Chapter 6

Information Flow Control: Beyond TINI

A λ-observer might well have a legitimate need or just have the capability to access the variables in $V_{\sqsubseteq\lambda}$ before, after, and during execution of a program S.

- Variables in $V_{\sqsubseteq\lambda}$ might be used to model the channels that the λ-observer is using while S executes.
- Malware concurrently executing on the same computer as S might be able to monitor the state as S executes. If the program that was co-opted by the malware has label λ then the malware is a λ-observer.

Therefore, we now consider noninterference policies for settings where the Batch and Asynchronous assumptions (page 208) adopted for TINI are replaced by:

> *Interactive.* For a program S with variables V, a λ-observer can read variables in $V_{\sqsubseteq\lambda}$ initially, during execution of S, and after terminating executions of S.
>
> *Termination Detection.* A λ-observer can detect the termination of a program S.

Due to Termination Detection, a λ-observer seeing no change to the variables in $V_{\sqsubseteq\lambda}$ can know whether the apparent lack of activity is because the execution of S has terminated or it is because the execution is continuing but all of the changes are to variables from $V_{\not\sqsubseteq\lambda}$ and, thus, are not visible to the λ-observer.

6.1 Progress Sensitive Noninterference

Progress sensitive noninterference (PSNI) specifies that the initial values of variables in $V_{\not\sqsubseteq\lambda}$ do not affect whether S terminates and do not affect the values of variables $V_{\sqsubseteq\lambda}$ in any of the intermediate states produced during a terminating

$$\sigma\colon \quad \begin{bmatrix} x \mapsto 0 \\ y \mapsto ? \\ z \mapsto ? \\ pc \mapsto \ell_1 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto ? \\ pc \mapsto \ell_2 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_3 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_4 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 0 \\ pc \mapsto \ell_5 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 2 \\ pc \mapsto \ell_9 \end{bmatrix}$$

$$\sigma'\colon \quad \begin{bmatrix} x \mapsto 1 \\ y \mapsto ? \\ z \mapsto ? \\ pc \mapsto \ell_1 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto ? \\ pc \mapsto \ell_2 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_3 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_6 \end{bmatrix} \begin{bmatrix} x \mapsto 4 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_7 \end{bmatrix} \begin{bmatrix} x \mapsto 4 \\ y \mapsto 0 \\ z \mapsto 2 \\ pc \mapsto \ell_8 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 2 \\ pc \mapsto \ell_9 \end{bmatrix}$$

Figure 6.1: Sequences σ and σ' of intermediate states for executions of (6.1)

or a non-terminating execution of S. So PSNI specifies what should not be revealed about the initial values of variables in $V_{\not\sqsubseteq\lambda}$ to a λ-observer that can detect if S has terminated and that can monitor the values of variables $V_{\sqsubseteq\lambda}$ during an execution.[1]

The following program illustrates how information about the initial values of variables from $V_{\not\sqsubseteq\lambda}$ might be revealed through the sequence of values assigned to variables in $V_{\sqsubseteq\lambda}$.

$$\begin{array}{l} \ell_1\colon y := 0;\ \ell_2\colon z := 0; \\ \ell_3\colon \texttt{if}\ x = 0\ \texttt{then}\ \ell_4\colon y := 1;\ \ell_5\colon z := 2 \\ \qquad\qquad\ \texttt{else}\ \ell_6\colon x := 4;\ \ell_7\colon z := 2;\ \ell_8\colon y := 1 \\ \quad \texttt{fi}; \\ \ell_9\colon \end{array} \tag{6.1}$$

The initial value of x determines the order of updates to y and z in the `if` statement. So if $\Gamma(x) = \mathsf{H}$, $\Gamma(y) = \mathsf{L}$, and $\Gamma(z) = \mathsf{L}$ hold, then the sequence of values assigned to variables $y, z \in V_{\sqsubseteq\mathsf{L}}$ in the intermediate states of an execution reveals information about the initial value of variable $x \in V_{\not\sqsubseteq\mathsf{L}}$. The final values of y and z are the same for all initial values of x, though, so program (6.1) does satisfy TINI.

For formally defining PSNI, we introduce predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ that holds if and only if executions of S from initial states that differ only in the values of variables in $\mathcal{V}$ produce indistinguishable sequences of updates to the variables in $\mathcal{W}$. As an example, $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold if S is program (6.1), because different initial values for x can cause executions where the sequences of values assigned to variables y and z during one execution can differ from the sequences

[1] *Progress insensitive noninterference* (PINI) has also been studied. It drops the Termination Detection assumption. PINI thus requires that the initial values of variables in $V_{\not\sqsubseteq\lambda}$ not affect the values of variables $V_{\sqsubseteq\lambda}$ in any prefix of the sequence of intermediate states produced by executing S. PINI, however, does allow the initial values of variables in $V_{\not\sqsubseteq\lambda}$ to affect when or whether S terminates.

assigned during another. Figure 6.1 illustrates by depicting two *execution traces*, each specifying an initial state (including program counter *pc*), followed by the intermediate states produced by the execution of program (6.1), where "?" indicates a value that is unknown or uninitialized. For $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ to hold, the sequences of updates to the values for y and z in σ and in σ' is required to be indistinguishable. We check this requirement by constructing $\sigma|_{\{y,z\}}$ and $\sigma'|_{\{y,z\}}$, where *projected execution trace* $\sigma|_{\mathcal{V}}$ is derived from an execution trace σ by replacing each state s in σ with state projection $s|_{\mathcal{V}}$ and then eliminating consecutive, identical states. The resulting projected execution traces are:

$$\sigma|_{\{y,z\}}\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{6.2}$$

$$\sigma'|_{\{y,z\}}\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 2 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{6.3}$$

The fourth states of (6.2) and (6.3) are different, so $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold—information about the initial value of x would be revealed to an observer monitoring y and z.

