1a)

I) a, b, c are all allocated on the stack. Valgrind runs on the binary and cannot instrument stack allocations so it misses this OOB access

Ii) –1000 is far outside the possible stack range so it’s caught. Valgrind can find invalid accesses that overrun the stack.

Iii) ASan creates redzones around each of the allocated arrays on both ends. So –10 falls within one of these redzones and ASan catches it.

Iv) -10 is OOB and therefore undefined behaviour so the compiler can just optimise the statement out. Alternatively, as b is never read from again and this is in main, b is optimised out.

b)

i) Using offset + (addr >> 3)

I get 0xFFFFFFF800C as the shadow address

Ii) Shadow value = 0 as all 8 bytes are addressable at that point

Iii) Trying to dereference an address in the shadow range would point you into the bad range which is page protected and would throw an error

C)

Taken from ASan paper

Optimisation 1 –

void inc(int \*a) {

(\*a)++;

}

Two memory accesses, one read and one write. Only need to instrument the first access.

Optimisation 2 -

Instrument only a[0] and a[n-1]:

for (int i = 0; i < n; i++)

a[i] = ...;

D)

ASan – Fast. Can capture many different bugs but not as many as Valgrind on its own. Good when you have the source code available. Will only catch if the input triggers it. Does not work with uninstrumented code

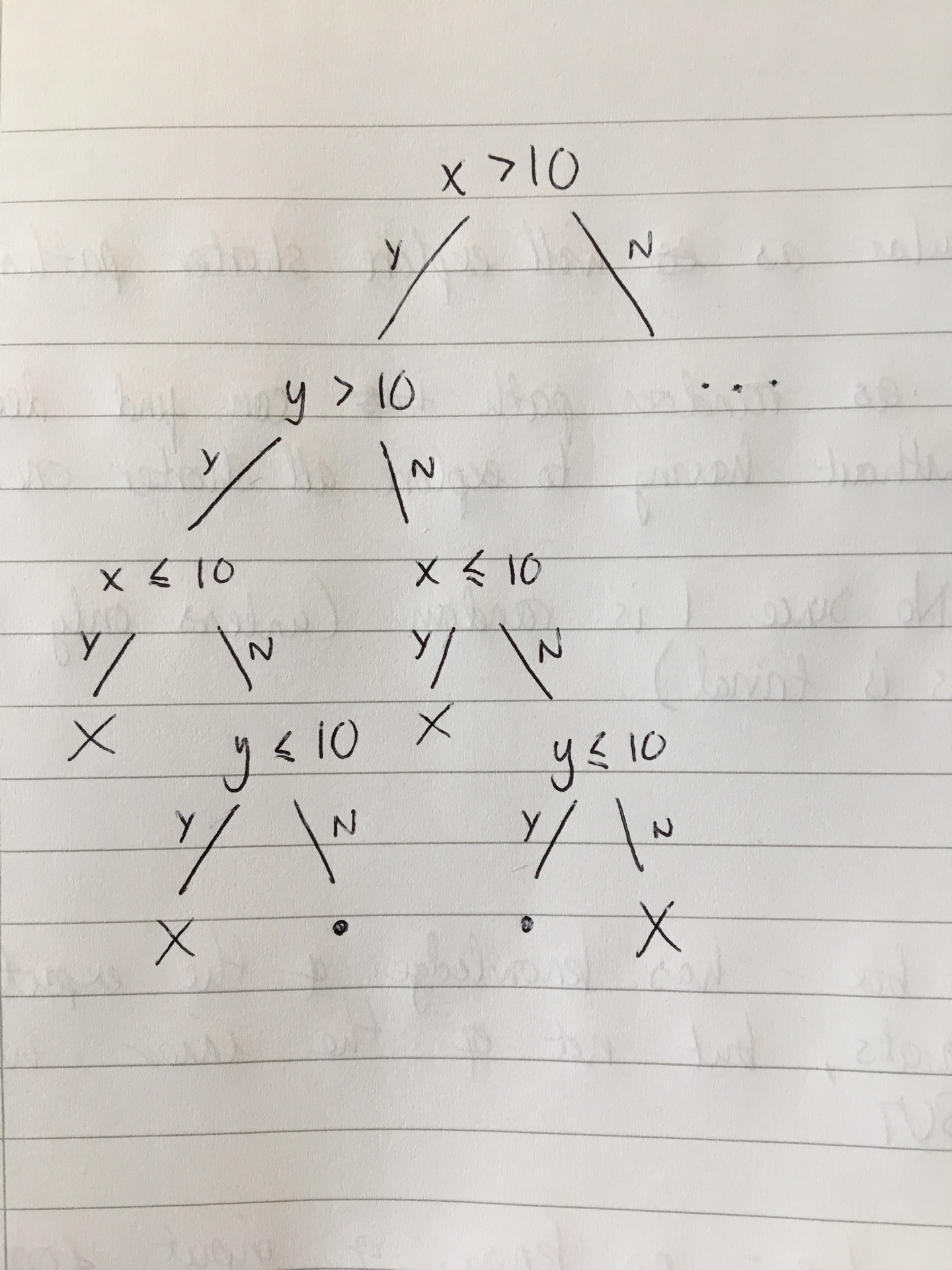
Valgrind – Slower. Doesn’t require source code. Again only works with concrete inputs. Works with uninstrumented code.

KLEE – Significantly more comprehensive but also heavier. Can catch buffer overflows that happen with any input as it is symbolic. Might not reason memory allocation well as they are concrete instead of symbolic (under-approximation, false negative)

2)

a)

I) 4 paths?



Ii) Foo and Bar are identical in what they return for any inputs x and y. So the assertion can be simplified to true by the compiler. This leaves one straight line path.

Iii) Just follow the steps in the DSE lecture starting with foo and then bar.

b) See previous question

c) Random path selection – Favours paths high in the tree. Random selection.

BFS – Not random. Goes across not down.

Similarities: More broad than deep - so explores shorter paths sooner. Good at avoiding starvation in loops?

They could coincide, but this would be entirely down to chance since random path selection is random. (Unless there's only 1 path through the program but this is trivial).