1a) incomplete and sound. (since lots of false positives and no chance of a false negative)

b) yes

I disagree, I think no in this case.

Int kek(int x) {

If(x == 4)

do smth

return 0

Else

do smth

return 1

Return 0.

}

We will always hit 100% branch coverage with some input x = 4 and x not 4 returning in the then or else of the branch, but the final return is unreachable.

The program can return early or crash.

c) no <https://stackoverflow.com/questions/14519416/a-difference-between-statement-and-decision-coverage>

Int test(int x)

If (x == 4)

do smth

Return 0

Exercising some testcase of x = 4 will hit all the statements (100% statement coverage), but we do not necessarily exercise all of the branches (we miss the else of the branch, which returns anyway).

d) 2x and 3x

e) ?

Based on the paper, we mask the query conditions sequentially (I’m gonna guess left->right), if the query without the masked one becomes satisfiable, we add the masked query to the UBset. We have a query in the form of H AND (set of dominators of fragment), so we mask those dominators. Guessing left->right again we can say H is (y != -2^31), and the dominators being (y = 0) and (y != 0), mask these two one at a time and see if the resulting query becomes satisfiable, if it is then the query we masked is relevant:

Masking Y = 0, we have (y ! = -2^31) AND (y != 0), this query is satisfiable, so (y = 0) is added to the minimal UBset. Masking (y != 0), we have (y != -2^31) AND (y = 0), this query is also satisfiable so (y != 0) is added to the UBset. We end up with (y = 0) and (y != 0). I think.

f) ITE(-1<=k<=3, 1, 2) > x+1

g) [Not sure what the J&K compiler is, but general ideas]

(not sure) Because foo() is uninstrumented, J&K compiler doesn’t know how large is the buffer, so it can’t detect q = q – 150 overflows and goes into the buffer for p.

2

This question likely has many valid answers based on different representations. Here is one (lazy) one, which skips some precise specifics as I couldn’t be bothered to type up everything but the idea holds

a) sets of variable parities, e.g {Odd\_a, Even\_b}

b) Forward

c) general transfer function: parity\_gen\_B Union (X \ parity\_kill\_B)

Where parity\_kill\_B is the set containing EVEN\_x and ODD\_x for all variables in B where they appear on the LHS of a statement (assignment)

Parity\_gen\_B is any new parity information for any variables appearing on the LHS of a statement in B (assignment), this differs per statement.

For “x = constant” statements, this is simply parity(x) = parity(constant)

d) For “z = x+y” the parity is: Unknown if either parity(x) or parity(y) are unknown, otherwise it is EVEN if parity(x) == parity(y) else ODD

Unknown is represented by both ODD\_x and EVEN\_x appearing in the set

e) Union, partial order is superset.

F)

1 X=4 IN: {}, OUT: {EVEN\_x}

2 Y=5 IN: {EVEN\_x}, OUT: {EVEN\_x, ODD\_y}

3 Y=x+y IN: {EVEN\_x, ODD\_y}, OUT: {EVEN\_x, ODD\_y}

g) 2 iterations

1 X = 1 IN: {}, OUT: {ODD\_x} |IN: {}, OUT: {ODD\_x}

2 while (X < 5) IN: {ODD\_x}, OUT:{ODD\_x} |IN: {ODD\_x, EVEN\_x}, OUT:{ODD\_x, EVEN\_x}

3 X = X + X IN: {ODD\_x}, OUT:{EVEN\_x} |IN: {ODD\_x, EVEN\_x}, OUT:{ODD\_x, EVEN\_x}