To give a formal definition for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$, we generalize predicate $s =_{\mathcal{V}} s'$ on states s and s' to handle execution traces σ and σ':

$$\sigma =_{\mathcal{V}} \sigma'\colon \quad \sigma|_{\mathcal{V}} = \sigma'|_{\mathcal{V}}$$

So $\sigma =_{\mathcal{V}} \sigma'$ holds if and only if projected execution traces $\sigma|_{\mathcal{V}}$ and $\sigma'|_{\mathcal{V}}$ are the same length and have identical i^{th} states, for each i.

Only because we collapse runs of identical states when forming a projected execution trace $\sigma|_{\mathcal{V}}$, do we obtain a sequence of states that models what an observer would see by monitoring the variables in $\mathcal{V}$, since state transitions are detectable by such an observer only if there is a change to the value of some variable being monitored. Notice, a projected execution trace $\sigma|_{\mathcal{V}}$ might have finite length even though σ has infinite length—this case arises with a nonterminating loop in which all assignment statements in the loop body update variables that are not visibile to the observer.

The formal definition for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ employs a function $[\![S]\!]^{\mathsf{tr}}(s)$ that, for a deterministic program S, maps an initial state s to an execution trace that begins with s and is followed by the (possibly infinite length) sequence of intermediate states that would be produced by executing S. For formally defining $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$, we are concerned with comparing pairs of projected execution traces that start in initial states s and s' satisfying $s =_{\overline{\mathcal{V}}} s'$.

$$\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}\colon \quad (\forall s, s' \in \mathit{Init}_S\colon \; s =_{\overline{\mathcal{V}}} s' \;\Rightarrow\; [\![S]\!]^{\mathsf{tr}}(s) =_{\mathcal{W}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{6.4}$$

So $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ holds if different initial values for variables in $\mathcal{V}$ result in indistinguishable projected execution sequences for variables in $\mathcal{W}$.

We illustrate the use of definition (6.4), by checking whether $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ holds when S is program (6.1). $\overline{\mathcal{V}}$ is $\{y, z\}$ (since $\mathcal{V}$ is $\{x\}$), and $\mathcal{W}$ is $\{y, z\}$, resulting in:

$$(\forall s, s' \in \mathit{Init}_S\colon\ s =_{\{y,z\}} s' \ \Rightarrow\ [\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{6.5}$$

The earlier claim that $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold would be confirmed by showing that (6.5) does not hold. The initial states of σ and σ' in Figure 6.1 do satisfy antecedent $s =_{\{y,z\}} s'$, but consequent $[\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')$ of (6.5) does not hold, since (6.2) and (6.3) are different. As expected, we have shown formally that $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold.

We now have the building blocks needed for specifying that the values of variables $V_{\not\sqsubseteq\lambda}$ in initial states are not allowed to affect the termination of a program S or affect the values of variables that a λ-observer can monitor in intermediate states.

> **Progress Sensitive Noninterference (PSNI).** For a deterministic program S where the variables have labels from a set Λ with a partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ps}} V_{\sqsubseteq\lambda})$$ □

6.1.1 PSNI Enforcement

Non-termination of a `while` statement can cause an implicit flow that violates PSNI. As an example, whether `while` statement ℓ_W terminates in IMP program

```
S: x := 0;
   ℓ_if: if z = 0 then skip
                  else ℓ_W: while y ≠ 0 do y := y + 1 end
         fi
    ℓ:   x := 23
```
(6.6)

depends on which of $y > 0$ or $y \leq 0$ holds initially. Moreover, because final assignment statement ℓ is neither in the body of `while` statement ℓ_W nor in the body of `if` statement ℓ_{if}, guards $z = 0$ and $y \neq 0$ are not in $\Theta_S(\ell)$. These guards nevertheless can affect whether assignment statement ℓ is reached: for ℓ to be reached, $z = 0$ must hold initially or $y = 0$ must hold eventually so that `while` statement ℓ_W terminates. Thus, there is an implicit flow from variables z and y in those guards to target x of assignment statement ℓ. If $\Gamma(z) \not\sqsubseteq \Gamma(x)$ or $\Gamma(y) \not\sqsubseteq \Gamma(x)$ holds then this implicit flow also is an illicit flow—even though ℓ complies with Θ_S-Safe Assignment Statements condition (5.24). We conclude that restrictions beyond those required for enforcing TINI are needed for enforcing PSNI.

