Assignment 2: Semantically Safe
15-316 Software Foundations of Security and Privacy

Due: **11:59pm**, Sunday 9/29/18
Total Points: 50

1. **Safe superdiversity (10 points).** The so-called "monoculture problem" of software security refers to the fact that when a large set of users runs bytecode-identical versions of the same application, then any vulnerability affecting that application will have dramatic impact as it applies to the whole population. In response to this, engineers looked for ways to diversify software by distributing different versions of the same application, each with a unique bytecode representation.

   One example of this is a technique called "superdiversification", proposed by researchers from Nokia and Microsoft in 2008. The technique is applied by a compiler when generating an executable, and performs a brute-force search of all short instruction sequences to look for semantically-equivalent machine code implementations for desired functions.

   It's probably not wise to dopt this crazy-sounding technique without convincing evidence that the implementations it produces are in fact equivalent to the original code you wrote. If this were not the case, your application might end up with arbitrary behaviors, potentially leading to even more vulnerabilities than would otherwise be present. Luckily, you are familiar with dynamic logic, and are able to rigorously prove that such implementations are correct.

   Prove that the following code negates y and stores it in x. In other words, write a specification by giving a precondition A and postcondition B that captures this functionality, and then use the axioms of dynamic logic to show that the following formula is valid.

   $$x := x - y; y := y + x; x := x - y$$

   Solution.

2. **Once and for all (10 pionts).** Having completed the previous exercise, there is an obvious problem when it comes to using this approach in practice. Even if we have a tool to do the proofs for us, it's going to be too much work to write down a brand new specification for the safety of each and every transformation that the system decides to use. Your goal in this problem is to write a single specification, as a dynamic logic formula, that covers all of our concerns for superdiversity.

   - You should assume that the original code fragment is represented by α, and the super-diversified replacement by β.
   - You may assume that the only variables that both α and β use are x and y.
   - Your specification should capture the fact that α and β must be equivalent in terms of their input-output behavior with respect to x and y. In other words, if they are executed in states that agree on the values of x and y, then when they terminate, they will agree on (potentially new) values of x and y.
   - If it helps you answer the question, you may assume that α and β always terminate. If your answer requires this assumption, then you must state it and explain why.
   - Likewise, if it helps, you may assume that α and β are programs in the language discussed in lecture. If your answer *does not* use this assumption, then be sure to say so and explain!

   In addition to providing a dynamic logic formula that meets these requirements, determine whether this formula corresponds to a safety property. If so, explain why by identifying the finite prefixes according to Definition 11 in the "Semantics, Safety, & Dynamic Logic" lecture notes. If not, provide a concise explanation why.

   Solution.

3. **Nondeterministically Satisfied (20 points).**

Now that we know how to reason about programs using logic, perhaps there are interesting ways to use logic directly in a program to make life easier. One example would be the "assign such that" statement, which would let us update a variable to take some nondeterministically-chosen value *such that* it satisfies a given condition. An example of this might look like the following, where x is assigned some arbitrary value between 0 and 15,316:

$$x \texttt{ :=? } 0 \leqslant x \wedge x \leqslant 15316$$

Aside from being useful for writing provably-correct code, this construct could maybe even help us generate good random passwords that satisfy those annoying character class requirements...

(a) Define the a formal semantics for this command. That is, define the following set of traces assuming that $p(x)$ is a formula with a free occurrence of the variable x:

$$[\![x \texttt{ :=? } p(x)]\!] = \{(\omega, \nu) : \ldots\}$$

(b) Then, give an axiom that enables compositional reasoning about programs that make use of the command.

$$[x \texttt{ :=? } p(x)]q(x) \leftrightarrow \ldots$$

The right-hand side that you fill in for this axiom should contain no box or diamond modalities.

(c) Finally, be sure to relate the axiom to your semantics by proving that it is sound.

4. **On second thought... (10 points).** Sometimes adding new features to a language can be more trouble than they're worth. In the lecture notes, we somewhat casually concluded that the following two contracts for a given program were equivalent.

$$[\alpha]A \land [\alpha]B$$
$$[\alpha](A \land B)$$

For the language we've discussed in lecture the same holds for disjunction, so:

$$\models [\alpha]A \lor [\alpha]B \leftrightarrow [\alpha](A \lor B)$$

Now that α might contain an "assign such-that" command, is this still the case?

- If so, then prove that $\models [x \texttt{ :=? } p(x)](A \lor B) \leftrightarrow [x \texttt{ :=? } p(x)]A \lor [x \texttt{ :=? } p(x)]B$.
- If not, then give an example of a program α that makes use of "assign such-that", and a postcondition $A \lor B$ such that $[\alpha](A \lor B)$ is not equivalent to $[\alpha]A \lor [\alpha]B$.