Feedback — week 5 quiz

Help

You submitted this quiz on **Sat 29 Nov 2014 8:46 AM PST**. You got a score of **47.00** out of **61.00**. You can attempt again, if you'd like.

Question 1

A static analysis

Your Answer		Score	Explanation
○ is always better than testing			
● analyzes a program's code without running it	✓	3.00	
○ is a kind of real analysis for solving numeric equations			
○ analyzes the fixed, or *static* portions of a program			
Total		3.00 / 3.00	

Question 2

Which of the following are advantages of static analysis over testing?

Your Answer		Score	Explanation
✓ A static analysis can analyze programs that are not necessarily executable on their own, e.g., libraries	✓	1.00	The analysis can consider all possible inputs to functions; they need not be specified, as with tests.
✓ A static analysis is more scalable than testing	✗	0.00	This is usually not true. Arbitrarily large programs can be tested (i.e., executed), but many static analyses are limited in the size of the programs they can consider.
A static analysis runs faster than testing	✓	1.00	Not always! It depends on the level of precision, etc. of the analysis, and how much coverage there is in the test suite.
✓ A static analysis can reason about *all* program paths, not just some of them	✓	1.00	This is the key benefit of abstraction, as used by static analysis. The abstraction makes it possible to consider an approximation of all runs.
Total		3.00 / 4.00	

Question 3

The halting problem is the problem of determining, for an arbitrary program and input, whether the program will finish running or continue to run forever. Which of the following statements about the halting problem are true?

Your Answer		Score	Explanation
☐ The halting problem is decidable.	✓	1.00	No, it is *un*decidable.
☑ Many other program analysis problems can be converted to the halting problem.	✓	1.00	This is true. We showed how the question of whether an array indexing expression is in bounds can be reduced to the halting problem (by converting the program we are interested in into a different program such that an answer by a termination analysis implies an answer about the original program).
☐ You cannot solve the halting problem with static analysis, but you can with symbolic execution.	✓	1.00	Not true. The undecidability of the halting problem is independent of the method used to determine whether a program terminates or not.
☑ The halting problem is undecidable.	✓	1.00	This means there is no solution, which can classify all programs' termination behavior perfectly.
☑ You cannot build an automated analysis that proves that a particular program *P* terminates.	✗	0.00	We can prove this statement false by providing a counterexample: The program "x = 5" clearly terminates, and it would be easy to build a tool to recognize this program and say as much. What you cannot do is write a tool indicates a program P terminates for all programs P that do, but no others.

Total	4.00 / 5.00

Question 4

Suppose we have a static analysis that aims to find buffer overflows in C programs. If the analysis is *sound*, then which of the following is true about it?

Your Answer		Score	Explanation
It will not have any false alarms, but may fail to report actual bugs			
It will report all actual bugs, and have no false alarms			
It may miss bugs, and have false alarms			
● It may have false alarms, but will not fail to report actual bugs	✓	3.00	
Total		3.00 / 3.00	

Question 5

A tainted flow is

Your Answer		Score	Explanation
○ A flow from a trusted source to both trusted and untrusted sinks			
● A flow from an untrusted source to both trusted and untrusted sinks	✓	3.00	
○ A flow from a trusted source to an untrusted sink			
○ A flow from an untrusted source to a trusted sink			
Total		3.00 / 3.00	

Question 6

Consider the program below, using the qualified types annotations for tainted flows given in the lecture (shown in comments).

In particular, notice that the variable `fmt` and the argument to `printf` are untainted, while the result of `fgets` is tainted.

Suppose we analyze this with a tainted flow analysis. *This program has no bugs,* but which kinds of analysis **report a false alarm**?

```
/* int printf(untainted char *fstr, ...); */
/* tainted char *fgets(...); */
```

```
char *chomp(char *s) {
  int i, len = strlen(s);
  for (i = 0; i<len; i++)
    if (s[i] == '\n') {
      s[i] = '\0';
      break;
    }
  return s;
}

void foo(FILE *networkFP, untainted char *fmt) {
  char buf[100];
  char *str = fgets(buf, sizeof(buf), networkFP);
  char *str1 = chomp(str);
  char *fmt1 = chomp(fmt);
  printf(fmt1,str1);
}
```

Your Answer		Score	Explanation
✓ flow-**IN**sensitive, context-sensitive	✗	0.00	A context-sensitive analysis will treat the two calls to chomp distinctly, so fmt1 will remain untainted, and thus be considered a legal argument to printf (regardless of the flow- or path-sensitivity of the analysis)
✓ flow-**IN**sensitive, context-**IN**sensitive	✓	1.00	A context insensitive analysis will report an alarm because all calls to chomp are conflated. Since we are passing both str1 and fmt to chomp, where the former is tainted, the output of chomp will always be considered tainted in a context-insensitive analysis, regardless of whether or not it is flow- or path-sensitive.