Generalizing, there will be an implicit flow to variables affected by any statement that could be reached after a `while` statement terminates. This implicit flow conveys information about the values of the variables in the `while` statement guard as well as the variables in the other guards that affect whether that `while` statement is reached. To characterize such an implicit flow, we define a set $\Delta_S(\ell)$ containing those guards that affect whether statement ℓ can be reached in some execution of S. So the elements of $\Delta_S(\ell)$ come from (i) control-flow statements in S having a body that contains ℓ (i.e., $\Theta_S(\ell)$) and (ii) the subset $\mathcal{W}_\ell$ of the `while` statements in S that, by not terminating, would prevent ℓ from being reached, where G_W is the guard on a `while` statement ℓ_W:

$$\Delta_S(\ell): \quad \Theta_S(\ell) \ \cup \ \bigcup_{\ell_W \in \mathcal{W}_\ell} (\{G_W\} \cup \Delta_S(\ell_W)) \tag{6.7}$$

Beware that for a statement ℓ in the body of a `while` statement, subset $\mathcal{W}_\ell$ used in (6.7) might include `while` statements appearing as alternatives to ℓ or appearing after ℓ in the program text. We see this for statement ℓ: S_1 in the program fragment:

```
while G1 do if G2 then ℓ: S1
                  else while G3 do S2 end
           fi;
           while G4 do S3 end
      end
```

Certain choices for guards G_i and statements S_i in this program fragment could result in G_3 (a guard appearing in an alternative to ℓ) and G_4 (a guard appearing after ℓ) being members of $\Delta_S(\ell)$. Both cases arise if ℓ can be executed after the first iteration of outer-most `while` statement.

We can avoid the illicit flows that guards affecting `while` statements cause if we replace $\Theta_S(\ell)$ with $\Delta_S(\ell)$ in Θ_S-Safe Assignment Statements condition (5.24).

Δ_S-Safe Assignment Statements. Ensure that

$$\left(\Gamma_{\mathcal{E}}(expr) \ \sqcup \ \left(\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G)\right)\right) \ \sqsubseteq \ \Gamma(w) \tag{6.8}$$

holds for each assignment statement ℓ: $w := expr$ that S executes. □

Δ_S-Safe Assignment Statements condition (6.8) is potentially more restrictive than Θ_S-Safe Assignment Statements condition (5.24), since $\Theta_S(\ell) \subseteq \Delta_S(\ell)$ holds by definition. The additional guards in $\Delta_S(\ell)$ are those that affect whether ℓ cannot be reached due to non-termination.

Calculation of $\Delta_S(\ell)$ is undecidable because $\Delta_S(\ell)$ is defined in terms of statement reachability—specifically, which `while` statements can prevent ℓ from being reached. However, the calculation of $\Delta_S(\ell)$ is straightforward for programs S that comply with some easily-checked restrictions.

Guard Restrictions for PSNI. If for every `while` statement ℓ_W with guard G_W in a program S

(i) $\Gamma_{\mathcal{E}}(G_W) = \bot_\Lambda$ holds, and

(ii) $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ holds for every guard $G \in \Theta_S(\ell_W)$.

then for every statement ℓ in S:[2] $\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$ □

So in programs satisfying restrictions (i) and (ii), Δ_S-Safe Assignment Statements condition (6.8) can be discharged by checking Θ_S-Safe Assignment Statements condition (5.24), which can be checked with a static analysis of S.

Extending IMP to Relax Guard Restrictions. Guard Restrictions for PSNI requires that iteration be controlled by variables with label $\bot_\Lambda$. Some relaxation of the restrictions are possible, though, for looping that is guaranteed to terminate. Only loops that sometimes terminate cause implicit flows.

A static analysis cannot determine whether a `while` statement will always terminate, due to the undecidability of the halting problem. However, we can avoid the need for such an analysis for loops that the programming language ensures will always terminate. The `for`-loop statement

$$\texttt{for } v := \mathit{expr} \textbf{ to } \mathit{expr}' \texttt{ do } T \texttt{ end} \tag{6.10}$$

is such a statement, provided (i) v is an integer variable and considered an assignment statement target; (ii) *expr* and *expr′* are integer-valued expressions; and (iii) body T does not contain any assignment statements or `for`-loop statements with v as a target or with any variable in $\mathit{Vars}(\mathit{expr}')$ as a target. In executions of `for`-loop statement (6.10), body T is executed between zero and some bounded number of times.

The guard for (6.10) is defined to be $\mathit{expr} \le v \le \mathit{expr}'$ since this is the predicate that holds whenever body T starts executing and iteration terminates when

[2] Here is that derivation. From definition (6.7) for $\Delta_S(\ell)$ we get:

$$\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \sqcup \bigsqcup_{\substack{\ell_W \in \mathcal{W}_\ell \\ G \in (\Delta_S(\ell_W) \cup \{G_W\})}} \Gamma_{\mathcal{E}}(G) \tag{6.9}$$

Restrictions (i) and (ii) in Guard Restrictions for PSNI imply that for every subset $\mathcal{W}$ of the `while` statements in a program S we have:

$$\bigsqcup_{\substack{\ell_W \in \mathcal{W} \\ G \in (\Delta_S(\ell_W) \cup \{G_W\})}} \Gamma_{\mathcal{E}}(G) = \bot_\Lambda$$

So substituting into (6.9) (with $\mathcal{W}$ instantiated by $\mathcal{W}_\ell$) we get

$$\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \sqcup \bot_\Lambda$$

this predicate becomes *false*. So for every statement ℓ in body T of a `for`-loop statement appearing in a program S

$$expr \le v \le expr' \in \Theta_S(\ell)$$

will hold. Moreover, if a program contains `for`-loop statements but no `while` statements then $\mathcal{W}_\ell = \varnothing$ holds for every statement ℓ. So $\Delta_S(\ell)$ simplifies to $\Theta_S(\ell)$, and Δ_S-Safe Assignment Statements simplifies to Θ_S-Safe Assignment Statements. Thus, Δ_S-Safe Assignment Statements can be discharged by performing the static analysis that discharges Θ_S-Safe Assignment Statements.

