Solutions

ENEE 457
Static Analysis Class Exercise

1. Assume we have an analyzer that takes as input any C program and has the following properties:
   a) The analyzer always terminates
   b) If the C program makes an array out-of-bounds memory access during its run on an input x, then the analyzer outputs 1.
   c) If the C program does not make an array out-of-bounds memory access during its run on an input x, then the analyzer outputs 0.

   Show that the analyzer can be used to solve the Halting Problem.

We need to show a reduction from the halting problem to the array out-of-bounds problem. Specifically, we get as input a C program P along with an input x and we want to know whether it terminates. We want to transform it into a program P' and input x' to feed into the array-out-of-bounds analyzer. If the analyzer tells us that P' makes an out-of-bounds access on input x', we would like to conclude that the original program P halts on input x. If the analyzer tells us that P' does not make an out-of-bounds access on input x', we would like to conclude that the original program P does not halt on input x.

At a high level, to perform this transformation, we look at the code of P. Every time there is an "exit" instruction (i.e. an instruction that causes the execution to terminate), we purposely insert an array-out-of-bounds access immediately before it. This transforms the program P into the program P'. The input x can stay the same.

The above solution is not complete, because it is possible that program P also makes array-out-of-bounds accesses in other places. Therefore, if our analyzer tells us that an array-out-of-bounds occurs, we do not know if this implies that program P halts or that program P makes an array-out-of-bounds access elsewhere. However, the above gives the high-level intuition of how to argue that static analyzers that always terminate and are always correct imply a solution to the halting problem.

ENEE 457
Static Analysis Class Exercise

2. Consider the following code snippet on which we would like to perform a taint analysis. Type qualifiers are represented by capital letters: A, B, C, D, E.

```
1   int printf(A char *fmt, ..);
2   B char *fgets(..);
3
4
5   int main () {
6           C char *mystring = fgets(.., network_fd);
7           D char *mystring2 = mystring;
8           E char *mystring3 = ‘‘Hello World’’;
9           mystring2 = mystring3;
10          printf(mystring2);
11          return 0;
12  }
```

i. Identify all the sources and sinks in the code snippet and determine the corresponding settings for the type qualifiers.

ln 1 (sink) A = untainted
ln 2 (source) B = tainted

ii. List all of the constraints on the type qualifiers.

ln 6 C >= tainted
ln 7 D >= C
ln 8 E >= untainted
ln 9 D >= E
ln 10 D < = untainted

iii. Is there a vulnerability in the above code? Is there a solution for the undetermined type qualifiers that satisfies all the constraints? If there is no vulnerability and no solution, it means that our taint analysis has produced a false positive. How can the taint analysis be modified so that the false positive is removed?

No vulnerability.
No solution since constraints from ln 6, ln 7, ln 10 imply:
tainted <= C <= D <= untainted
This implies that tainted <= untainted, which is false, since we assume tainted > untainted
When mystring2 is assigned in ln 9, it should be given a new name (each variable should only be assigned once). ln 9 becomes:
F char *mystring4 = mystring3
ln 10 becomes: printf(mystring4)
Now all constraints can be satisfied.