Assignment 5: The Highs and Lows of Information Flow
15-316 Software Foundations of Security and Privacy

1. **Flow through abort (25 points).**

   In lecture, we defined non-interference in terms of a language that contains assignment, composition, conditional statements, and while loops.

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

   This definition depends on the relation $\approx_{\mathtt{L}}$, which says that two states are "low equivalent" whenever their low-variables are the same.

   $$\omega_1 \approx_{\mathtt{L}} \omega_2 \text{ if and only if } \forall x. \Gamma(x) = \mathtt{L} \rightarrow \omega_1(x) = \omega_2(x) \tag{2}$$

   This question will develop an extention to this notion of noninterference that accounts for $\mathtt{assert}(P)$ commands.

   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.

   $$\frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{true}}{\langle \omega, \mathtt{assert}(P) \rangle \Downarrow_{\mathbb{B}} \omega} \qquad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{false}}{\langle \omega, \mathtt{assert}(P) \rangle \Downarrow_{\mathbb{B}} \Lambda}$$

   Part 1 (5 points). Show how the following program leaks information labeled `H` to an observer who can see whether the final state is Λ, as well as the initial and final values of `L` variables. You should assume that the policy is $\Gamma(x) = \mathtt{H}, \Gamma(y) = \mathtt{L}$.

   $$\mathtt{if}(y \neq 0)\,\{x := 2\}\,\mathtt{else}\,\{\mathtt{assert}(x = 2)\}$$

   Your solution should provide two initial `L`-equivalent states, and explain how the observer learns information about the `H` variables of the initial states from their observations.

Part 2 (10 points). Modify Equations 1 and 2 above to arrive at a formal definition of "abort-sensitive non-interference", which characterizes programs that do not leak information about `H` variables through the `L` variables in final states, or through the program's termination status (i.e., whether the final state is Λ).

Note: depending on your solution, you may only need to modify one of Eqs 1 and 2. If so, just state that the other equation is unchanged.

Part 3 (10 points). Design a typing rule for `assert`(Q) commands, and explain why it is sound with respect to your answer to Part 2. You do not need to provide a proof, but if you wish to, then be sure to first define the big-step semantics of `assert`.

2. **Dynamic pitfalls (15 points).** While the static type system studied in class may reject some programs that satisfy noninterference, it is sound: it will never accept a program that violates the policy. An often-raised proposal for mitigating some of the "false" rejections, i.e., cases where the type system unnecessarily rejects a program, is to track information flow *dynamically* at runtime while still preserving soundness.

   One such approach resembles taint analysis. The runtime monitor keeps track of the security label of each variable, raising and lowering variables' labels depending on what is assigned to them as the program executes. It also tracks the label of the program counter, to make sure that implicit flows can be prevented. When an assignment occurs inside of a conditional or loop, the label of the target variable is set to the least upper-bound of the program counter's label and the label of the assignments right hand side. When the program finishes executing, we assume that an attacker can see the values only of variables that are tracked with label `L` when the program terminates.

   So for example, under the initial policy $\Gamma = [x : \texttt{L}, y : \texttt{H}]$, the following program would terminate with both x and y marked as label `L` because y is overwritten by a constant:

   $$y := 0; x := y$$

   Accordingly, both of their final values would be observable by an attacker, because the monitor labels them as `L` when the program terminates.

   Show why this enforcement mechanism is unsound by giving an example of a program that violates the policy $\Gamma = [x : \texttt{L}, y : \texttt{H}]$ by leaking information about y when it is run under this mechanism.