☐ path-sensitive, context-sensitive	✓	1.00	A context-sensitive analysis will treat the two calls to chomp distinctly, so fmt1 will remain untainted, and thus be considered a legal argument to printf (regardless of the flow- or path-sensitivity of the analysis)
☐ flow-sensitive, context-sensitive	✓	1.00	A context-sensitive analysis will treat the two calls to chomp distinctly, so fmt1 will remain untainted, and thus be considered a legal argument to printf (regardless of the flow- or path-sensitivity of the analysis)
☑ flow-sensitive, context-**IN**sensitive	✓	1.00	A context insensitive analysis will report an alarm because all calls to chomp are conflated. Since we are passing both str1 and fmt to chomp, where the former is tainted, the output of chomp will always be considered tainted in a context-insensitive analysis, regardless of whether or not it is flow- or path-sensitive.
☑ path-sensitive, context-**IN**sensitive	✓	1.00	A context insensitive analysis will report an alarm because all calls to chomp are conflated. Since we are passing both str1 and fmt to chomp, where the former is tainted, the output of chomp will always be considered tainted in a context-insensitive analysis, regardless of whether or not it is flow- or path-sensitive.
Total		5.00 / 6.00	

Question 7

Consider the following code, where the referenced `chomp` function is the same as in the previous question. Suppose we analyze this with a tainted flow analysis. Once again, this program has no bugs, but which kinds of analysis **report a false**

alarm?

```
void bar(FILE *networkFP, char *fmt, int testing) {
  char buf[100];
  char *str = fgets(buf, sizeof(buf), networkFP);
  char *str1 = chomp(str);
  if (testing)
    str1 = "test format";
  printf(fmt,str1);
  if (testing)
    printf(str1);
}
```

Your Answer		Score	Explanation
✓ path-sensitive, context-**IN**sensitive	✗	0.00	
flow-**IN**sensitive, context-**IN**sensitive	✗	0.00	Only a path-sensitive analysis will know that the printf call on the last line will only ever occur with str1 set to a constant, which is untainted, rather than a tainted value.
✓ path-sensitive, context-sensitive	✗	0.00	
✓ flow-sensitive, context-sensitive	✓	1.00	Only a path-sensitive analysis will know that the printf call on the last line will only ever occur with str1 set to a constant, which is untainted, rather than a tainted value.

✓ flow-sensitive, context-**IN**sensitive	✓	1.00	Only a path-sensitive analysis will know that the printf call on the last line will only ever occur with str1 set to a constant, which is untainted, rather than a tainted value.
✓ flow-**IN**sensitive, context-sensitive	✓	1.00	Only a path-sensitive analysis will know that the printf call on the last line will only ever occur with str1 set to a constant, which is untainted, rather than a tainted value.
Total		3.00 / 6.00	

Question 8

Consider the following code, where the referenced `chomp` function is the same as in the previous question. Suppose we analyze this with a a tainted flow analysis. Once again, this program has no bugs, but which kinds of analysis **report a false alarm**?

```
void bar(FILE *networkFP, char *fmt, int testing) {
  char buf[100];
  char *str = fgets(buf, sizeof(buf), networkFP);
  char *str1 = chomp(str);
  if (testing)
    str1 = chomp("test format");
  printf(fmt,str1);
  if (testing)
```

```
    printf(fmt,"how did the test string look?\n");
}
```

Your Answer		Score	Explanation
✓ path-sensitive, context-**IN**sensitive	✗	0.00	
☐ flow-sensitive, context-sensitive	✓	1.00	
✓ flow-**IN**sensitive, context-**IN**sensitive	✗	0.00	
✓ flow-sensitive, context-**IN**sensitive	✗	0.00	
✓ flow-**IN**sensitive, context-sensitive	✗	0.00	
☐ path-sensitive, context-sensitive	✓	1.00	
Total		2.00 / 6.00	

Question Explanation

Even the least precise analysis will not throw a false alarm for this program. This is because fmt is always treated as untainted, no matter whether the calls to chomp are sensitive, or whether the order (or path) of statements is considered.

