Program Testing and Analysis
—Final Exam—

Department of Computer Science
University of Stuttgart

Winter semester 2019/20, February 26, 2020

Name, first name:
Matriculation number:

GENERAL GUIDELINES AND INFORMATION

1. Start this exam only after the instructor has announced that the examination can begin. Please have a picture ID handy for inspection.

2. You have 60 minutes and there are 60 points. Use the number of points as guidance on how much time to spend on a question.

3. For multiple choice questions, you get the indicated number of points if your answer is correct, and zero points otherwise (i.e., no negative points for incorrect answers).

4. You can leave the room when you have turned in your exam, but to maintain a quiet setting nobody is allowed to leave the room during the last 15 minutes of the exam.

5. You should write your answers directly on the test. Use a ballpoint pen or similar, do not use a pencil. Use the space provided (if you need more space your answer is probably too long). Do not provide multiple solutions to a question.

6. Be sure to provide your name. Do this first so that you do not forget! If you must add extra pages, write your name on each page.

7. Clarity of presentation is essential and influences the grade. Please write or print legibly. State all assumptions that you make in addition to those stated as part of a question.

8. Your answers can be given either in English or in German.

9. With your signature below you certify that you read the instructions, that you answered the questions on your own, that you turn in your solution, and that there were no environmental or other factors that disturbed you during the exam or that diminished your performance.

Signature:

To be filled out by the correctors:

<table>
<thead>
<tr>
<th>Part</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
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 \( s = \{ a \mapsto 7, b \mapsto 4, c \mapsto 5 \} \):

\[
b := !a; \text{ if } !b = 5 \text{ then skip else } c := 9
\]

- 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).

\[ \langle b := !a; \text{ if } !b = 5 \text{ then skip else } c := 9, s \rangle \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]
• Is the program divergent?

• Is the program blocked?

• Is the program terminating?
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 assume 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.
Part 4 [10 points]

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

```javascript
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.

<table>
<thead>
<tr>
<th>Line</th>
<th>Variable or expression</th>
<th>Security label of variable or expression (after executing the line)</th>
<th>Security stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x === 42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>y + z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>y === 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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.
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?

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

```java
class X {
    void m1() {
        X x = new Y();
        Y y = new Y();
        Z z = new Z();
        y = z;
        x.m2(y);
    }
    void m2(X a) {
        a.m3();
    }
    void m3() {}
}

class Y extends X {
    void m2(X b) {
        b.m3();
    }
    void m3() {}
}

class Z extends Y {
    void m2(X c) {}
    void m3() {}
}
```

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

```
X.m1

X.m2  Y.m2  Z.m2

X.m3  Y.m3  Z.m3
```

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.

```
   X   Y   Z

   a   b   c
```

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`. 
Appendix

Feel free to remove this page for easier reading.

Reduction Semantics of Expressions:

\[
\frac{\langle l, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } s(l) = n}{(\text{var})}
\]

\[
\frac{\langle n_1 \ op \ n_2, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } n = (n_1 \ op \ n_2)}{(\text{op})}
\]

\[
\frac{\langle n_1 \ bop \ n_2, s \rangle \rightarrow \langle b, s \rangle \quad \text{if } b = (n_1 \ bop \ n_2)}{(\text{bop})}
\]

\[
\frac{\langle E_1, s \rangle \rightarrow \langle E'_1, s' \rangle}{(\text{opL})}
\]

\[
\frac{\langle E_1 \ op E_2, s \rangle \rightarrow \langle E'_{1 \ op} E_2, s' \rangle}{(\text{opR})}
\]

\[
\frac{\langle b_1 \land b_2, s \rangle \rightarrow \langle b, s \rangle \quad \text{if } b = (b_1 \ and \ b_2)}{(\text{and})}
\]

\[
\frac{\langle b, s \rangle \rightarrow \langle b', s \rangle \quad \text{if } b' = \neg b}{(\text{not})}
\]

\[
\frac{\langle B_1, s \rangle \rightarrow \langle B'_1, s' \rangle}{(\text{notArg})}
\]

\[
\frac{\langle B_2, s \rangle \rightarrow \langle B'_2, s' \rangle}{(\text{andR})}
\]

Reduction Semantics of Commands:

\[
\frac{\langle E, s \rangle \rightarrow \langle E', s' \rangle}{(\text{:=R})}
\]

\[
\frac{\langle l := E, s \rangle \rightarrow \langle l := E', s' \rangle}{(\text{:=})}
\]

\[
\frac{\langle C_1, s \rangle \rightarrow \langle C'_1, s' \rangle}{(\text{seq})}
\]

\[
\frac{\langle \text{skip}; C, s \rangle \rightarrow \langle C, s \rangle}{(\text{skip})}
\]

\[
\frac{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle \text{if } B' \text{ then } C'_1 \text{ else } C'_2, s' \rangle}{(\text{if})}
\]

\[
\frac{\langle \text{if } \text{True} \text{ then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_1, s \rangle}{(\text{ifT})}
\]

\[
\frac{\langle \text{if } \text{False} \text{ then } C_1 \text{ else } C_2, s \rangle \rightarrow \langle C_2, s \rangle}{(\text{ifF})}
\]

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