**Question 1 - multiple choice, shuffle**

A static analysis

\*A: analyzes a program's code without running it

B: analyzes the fixed, or *static* portions of a program

Feedback: Not true---it considers all of a program. A static analysis is so named because it works on the code without running it, i.e., the code is static while being analyzed.

C: is a kind of real analysis for solving numeric equations

Feedback: Static analyses work on all programs, not just particular kinds.

D: is always better than testing

Feedback: Not true---a static analysis has downsides that testing does not have, and vice versa.

**Question 2 - checkbox, shuffle, partial credit**

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

\*A: A static analysis can analyze programs that are not necessarily executable on their own, e.g., libraries

Feedback: The analysis can consider all possible inputs to functions; they need not be specified, as with tests.

B: A static analysis runs faster than testing

Feedback: Not always! It depends on the level of precision, etc. of the analysis, and how much coverage there is in the test suite.

C: A static analysis is more precise than testing

Feedback: This is not true: Static analysis *abstracts* an execution, ignoring some details to focus on a property of interest, and to scale. Testing is perfectly precise, because it's actually running the program.

**Question 2 - checkbox, variation 1, shuffle, partial credit**

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

\*A: A static analysis can reason about *all* program paths, not just some of them

Feedback: This is the key benefit of abstraction, as used by static analysis. The abstraction makes it possible to consider an approximation of all runs.

B: A static analysis is more precise than testing

Feedback: This is not true: Static analysis *abstracts* an execution, ignoring some details to focus on a property of interest, and to scale. Testing is perfectly precise, because it's actually running the program.

C: A static analysis is usually more scalable than testing

Feedback: This is not true in general. Arbitrarily large programs can be tested (i.e., executed), but many static analyses are limited in the size of the programs they can consider. OTOH, very simple static analyses can run very fast.

**Question 2 - checkbox, variation 2, shuffle, partial credit**

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

\*A: A static analysis can analyze programs that are not necessarily executable on their own, e.g., libraries

Feedback: The analysis can consider all possible inputs to functions; they need not be specified, as with tests.

B: A static analysis is usually more scalable than testing

Feedback: This is not true in general. Arbitrarily large programs can be tested (i.e., executed), but many static analyses are limited in the size of the programs they can consider. OTOH, very simple static analyses can run very fast.

C: A static analysis runs faster than testing

Feedback: Not always! It depends on the level of precision, etc. of the analysis, and how much coverage there is in the test suite.

**Question 3 - checkbox, shuffle, partial credit**

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?

\*A: Many other program analysis problems can be converted to the halting problem.

Feedback: 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).

B: You cannot build an automated analysis that proves that a particular program *P* terminates.

Feedback: 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.

C: You cannot solve the halting problem with static analysis, but you can with symbolic execution.

Feedback: Not true. The undecidability of the halting problem is independent of the method used to determine whether a program terminates or not.

**Question 3 - checkbox, variation 1, shuffle, partial credit**

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?

\*A: The halting problem is undecidable.

Feedback: This means there is no solution, which can classify all programs' termination behavior perfectly.

B: You cannot build an automated analysis that proves that a particular program *P* terminates.

Feedback: 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.

C: The halting problem is decidable.

Feedback: No, it is *un*decidable.

**Question 3 - checkbox, variation 2, shuffle, partial credit**

(3 points) 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?

\*A: The halting problem is undecidable.

Feedback: This means there is no solution, which can classify all programs' termination behavior perfectly.

B: You cannot build an automated analysis that proves that a particular program *P* terminates.

Feedback: 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.

C: You cannot solve the halting problem with static analysis, but you can with symbolic execution.

Feedback: Not true. The undecidability of the halting problem is independent of the method used to determine whether a program terminates or not.

**Question 4 - multiple choice, shuffle**

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?

\*A: It may have false alarms, but will not fail to report actual bugs

B: It will not have any false alarms, but may fail to report actual bugs

C: It will report all actual bugs, and have no false alarms

D: It may miss bugs, and have false alarms

**Question 5 - multiple choice, no shuffle**

A tainted flow is

\*A: A flow from an untrusted source to both trusted and untrusted sinks

B: A flow from an untrusted source to a trusted sink

Feedback: This is true, but not completely true. Really, a tainted flow is still tainted even if it ends up at an untrusted sink

C: A flow from a trusted source to an untrusted sink

Feedback: Flows from trusted sources are not tainted

D: A flow from a trusted source to both trusted and untrusted sinks

Feedback: Flows from trusted sources are not tainted

**Question 6 - checkbox, shuffle, partial credit**

(4 points) 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 $$\color{red}{\verb|fmt|}$$ and the argument to $$\color{red}{\verb|printf|}$$ are untainted, while the result of $$\color{red}{\verb|fgets|}$$ 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); }

\*A: flow-sensitive, context-**IN**sensitive

Feedback: 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.

