Assignment 4: The Highs and Lows of Information Flow
15-316 Software Foundations of Security and Privacy

Total Points: 50

1. **Flow through abort (15 points).**

   The definition of non-interference described in lecture (Eq. 1) does not account for aborted executions.

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2 \wedge \langle \omega_1, \alpha \rangle \Downarrow \omega_1' \wedge \langle \omega_2, \alpha \rangle \Downarrow \omega_2' \rightarrow \omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2' \tag{1}$$

   In particular, consider the following program under the policy $\Gamma = (x : \mathtt{H})$:

```
assert(x > 0)
```

   If our threat model allows an attacker to detect whether a trace of this program aborts, then the attacker can learn information about the value of x by observing whether the final state is Λ or not.

   Part 1 (10 points). First, provide a big-step semantics for the $\mathtt{assert}(Q)$ command; your semantics should match the trace semantics for `assert` given in prior lectures, in the sense that:

$$\langle \omega, \mathtt{assert}(Q) \rangle \Downarrow \nu \text{ if and only if } (\omega, \nu) \in [\![\mathtt{assert}(Q)]\!]$$

   Then, explain how to modify the definition of $\approx_{\Gamma,\mathtt{L}}$ and Equation (1) to arrive at "failure-sensitive non-interference", which characterizes programs that do not leak information about `H` variables either through the `L` variables in initial and final states, or whether the final state is Λ or not.

Part 2 (10 points). Design a typing rule for `assert`(Q) commands, and prove its soundness. In other words, prove that if $\Gamma \vdash$ `assert`(Q), then `assert`(Q) satisfies your definition of failure-sensitive non-interference under Γ. Then, discuss whether any of the typing rules discussed in lecture and the notes need to be changed to enforce failure-sensitive non-interference, and how the soundness argument for the entire system (including your new rule) would need to be changed.

2. **Leveraging interference (10 points).**

   Consider the following program, under the type context $\Gamma = (a : \mathtt{H}, b : \mathtt{L}, c : \mathtt{L})$.

```
if(a < 0) {
  if(b < a) c := 0
  else c := 1
} else {
  if(a < b) c := 0
  else c := 1
}
```

   Describe a procedure that leverages the fact that this program does not satisfy non-interference under Γ to learn the value of the H-typed variable. You can make use of the following assumptions.

   - Assume that an attacker can control the values of L-typed variables prior to executing the program, and observe their value afterwards. They can neither control nor observe H variables at any point.
   - $-N \leqslant a \leqslant N$ for some constant N. State whether your procedure requires that the attacker know N in order to run it.
   - Finally, the attacker can run the program with different L inputs any number of times, and the H input will remain the same.

   How many times does the attacker need to run the program using your procedure to learn a?

3. **Flow types (20 points).**

Consider the following program.

```
if(a = b) {
  d := c × e;
  e := 0
} else {
  c := 0;
  d := d + 1
}
b := c × e
```

Part 1 (10 points). List the information flow constraints required for this program to typecheck under a policy Γ, assuming that $\mathtt{H} \sqsubseteq \Gamma(a)$ (i.e., this must be one of your constraints). For example, if we were to list out the constraints for the program $\mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}$, they would be:

$$\Gamma(y) \sqsubseteq \Gamma(x), \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(x), \Gamma(x) \sqsubseteq \Gamma(z), \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(z)$$

Note that to make the notation less burdensome, it would be fine and perfectly understandable to write the following instead:

$$y \sqsubseteq x, \mathtt{pc} \sqsubseteq x, x \sqsubseteq z, \mathtt{pc} \sqsubseteq z$$

In other words, y must flow to x, x must flow to z, and $\mathtt{pc}$ must flow to both x and z. These constraints follow because the rule for typechecking $\mathtt{x} := \mathtt{y}$ requires that $\Gamma(y) \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(x)$, and the rule for type-checking $\mathtt{z} := \mathtt{x}$ requires that $\Gamma(x) \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(z)$.

Then, identify a *minimal* policy Γ^* under which this program α typechecks, and which satisfies the constraints you provided above. Here, minimal means that for any policy Γ that satisfies the constraints where $\Gamma \vdash \alpha$, and any variable x, $\Gamma^*(x) \sqsubseteq \Gamma(x)$.

Part 2 (10 points). The policy that you wrote for Part 1, being a minimal policy that satisfies the constraints implied by $\mathtt{H} \sqsubseteq \Gamma(a)$, is the *most permissive* policy which ensures that the contenta of a remain confidential under enforcement by typechecking. However, because the type system discussed in lecture is sound but not complete, it may err on the side of requiring that more variables be typed `H` than are truly necessary for noninterference to hold.

Show that the minimal policy you provided in Part 1 is conservative by identifying at least one variable which is typed `H`, but cannot be influenced by a. Then, use self-composition to construct a dynamic logic formula whose validity implies that this program satisfies noninterference under a policy Γ', which is identical to your Γ^* except that the identified variable is labeled `L` instead of `H`. Your formula may use α to refer to the program listed at the beginning of this problem, and α' to refer to its "primed" version where each variable x is replaced with x'.

Extra credit (5 points). Rewrite the program given at the beginning of this problem so that it can be typechecked by a policy Γ that assigns $\Gamma(a) = \texttt{H}$ and $\Gamma(\cdot) = \texttt{L}$, where $\cdot$ is the variable that you identified in Part 2. The re-written program should be semantically equivalent to the original. For credit, you must use the typing rules to show that $\Gamma \vdash \alpha'$, where α' is your re-written solution.