6.1.2 Dynamic Enforcement of PSNI

By monitoring state changes as execution proceeds, a λ-observer might be able to detect that a dynamic enforcement mechanism has blocked an execution. To account for this possibility, we add

> *Blocking Detection.* A λ-observer can detect when execution of a program becomes blocked by a dynamic enforcement mechanism.

to the Interactive assumption and the Termination Detection assumption (see page 227) that we have been using to characterize the capabilities of λ-observers.

Detecting that an execution has become blocked can result in an illicit implicit flow. This is illustrated in the following IMP program, which implements the assignment statement $x_\mathsf{L} := x_\mathsf{H}$ if $0 \le x_\mathsf{H} \le N$, $\Gamma(i_\mathsf{L}) = \mathsf{L}$, $\Gamma(x_\mathsf{H}) = \mathsf{H}$, and $\mathsf{H} \not\in \mathsf{L}$ hold.

```
S:  i_L := 0;
    while i_L ≤ N do
       if x_H = i_L then ℓ: x_L := i_L  else skip fi
       i_L := i_L + 1
       end
```
(6.11)

Assignment statement ℓ to x_L does not comply with Δ_S-Safe Assignment Statements condition (6.8), because ℓ appears in the body of an `if` statement with guard $x_\mathsf{H} = i_\mathsf{L}$ where $\Gamma_\mathcal{E}(x_\mathsf{H} = i_\mathsf{L}) \not\in \Gamma(x_\mathsf{L})$ holds. Moreover, a dynamic enforcement mechanism that blocks execution when assignment statement ℓ is reached creates an illicit implicit flow—the last value of i_L that the L-observer reads before detecting that execution became blocked is the initial value of x_H.

We conclude that a dynamic enforcement mechanism for PSNI must prevent illicit *detection flows*, wherein detecting that an execution became blocked allows a λ-observer to learn information about the initial value of a variable in $V_{\not\sqsubseteq\lambda}$ from having monitored the values of variables in $V_{\sqsubseteq\lambda}$ during the execution. An illicit detection flow occurs whenever (i) a λ-observer is able to determine that execution became blocked upon reaching some statement ℓ, and (ii) $\Gamma(v) \not\in \lambda$ holds for some variable $v \in Vars(G)$ in a guard $G \in \Delta_S(\ell)$ that was evaluated prior to execution becoming blocked. For example, program (6.11) exhibits an illicit detection flow because reaching assignment statement ℓ is the only

reason that execution could become blocked, guard $x_{\mathsf{H}} = i_{\mathsf{L}}$ is evaluated before execution becomes blocked, and $\Gamma(x_{\mathsf{H}}) \not\sqsubseteq \mathsf{L}$ holds.

The following requirements suffice to ensure that a reference monitor enforces PSNI without causing illicit detection flows.

(i) To avoid illicit flows from an assignment statement ℓ: $w := expr$, execution of the monitored program must be blocked before reaching ℓ if ℓ does not satisfy Δ_S-Safe Assignment Statements condition (6.8).

(ii) To avoid illicit flows due to non-termination of a `while` statement ℓ_W with guard G_W, execution of the monitored program must be blocked before reaching ℓ_W if $\Gamma_{\mathcal{E}}(G_W) \neq \bot_\Lambda$ holds or if $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$ holds for some $G \in \Delta_S(\ell_W)$ that has been evaluated.

(iii) To avoid illicit detection flows, execution must not be blocked upon reaching a statement ℓ unless $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ holds, for all guards $G \in \Delta_S(\ell)$.

Notice, requirements ((i)) and ((ii)) can be satisfied by blocking execution of the monitored program in anticipation of executing a problematic statement. Even with this flexibility to block execution early, though, implementing a runtime mechanism that complies with requirement ((iii)) presents a challenge. A reference monitor must operate ignorant of code that has not yet been executed, so a reference monitor that is invoked when a control-flow statement ℓ is reached cannot determine if the body of ℓ contains a statement that should cause execution to be blocked in anticipation of the need to satisfy requirements ((i)) or ((ii)).

One solution is to employ a reference monitor $\mathcal{R}_{PS}$ that is conservative and, therefore, proceeds as if the body of every control-flow statement contains a statement that would require execution to be blocked for complying with requirements ((i)) and ((ii)). Such a reference monitor would block any execution upon reaching a control-flow statement having a guard G where $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$ holds. Given this (admittedly draconian) restriction, $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ holds for all guards that have been evaluated. Compliance with requirements ((ii)) and ((iii)) is thus guaranteed. Also, because no guard G will be evaluated where $\Gamma_{\mathcal{E}}(G)) \neq \bot_\Lambda$ holds, all guards will have label $\bot_\Lambda$ in composition $\mathcal{R}_{PS} \rhd S$ of the reference monitor $\mathcal{R}_{PS}$ and S. So program $\mathcal{R}_{PS} \rhd S$ satisfies Guard Restrictions for PSNI (page 232) and, therefore Δ_S-Safe Assignment Statements condition (6.8) is equivalent to $\Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$, which discharges requirement ((i)). The actions to implement reference monitor $\mathcal{R}_{PS}$ are given in Figure 6.2.

Hybrid Enforcement for PSNI. With a *hybrid enforcement mechanism*, a reference monitor invokes a static analyzer before and/or during executions of the monitored program. The static analyzer's results enable actions by the reference monitor to account for code that will be executed in the future or is as an alternative.