\*B: flow-**IN**sensitive, context-**IN**sensitive

Feedback: 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.

C: flow-sensitive, context-sensitive

Feedback: 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)

D: path-sensitive, context-sensitive

Feedback: 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)

**Question 6 - checkbox, variation 1, shuffle, partial credit**

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 $$\color{red}{\verb|fmt|}$$ and the argument to $$\color{red}{\verb|printf|}$$ are untainted, while the result of $$\color{red}{\verb|fgets|}$$ 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); }

\*A: path-sensitive, context-**IN**sensitive

Feedback: 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.

\*B: flow-**IN**sensitive, context-**IN**sensitive

Feedback: 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.

C: flow-sensitive, context-sensitive

Feedback: 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)

D: flow-**IN**sensitive, context-sensitive

Feedback: 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)

**Question 6 - checkbox, variation 2, shuffle, partial credit**

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 $$\color{red}{\verb|fmt|}$$ and the argument to $$\color{red}{\verb|printf|}$$ are untainted, while the result of $$\color{red}{\verb|fgets|}$$ 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); }

\*A: flow-**IN**sensitive, context-**IN**sensitive

Feedback: 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.

\*B: flow-sensitive, context-**IN**sensitive

Feedback: 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.

C: flow-**IN**sensitive, context-sensitive

Feedback: 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)

D: flow-sensitive, context-sensitive

Feedback: 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)

**Question 7 - checkbox, shuffle, partial credit**

(4 points) Consider the following code, where the referenced $$\color{red}{\verb|chomp|}$$ 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); }

\*A: flow-sensitive, context-sensitive

Feedback: 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.

\*B: flow-sensitive, context-**IN**sensitive

Feedback: 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.

\*C: flow-**IN**sensitive, context-**IN**sensitive

Feedback: 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.

D: path-sensitive, context-**IN**sensitive

**Question 7 - checkbox, variation 1, shuffle, partial credit**

Consider the following code, where the referenced $$\color{red}{\verb|chomp|}$$ 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); }

\*A: flow-sensitive, context-sensitive

Feedback: 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.

\*B: flow-**IN**sensitive, context-sensitive

Feedback: 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.

\*C: flow-**IN**sensitive, context-**IN**sensitive

Feedback: 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.

D: path-sensitive, context-sensitive

**Question 8 - checkbox, shuffle, partial credit**

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

\*A: Implicit flows are rarely detected by tainted flow analyses, because detecting them can increase false alarms

Feedback: This is true, as stated in lecture.

\*B: One can occur when assigning an untainted value to an untainted variable, but conditioned on a tainted value

Feedback: This is basically the example given in the lectures.

C: They only arise in object-oriented languages with dynamic dispatch, since the choice of method to call is implicit

Feedback: The example given in lecture does not use object oriented language features, so they are not a prerequisite for implicit flows.

**Question 8 - checkbox, variation 1, shuffle, partial credit**

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

A: One occurs when assigning a tainted value to an untainted variable

Feedback: This is a direct flow, not an implicit one.

\*B: Implicit flows are rarely detected by tainted flow analyses, because detecting them can increase false alarms

Feedback: This is true, as stated in lecture.

C: They only arise in object-oriented languages with dynamic dispatch, since the choice of method to call is implicit

Feedback: The example given in lecture does not use object oriented language features, so they are not a prerequisite for implicit flows.

**Question 8 - checkbox, variation 2, shuffle, partial credit**

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

A: Implicit flows can be stopped by using cryptography to encrypt the data on the flow

Feedback: This is not true: Cryptography will not block the leak of information due to an observation that depends on a secret value.

\*B: Implicit flows are rarely detected by tainted flow analyses, because detecting them can increase false alarms

Feedback: This is true, as stated in lecture.

\*C: One can occur when assigning an untainted value to an untainted variable, but conditioned on a tainted value

Feedback: This is basically the example given in the lectures.

