The analysis considers implicit flows, but not hidden implicit flows. There are two security classes, called *secret* and *public*, with *secret* being the top of the lattice and *public* being the bottom of the lattice. Initially, the value stored in `passwd` is labeled as *secret*. The function `addToLogFile` will expose data to a publicly visible space, and hence, is considered a sink. The information flow policy is that only *public* data should flow into `addToLogFile`.

1. Consider an execution where `passwd` is "topSecret". Show the state of the analysis after each executed line by providing the labels of `gotIt`, `paddedPasswd`, and `knownPasswd`, as well as the security stack. If a variable is not defined yet, just indicate a hyphen ("—") instead of its label. Use the following as a template to provide your solution.

   Solution:

Line	After executing the line			
	Security label			Security stack
	`gotIt`	`paddedPasswd`	`knownPasswd`	
1	*public*	—	—	(empty)
2	*public*	*secret*	—	(empty)
3	*public*	*secret*	*public*	(empty)
4	*public*	*secret*	*public*	*secret*
5	*secret*	*secret*	*public*	*secret*
6	*secret*	*secret*	*secret*	(empty)
8	*secret*	*secret*	*secret*	(empty)

2. Is there a violation of the security policy? Briefly explain your answer. *Solution*:

   Yes, because the argument given into `addToLogFile` is labeled as *secret*.

3. Now, consider an execution where `passwd` is "fooBar". Again, show the state of the analysis using the following template.

   Solution:

Line	After executing the line			
	Security label			Security stack
	`gotIt`	`paddedPasswd`	`knownPasswd`	
1	*public*	—	—	(empty)
2	*public*	*secret*	—	(empty)
3	*public*	*secret*	*public*	(empty)
4	*public*	*secret*	*public*	(empty)
8	*public*	*secret*	*public*	(empty)

4. Is there a violation of the security policy? Briefly explain your answer.

   Solution:

   No, because the argument given into `addToLogFile` is labeled as *public*.

5. What information about the password is leaked in the second execution?

   Solution:

   An attacker with access to the log file written by `addToLogFile` will learn that the password is not "topSecret".

6. How to detect such data leakage in a dynamic information flow analysis? Explain your answer.

   Solution:

   One option is to modify the program by introducing spurious writes of `gotIt` and `knownPasswd` in a newly added else-branch. In this newly added branch, there will be two statements:
   ```
   gotIt = gotIt;
   knownPasswd = knownPasswd;
   ```
   As a result, the analysis will label `gotIt` as *secret* no matter whether the if-branch is taken, which will cause the policy to be violated for both the first and the second execution.

Part 6 [6 points]

In the lecture, two variants of the Eraser algorithm for detecting data races have been discussed. While the "simple" variant of the algorithm tracks only the lockset of each shared variable, the "refined" variant also maintains the state each of variable based on a state machine with four states (called "virgin", "exclusive", "shared", and "shared-modified"). Provide an example program that illustrates the need for the refined algorithm. Use the following template to give your solution. The program does not need to exactly follow the syntax of a real-world programming language, as long as the presented code is self-explanatory.

1. Example program:

   Solution:

   ```
   // main thread (running before spawing threads 1 and 2)
   x = 4;


   // thread 1 (running concurrently with thread 2)
   acquire L
   y = x;
   release L


   // thread 2 (running concurrently with thread 1)
   z = x;
   ```

2. Explanation of what happens using the simple variant of Eraser, and why Eraser's behavior is suboptimal:

   Solution:

   Variable `x` is shared between the threads, and hence, will be monitored by Eraser. Consider an execution where all code in thread 1 executes before thread 2. When thread 1 reads `x`, the algorithm determines its lockset to be $C(x) = \{L\}$. Later, when thread 2 reads `x`, the lockset will become empty, as the $locksHeld$ set is empty at this point. The algorithm hence reports a data race because the consistent locking discipline gets violated. However, the code does not really have a data race because the two concurrent access to `x` are both only reading, but not writing, the value. In other words, the reported data race is false positive.

3. Explanation of what instead happens using the refined variant of Eraser.

   Solution:

   Again, consider an execution where all code in thread 1 executes before thread 2. In addition to the locksets as explained above, the algorithm also maintains the state of variable `x`. Initially, the variable is in the "virgin" state. When the main thread writes to it, the state is updated to "exclusive". Once thread 1 executes and reads `x`, the state becomes "shared". Later, when thread 2 executes, the state remains "shared", as no other thread than the main thread has ever written to `x`. As a result, Eraser does not report any warnings, even when the lockset becomes empty (because warnings are issued only in the "shared-modified" state), avoiding the false positive.

Appendix

You may remove the pages of the appendix to allow for easier reading.

For Part 2: Transition rules of small step operational semantics for SIMP (copied from Fernandez' book).

Reduction Semantics of Expressions:

$$\frac{}{\langle !l, s\rangle \to \langle n, s\rangle \ \text{ if } s(l) = n}\ (\text{var})$$

$$\frac{}{\langle n_1\ op\ n_2, s\rangle \to \langle n, s\rangle \ \text{ if } n = (n_1\ op\ n_2)}\ (\text{op})$$

$$\frac{}{\langle n_1\ bop\ n_2, s\rangle \to \langle b, s\rangle \ \text{ if } b = (n_1\ bop\ n_2)}\ (\text{bop})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1' op E_2, s'\rangle}\ (\text{op}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 op E_2, s\rangle \to \langle n_1 op E_2', s'\rangle}\ (\text{op}_\text{R})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1' bop E_2, s'\rangle}\ (\text{bop}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 bop E_2, s\rangle \to \langle n_1 bop E_2', s'\rangle}\ (\text{bop}_\text{R})$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \to \langle b, s\rangle \ \text{ if } b = (b_1\ and\ b_2)}\ (\text{and})$$

$$\frac{}{\langle \neg b, s\rangle \to \langle b', s\rangle \ \text{ if } b' = not\ b}\ (\text{not}) \qquad \frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \to \langle \neg B_1', s'\rangle}\ (\text{notArg})$$

$$\frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1' \wedge B_2, s'\rangle}\ (\text{and}_\text{L}) \qquad \frac{\langle B_2, s\rangle \to \langle B_2', s'\rangle}{\langle b_1 \wedge B_2, s\rangle \to \langle b_1 \wedge B_2', s'\rangle}\ (\text{and}_\text{R})$$

Reduction Semantics of Commands:

$$\frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle l := E, s\rangle \to \langle l := E', s'\rangle}\ (:=_\text{R}) \qquad \frac{}{\langle l := n, s\rangle \to \langle skip, s[l \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}\ (\text{seq}) \qquad \frac{}{\langle skip; C, s\rangle \to \langle C, s\rangle}\ (\text{skip})$$

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle if\, B\ then\ C_1\ else\ C_2, s\rangle \to \langle if\, B'\ then\ C_1\ else\ C_2, s'\rangle}\ (\text{if})$$

$$\frac{}{\langle if\, True\ then\ C_1\ else\ C_2, s\rangle \to \langle C_1, s\rangle}\ (\text{if}_\text{T})$$

$$\frac{}{\langle if\, False\ then\ C_1\ else\ C_2, s\rangle \to \langle C_2, s\rangle}\ (\text{if}_\text{F})$$

$$\frac{}{\langle while\ B\ do\ C, s\rangle \to \langle if\, B\ then\ (C; while\ B\ do\ C)\ else\ skip, s\rangle}\ (\text{while})$$