Our implementation of a PSNI hybrid enforcement mechanism employs a reference monitor $\mathcal{R}_{HPS}$ that invokes a static analyzer $\mathcal{T}_S(\cdot)$, where $\mathcal{T}_S(T)$ returns

upon S reaching •	action to be performed
• $w := expr$	require($\Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$)
• `if` G `then` ...	require($\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$)
• `while` G `do` ...	require($\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$)

Figure 6.2: Reference monitor $\mathcal{R}_{PS}$ for PSNI

true if T contains (i) no `while` statements and (ii) no assignment statement that would be blocked by $\mathcal{R}_{HPS}$. Thus, if $\mathcal{T}_S(T)$ returns *true* then fragment T is guaranteed to terminate.

Figure 6.3 gives the actions of $\mathcal{R}_{HPS}$. Notice that execution of control-flow statements are blocked by the $\mathcal{R}_{HPS}$ actions, as follows.

- A `while` statement with guard G is blocked if $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$ holds.
- An `if` statement with guard G and body T is blocked if $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$ holds and the `then` alternative S' or the `else` alternative S'' contains a `while` statement that $\mathcal{R}_{HPS}$ would block or contains an assignment statement that $\mathcal{R}_{HPS}$ would block.

This blocking prevents illicit detection flows, because a blocked execution cannot reveal information about results obtained by evaluating a guard G that satisfies $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$.

In addition, the $\mathcal{R}_{HPS}$ actions in Figure 6.3 ensure compliance with Guard Restrictions for PSNI, which allows Δ_S-Safe Assignment Statements condition (6.8) to be checked for an assignment statement ℓ by using a stack where `top()` satisfies

$$\texttt{top}(sps) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G). \tag{6.12}$$

upon S reaching •	action to be performed
• `if` G `then` S' `else` S'' `fi`	require($\Gamma_{\mathcal{E}}(G) = \bot_\Lambda \ \vee \ (\mathcal{T}_S(S') \wedge \mathcal{T}_S(S''))$) push($sps$, top($sps$) $\sqcup \Gamma_{\mathcal{E}}(G)$)
... `fi` •	pop(sps)
• `while` G `do` S `end`	require($\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$)
• $w := expr$	require(top(sps) $\sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$)

Figure 6.3: Hybrid enforcement $\mathcal{R}_{HPS}$ for PSNI

$\mathcal{R}_{HPS}$ implements such a stack by the actions it performs when an `if` or a `fi` is reached. No change to the stack is needed when a `while` statement is entered or exited, because the guard label will be $\bot_\Lambda$, so (6.12) continues to hold.

$\mathcal{R}_{HPS}$ and $\mathcal{R}_{PS}$ both block the same `while` statements. Some `if` statements that $\mathcal{R}_{PS}$ blocks are not blocked by $\mathcal{R}_{HPS}$. Here is an example:

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{H}} := x_{\mathsf{H}} + 1 \texttt{ else skip fi} \tag{6.13}$$

So $\mathcal{R}_{HPS}$ is more permissive than $\mathcal{R}_{PS}$. Hybrid enforcement can be as permissive as any type system or other form of static analysis. To be more permissive than this, a hybrid enforcement mechanism would have to identify and ignore program fragments that violate PSNI but cannot be reached during an execution. No static analyzer can perform such an analysis, since that analysis requires solving an undecidable problem.

6.1.3 Typing Rules to Enforce PSNI

To use type-correctness for enforcing PSNI, we employ judgements $\Gamma, \gamma \vdash_{\mathsf{ps}} S$, where typing context Γ gives a label $\Gamma(v)$ to each variable $v \in \mathit{Vars}(S)$, control context γ is a label, and S is an IMP program. Validity for judgements $\Gamma, \gamma \vdash_{\mathsf{ps}} S$ is defined as follows.

> **Valid Judgements for PSNI.** Judgement $\Gamma, \gamma \vdash_{\mathsf{ps}} S$ for a deterministic program S is *valid* if and only if
>
> (i) $(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\longrightarrow}_{\mathsf{ps}} V_{\sqsubseteq\lambda})$.
>
> (ii) $\gamma \sqsubseteq \Gamma(w)$ holds for target w of every assignment statement in S. □

A program S is defined to be type-correct if judgement $\Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} S$ can be derived using the typing rules in Figure 6.4.[3] Each of theses typing rules derives a valid judgement whenever all of the rule's hypotheses are valid. So if a program S is type-correct then $\Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} S$ is a valid judgement and, due to (i) in Valid Judgements for PSNI, the program will satisfy PSNI.