**Question 9 - multiple choice, shuffle**

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

\*A: As a generalized form of testing, when a symbolic executor finds a bug, we are sure it is not a false alarm

Feedback: Moreover, one can often produce a test case from the alarm that reproduces the bug, making it easier to fix

B: Symbolic executors consider all possible program runs, while static analyses don't

Feedback: In fact, it's more likely to be the opposite.

C: Symbolic executors can consider partial programs (e.g., libraries) while static analyzers cannot

Feedback: Not true -- both can consider partial programs.

D: Symbolic executors are both sound and complete, while static analyzers can only be one or the other

Feedback: Not true -- this really a question of the problem being considered, not the method for analyzing the code to solve the problem. Undecidable is undecidable, whatever the tool used.

**Question 10 - multiple choice, no shuffle**

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

A: Flow-sensitivity

Feedback: Yes, but not just this.

B: Context-sensitivity

Feedback: Yes, but not just this.

C: Path-sensitivity

Feedback: Yes, but not just this.

\*D: All of the above

**Question 11 - multiple choice, shuffle**

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

\*A: Its search strategy is to choose new test cases based on constraints generated by terminating runs

Feedback: As such, a non-terminating program may not produce a terminating test, and thus will never produce constraints to produce the next test.

B: Non-terminating programs will consume too many resources

Feedback: Possibly, but not necessarily. E.g., the program $$\color{red}{\verb|while (1);|}$$ will simply run forever and consume few resources

C: Non-terminating programs require user interaction, which concolic execution does not handle

Feedback: Non-terminating programs do not necessarily require user interaction.

D: Concolic execution takes a breadth-first approach, but non-terminating programs are better suited to a depth-first approach

Feedback: It's actually the opposite: concolic is depth-first, not breadth-first.

**Question 12 - multiple choice, shuffle**

Suppose that $$\color{red}{\verb|x|}$$ and $$\color{red}{\verb|y|}$$ 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"); } } }

A: x > 5 ∧ y > 7 ∧ x < 20

\*B: x > 5 ∧ ¬(y > 7) ∧ x < 20

C: ¬(y > 7) ∧ x < 20

D: x > 5 ∧ ¬(y > 7) ∧ ¬(x < 20)

**Question 13 - multiple choice, no shuffle**

Suppose that $$\color{red}{\verb|x|}$$ 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"); }

A: x > 5 ∧ z > 3

Feedback: Path conditions only mention symbolic variables, and z is not symbolic.

\*B: x > 5

Feedback: Path conditions only mention symbolic variables, and this is the only condition that needs to be satisfied to reach the desired line.

C: ¬(x > 5) ∧ z > 3

Feedback: Path conditions only mention symbolic variables, and z is not symbolic.

D: z > 3

Feedback: Path conditions only mention symbolic variables, and z is not symbolic.

**Question 14 - checkbox, shuffle, partial credit**

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

\*A: Randomly restart the search from the main function

Feedback: This avoids the problem of being stuck in a "local minimum", i.e., a portion of the program that has many paths, at the expense of exploring other parts of the program

\*B: Choose between two paths based on whether one reaches program statements not previously executed

Feedback: This approach intends to maximize "coverage" under the theory that executing all lines of code is more important than executing arbitrary groups of paths in the same code area

C: Switch between concolic and non-concolic execution

Feedback: Concolic and non-concolic execution are two kinds of symbolic execution algorithm, not elements of a single algorithm

**Question 14 - checkbox, variation 1, shuffle, partial credit**

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

A: Switch between path-sensitive and path-insensitive analysis

Feedback: Symbolic executors are essentially always path sensitive

\*B: Randomly restart the search from the main function

Feedback: This avoids the problem of being stuck in a "local minimum", i.e., a portion of the program that has many paths, at the expense of exploring other parts of the program

C: Switch between concolic and non-concolic execution

Feedback: Concolic and non-concolic execution are two kinds of symbolic execution algorithm, not elements of a single algorithm

**Question 14 - checkbox, variation 2, shuffle, partial credit**

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

A: Switch between concolic and non-concolic execution

Feedback: Concolic and non-concolic execution are two kinds of symbolic execution algorithm, not elements of a single algorithm

\*B: Choose between two paths based on whether one reaches program statements not previously executed

Feedback: This approach intends to maximize "coverage" under the theory that executing all lines of code is more important than executing arbitrary groups of paths in the same code area

\*C: Choose between two paths based on a notion of priority

Feedback: Coverage is one kind of priority; another kind might be based on whether some other static analysis tool finds part of the path suspicious