Question 9

Which of the following are true of *implicit flows*?

Your Answer		Score	Explanation
✓ One occurs when assigning a tainted value to an untainted variable	✗	0.00	This is a direct flow, not an implicit one.
✓ Implicit flows are rarely detected by tainted flow analyses, because detecting them can increase false alarms	✓	1.00	This is true, as stated in lecture.
☐ They only arise in object-oriented languages with dynamic dispatch, since the choice of method to call is implicit	✓	1.00	The example given in lecture does not use object oriented language features, so they are not a prerequisite for implicit flows.
✓ One occurs when assigning an untainted value to an untainted variable, but conditioned on a tainted value	✓	1.00	This is basically the example given in the lectures.
Total		3.00 / 4.00	

Question 10

What is a key advantage of symbolic execution over static analysis?

Your Answer		Score	Explanation
● As a generalized form of testing, when a symbolic executor finds a bug, we are sure it is not a false alarm	✓	3.00	
Symbolic executors consider all possible program runs, while static analyses don't			
Symbolic executors are both sound and complete, while static analyzers can only be one or the other			
Symbolic executors can consider partial programs (e.g., libraries) while static analyzers cannot			
Total		3.00 / 3.00	

Question 11

Symbolic execution, viewed as a kind of static analysis, has which of the following "sensitivities?"

Your Answer	Score	Explanation
Flow-sensitivity		
Context-sensitivity		

○ Path-sensitivity			
● All of the above	✓	3.00	
Total		3.00 / 3.00	

Question 12

Why is *concolic execution* problematic for non-terminating programs?

Your Answer		Score	Explanation
○ Its search strategy is to choose new test cases based on constraints generated by terminating runs			
● Non-terminating programs will consume too many resources	✗	0.00	Possibly, but not necessarily. E.g., the program `while (1);` will simply run forever and consume few resources
○ Non-terminating programs require user interaction, which concolic execution does not handle			
○ Concolic execution takes a breadth-first approach, but non-terminating programs are			

better suited to a depth-first approach

Total	0.00 / 3.00

Question 13

Suppose that `x` and `y` in the following program are symbolic. When the symbolic executor reaches the line that prints `"everywhere"` what will the path condition be?

```
/* assume x and y are both symbolic */
void foo(int x, int y) {
  if (x > 5) {
    if (y > 7) {
      printf("here\n");
    } else {
      if (x < 20)
        printf("everywhere\n");
      else
        printf("nowhere\n");
    }
  }
}
```

Your Answer	Score	Explanation

○ x > 5 ∧ ¬(y > 7) ∧ ¬(x		
○ x > 5 ∧ y > 7 ∧ x		
● x > 5 ∧ ¬(y > 7) ∧ x	✓	4.00
○ ¬(y > 7) ∧ x		
Total		4.00 / 4.00

Question 14

Suppose that `x` in the following program is symbolic. When the symbolic executor reaches the line that prints `"here"` what will the path condition be?

```
void bar(int x) {
  int z;
  if (x > 5)
    z = 5;
  else
    z = 1;
  if (z > 3)
    printf("here\n");
}
```

Your Answer		Score	Explanation
○ ¬(x > 5) ∧ z > 3			
[● x > 5]	✓	4.00	Path conditions only mention symbolic variables, and this is the only condition that needs to be satisfied to reach the desired line.
○ x > 5 ∧ z > 3			
○ z > 3			
Total		4.00 / 4.00	

Question 15

Which of the following are heuristics that symbolic executors use to cover more of the search space?

Your Answer		Score	Explanation
[✓ Choose between two paths based on a notion of priority]	✓	1.00	

✓ Randomly restart the search from the main function	✓	1.00	
☐ Choose between concolic and non-concolic execution	✓	1.00	Concolic and non-concolic execution are two kinds of symbolic execution algorithm, not elements of a single algorithm
✓ Choose between two paths based on whether one reaches program statements not previously executed	✓	1.00	
Total		4.00 / 4.00	