The typing rules in Figure 6.4 resemble the TINI typing rules in Figure 5.7, but with rule WHILE modified to ensure that every `while` statement in a type-correct program complies with the requirements of Guard Restrictions for PSNI (page 232): compliance with condition (i) follows from the first hypothesis of

[3] Here is a typing rule to handle `for`-loop statements, where $\mathit{tgts}(S)$ denotes the set of variables that are the targets of assignment statements in S.

$$\text{FOR:}\ \frac{\begin{array}{c} v \notin \mathit{tgts}(S),\ \ \mathit{tgts}(S) \cap \mathit{Vars}(\mathit{expr}') = \varnothing, \\ \Gamma_{\mathcal{E}}(\mathit{expr}) = \lambda,\ \ \Gamma_{\mathcal{E}}(\mathit{expr}') = \lambda',\ \ \Gamma(v) = \lambda'', \\ \gamma \sqcup \lambda \sqcup \lambda' \sqsubseteq \lambda'',\ \ \Gamma, \gamma \sqcup \lambda \sqcup \lambda' \sqcup \lambda'' \vdash_{\mathsf{ps}} S \end{array}}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{for } v := \mathit{expr} \texttt{ to } \mathit{expr}' \texttt{ do } S \texttt{ end}}$$

Observe that the hypotheses of rule FOR imply that (i) $v := \mathit{expr}$ and $v := v+1$ used to implement the `for`-loop statement each complies with Δ_S-Safe Assignment Statements condition (6.8) and (ii) the iterations eventually end because variable w and variables in upper bound expr' are not updated in body S of the `for`-loop statement.

$$\text{SKIP:}\ \frac{}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{skip}} \qquad \text{ASSIGN:}\ \frac{\gamma \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(v)}{\Gamma, \gamma \vdash_{\mathsf{ps}} v := expr}$$

$$\text{IF:}\ \frac{\Gamma_{\mathcal{E}}(expr) = \lambda, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ps}} S, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ps}} S'}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{if}\ expr\ \texttt{then}\ S\ \texttt{else}\ S'\ \texttt{fi}}$$

$$\text{WHILE:}\ \frac{\Gamma_{\mathcal{E}}(expr) = \bot_\Lambda, \quad \Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} S}{\Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} \texttt{while}\ expr\ \texttt{do}\ S\ \texttt{end}} \qquad \text{SEQ:}\ \frac{\Gamma, \gamma \vdash_{\mathsf{ps}} S, \quad \Gamma, \gamma \vdash_{\mathsf{ps}} S'}{\Gamma, \gamma \vdash_{\mathsf{ps}} S;\ S'}$$

Figure 6.4: Typing rules for PSNI compliance

rule WHILE and compliance with condition (ii) follows because having the control context be $\bot_\Lambda$ in the hypothesis and conclusion of rule WHILE forces $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ to hold for all $G \in \Theta_S(\ell_W)$. Since Guard Restrictions for PSNI is satisfied for every `while` statement, rule ASSIGN, which ensures that Θ_S-Safe Assignment Statements condition (5.24) holds for an assignment statement, then also ensures that Δ_S-Safe Assignment Statements condition (6.8) holds for that assignment statement.

6.2 Flow-Sensitive Noninterference

We now consider programs having variables with labels that vary. A *fixed variable* has the same label throughout all executions; a *flexible variable*, which we give tilde-topped names (e.g., $\widetilde{y}$), has a label that the program can change during an execution. Label changes allow a flexible variable to be used for different purposes during execution.

To specify label assignments for both fixed and flexible variables, we implement the label assignment using a new variable that is an array $\widetilde{\Gamma}[\cdot]$ of labels, indexed by variable names.[4] Writing $s.v$ to denote the value that a program state s gives to a variable v, the label that a program state s gives to a fixed or flexible variable v is $s.\widetilde{\Gamma}[v]$, and the label that s gives to an expression E is defined by:

$$s.\widetilde{\Gamma}_{\mathcal{E}}(E)\text{:}\ \begin{cases} \bot_\Lambda & \text{if } E \text{ is a constant } c \\ s.\widetilde{\Gamma}[v] & \text{if } E \text{ is a fixed or flexibile variable } v \\ \bigsqcup\limits_{1 \le i \le n} s.\widetilde{\Gamma}_{\mathcal{E}}(E_i) & \text{if } E \text{ is } f(E_1, E_2, \ldots, E_n) \end{cases}$$

[4] IMP statement execution does not depend on the labels of variables, but an enforcement mechanism could require this information. So whether $\widetilde{\Gamma}[\cdot]$ is actually stored in memory when an IMP program executes will depend on the enforcement mechanism.

We omit the state and just write $\widetilde{\Gamma}[v]$ or $\widetilde{\Gamma}_{\mathcal{E}}(E)$ when the state is clear from the context or if v is a fixed variable (since fixed variables are given the same label in all states).

Flow-sensitive PSNI (F-PSNI) generalizes progress sensitive noninterference (PSNI). PSNI is defined in terms of $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$, which has a formal definition (6.4) that employs state projections $s|_{\mathcal{W}}$ for comparing values of a given set $\mathcal{W}$ of (fixed) variables in corresponding program states produced during certain pairs of executions. With F-PSNI, which variables are compared depends on their label assignments in the states being compared. We formalize that by using state projections $s|_{F(\cdot)}$, where $F(\cdot)$ is a function from states to sets of variables, and $F(s)$ is the set of variables used for constructing the state projection from state s. The value of a fixed or flexible variable v in a state projection $s|_{F(\cdot)}$ is thus defined by:

$$s|_{F(\cdot)}.v\text{:}\quad \begin{cases} s.v & \text{if } v \in F(s) \\ ? & \text{otherwise} \end{cases}$$

And the formal definition of F-PSNI uses functions

$$\widetilde{V}_{\sqsubseteq\lambda}(s)\text{:}\ \ \{v \in \mathit{Vars}(S) \mid s.\widetilde{\Gamma}[v] \sqsubseteq \lambda\} \qquad \widetilde{V}_{\not\sqsubseteq\lambda}(s)\text{:}\ \ \{v \in \mathit{Vars}(S) \mid s.\widetilde{\Gamma}[v] \not\sqsubseteq \lambda\}$$

instead of $V_{\sqsubseteq\lambda}$ and $V_{\not\sqsubseteq\lambda}$ in the formal definition of PSNI, resulting in:

