**Assignment 4: 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 $\texttt{assert}(P)$ 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 $\Lambda$ or not.

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

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

   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 Equation 1 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 $\Lambda$).

**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.

   **Part 1 (10 points).** One way to accomplish this is to build an enforcement mechanism that terminates the program only when high-security variables are about to be assigned to low-security ones. The mechanism also tracks the label of the program counter, so that it can terminate before an implicit flow is made as well. So, for example, consider the following program under policy $\Gamma = [x : \mathtt{L}, y : \mathtt{H}]$:

   $$\mathtt{if}(0 = 0)\, x := 1\, \mathtt{else}\, x := y$$

   The typesystem will reject this program, but the dynamic mechanism will always let it run.

   Is this mechanism sound with respect to an attacker who can observe whether the program is terminated? If so, explain your reasoning. If not, give an example of a program that will leak information when this enforcement mechanism is used.

**Part 2 (10 points).** Another 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 policy $\Gamma = [x : \texttt{L}, y : \texttt{H}]$, the following program would terminate with both $x$ and $y$ marked as label `L`, and both of their values would be observable by an attacker:

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

Is this mechanism sound for an attacker who can observe the final values of `L` variables as described above? If so, explain your reasoning. If not, give an example of a program that will leak information when this enforcement mechanism is used.