> **Flow-Sensitive PSNI (F-PSNI).** For a deterministic program S having states that assign values to fixed variables, to flexible variables, and to array $\widetilde{\Gamma}[\cdot]$ mapping variable names to labels in Λ with a partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\text{:}\ \ \widetilde{V}_{\not\sqsubseteq\lambda}(\cdot) \overset{S}{\not\rightarrow}_{\mathsf{ps}} \widetilde{V}_{\sqsubseteq\lambda}(\cdot)) \qquad \square$$

6.2.1 Flexible Variables and Enforcement

To prevent an update to a flexible variable from violating F-PSNI we must prevent that update from violating Δ_S-Safe Assignment Statements condition (6.8). But a change to the label of the flexible variables suffices.

> **Flexible Variable Label Update.** Label $\widetilde{\Gamma}[\widetilde{y}]$ is changed as follows
>
> $$\widetilde{\Gamma}[\widetilde{y}] \ :=\ \widetilde{\Gamma}_{\mathcal{E}}(\mathit{expr}) \ \sqcup \bigsqcup_{G \in \Delta_S(\ell)} \widetilde{\Gamma}_{\mathcal{E}}(G)$$
>
> just before executing an assignment statement ℓ: $\widetilde{y} := \mathit{expr}$ having a flexible variable $\widetilde{y}$ as its target. $\square$

With this new label for its target $\widetilde{y}$, assignment statement ℓ trivially satisfies Δ_S-Safe Assignment Statements condition (6.8).

These changes to labels, however, can cause F-PSNI violations. We illustrate with IMP program (6.14) below, where the set of labels is Λ_{LH} and fixed variable x_{H} has label H. The effect of performing Flexible Variable Label Update is shown

by depicting (in boxes) the new values for $\widetilde{\Gamma}[\widetilde{y}]$ during two executions. In one execution, $x_{\mathsf{H}} = 0$ holds initially; in the other execution, it doesn't.

$$\begin{array}{lll} x_{\mathsf{H}} = 0\text{:} & \boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{L}} & \boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{H}} \\ & \downarrow & \downarrow \\ & \multicolumn{2}{l}{\widetilde{y} := 0 \texttt{ if } x_{\mathsf{H}} = 0 \texttt{ then } \widetilde{y} := 0 \texttt{ else skip fi}} \\ & \uparrow & \\ x_{\mathsf{H}} \neq 0\text{:} & \boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{L}} & \end{array} \tag{6.14}$$

Letting s and s' denote the final states of these executions, the different final labels for $\widetilde{y}$ cause state projections $s|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)}$ and $s'|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)}$ to be different—even though s and s' give the same value (*viz.* 0) to flexible variable $\widetilde{y}$. One state projection maps $\widetilde{y}$ to 0; the other state projection maps $\widetilde{y}$ to "?". So, by definition, F-PSNI does not hold. And it shouldn't hold—an L-observer would read different values for $\widetilde{y}$ in the final states of executions having initial states that differ only in the value of $x_{\mathsf{H}} = 0$. Program (6.14) leaks $x_{\mathsf{H}} = 0$. However, if executing the `else` alternative also were to change label $\widetilde{\Gamma}[\widetilde{y}]$ to the label of guard $x_{\mathsf{H}} = 0$ then the L-observer would not read different values for $\widetilde{y}$, and F-PSNI would hold.

This kind of leak is possible with any control-flow statement ℓ having a guard G and a body containing an assignment statement to some flexible variable $\widetilde{y}$ where[5]

$$\widetilde{\Gamma}_{\mathcal{E}}(G) \sqcup \bigsqcup_{G \in \Delta_S(\ell)} \widetilde{\Gamma}_{\mathcal{E}}(G) \;\not\in\; \widetilde{\Gamma}[\widetilde{y}] \tag{6.15}$$

holds when ℓ is reached. Here are some examples.

- An `if` statement ℓ with only one alternative that executes an assignment statement to $\widetilde{y}$. So Flexible Variable Update changes $\widetilde{\Gamma}[\widetilde{y}]$ if that alternative is selected but not if the other alternative is selected.
- A `while` statement ℓ with a body that contains an assignment statement to $\widetilde{y}$. So Flexible Variable Update might change $\widetilde{\Gamma}[\widetilde{y}]$ if the body is executed because G was *true*. But $\widetilde{\Gamma}[\widetilde{y}]$ would not have been changed if G was *false* causing the body not to be executed.

In both cases, different labels for $\widetilde{y}$ at the exit control point for the control-flow statement cause a λ-observer to read different values for $\widetilde{y}$, potentially leaking G and the guards in $\Delta_S(\ell)$, since all of those guards affect whether the assignment statement to $\widetilde{y}$ is reached.

Such F-PSNI violations can be avoided if all executions of a control-flow statement ℓ update the label $\widetilde{\Gamma}[\widetilde{y}]$ of any flexible variable $\widetilde{y}$ that could be updated during an execution of ℓ—even if an assignment statement to $\widetilde{y}$ is not executed during that execution of ℓ.

[5] Recall from definition (6.7) on page 231 that $\Delta_S(\ell)$ is the set of guards that determine whether an execution reaches ℓ.

Untaken Trajectory Label Compensation. For every flexible variable $\widetilde{y}$ that is the target of an assignment statement in a control-flow statement ℓ that has guard G, label $\widetilde{\Gamma}[\widetilde{y}]$ is changed as follows

$$\widetilde{\Gamma}[\widetilde{y}] \;:=\; \widetilde{\Gamma}[\widetilde{y}] \sqcup \widetilde{\Gamma}_{\mathcal{E}}(G) \sqcup \bigsqcup_{G \in \Delta_S(\ell)} \widetilde{\Gamma}_{\mathcal{E}}(G)$$

just before executing ℓ. □

Studying an example is a good way to build an informal understanding of why Untaken Trajectory Label Compensation prevents guards from leaking; a sketch of the proof that it works can be found below.[6]

Consider a program having a set $\Lambda_{\mathsf{LMH}} = \{\mathsf{L}, \mathsf{M}, \mathsf{H}\}$ of labels, where $\mathsf{L} \sqsubset \mathsf{M} \sqsubset \mathsf{H}$ holds. Fixed variables x_{M} and x_{H} satisfy $\widetilde{\Gamma}[x_{\mathsf{M}}] = \mathsf{M}$ and $\widetilde{\Gamma}[x_{\mathsf{H}}] = \mathsf{H}$:

$$\ell_1\colon \widetilde{y} := 0;\ \texttt{if}\ x_{\mathsf{M}} = 0\ \texttt{then}\ \ell_2\colon \widetilde{y} := x_{\mathsf{H}}\ \texttt{else skip fi}$$

So Flexible Variable Label Update (FVLU) changes label $\widetilde{\Gamma}[\widetilde{y}]$ just before executing assignment statement ℓ_1 and just before executing assignment statement ℓ_2. And Untaken Trajectory Label Compensation (UTLC) changes $\widetilde{\Gamma}[\widetilde{y}]$ just before the `if` statement, because the body of that `if` statement contains an assignment statement to flexible variable $\widetilde{y}$. The new values for $\widetilde{\Gamma}[\widetilde{y}]$ are depicted below in boxes.

$$\begin{array}{l}
\boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{L}\ \text{(due to FVLU)}}\ \ \ell_1\colon \widetilde{y} := 0; \\
\boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{M}\ \text{due to (UTLC)}} \\
\texttt{if}\ x_{\mathsf{M}} = 0\ \texttt{then}\ \boxed{\widetilde{\Gamma}[\widetilde{y}] = \mathsf{H}\ \text{(due to FVLU)}}\ \ \ell_2\colon \widetilde{y} := x_{\mathsf{H}} \\
\qquad\qquad \texttt{else skip} \\
\texttt{fi}
\end{array} \tag{6.16}$$

The changes to $\widetilde{\Gamma}[\widetilde{y}]$ ensure that (i) all assignment statements to flexible variables satisfy Δ_S-Safe Assignment Statements condition (6.8) and (ii) all executions of (6.16) terminate in states satisfying $\Gamma_{\mathcal{E}}(x_{\mathsf{M}} = 0) \sqsubseteq \widetilde{\Gamma}[\widetilde{y}]$. Label $\widetilde{\Gamma}[\widetilde{y}]$ in

[6] Consider a λ-observer that reads $\widetilde{y}$ upon reaching a given control point and gets "?" in one execution but not in another. Thus, there is an execution where the λ-observer reads $\widetilde{y}$ and $\widetilde{\Gamma}[\widetilde{y}] \sqsubseteq \lambda$ holds at that point. We show that:

If G^* affects label $\widetilde{\Gamma}[\widetilde{y}]$ then $\widetilde{\Gamma}_{\mathcal{E}}(G^*) \sqsubseteq \widetilde{\Gamma}[\widetilde{y}]$ holds. (†)

If (†) holds then, by transitivity, $\widetilde{\Gamma}_{\mathcal{E}}(G^*) \sqsubseteq \lambda$ holds, which means that a λ-observer learning G^* by reading $\widetilde{y}$ is not a leak.

The argument that (†) holds for an execution proceeds by induction over that execution. Untaken Trajectory Label Compensation ensures that the labels on flexible variables $\widetilde{y}$ that are targets of assignment statements in a control-flow statement ℓ with guard G satisfy $\widetilde{\Gamma}_{\mathcal{E}}(G^*) \sqsubseteq \widetilde{\Gamma}[\widetilde{y}]$ for guards G^* in $\Delta_S(\ell) \cup \{G\}$. By definition, the set of guards that affect $\widetilde{\Gamma}[\widetilde{y}]$ at control point ℓ is $\Delta_S(\ell) \cup \{G\}$, so (†) holds when execution first reaches ℓ. As execution proceeds in the body of ℓ, assignment statements ℓ' to $\widetilde{y}$ may be encountered, causing Flexible Variable Label Update to change label $\widetilde{\Gamma}[\widetilde{y}]$. However, $\Delta_S(\ell) \subseteq \Delta_S(\ell')$ holds, by defintion. Therefore, the updated label does not invalidate $\widetilde{\Gamma}_{\mathcal{E}}(G^*) \sqsubseteq \widetilde{\Gamma}[\widetilde{y}]$ for any guard G^* that affects whether the update is reached and, thus, (†) continues to hold.

the final state, however, differs depending on whether the initial state satisfies $x_{\mathsf{M}} = 0$, which raises the question: Could the differences in label $\widetilde{\Gamma}[\widetilde{y}]$ at that final state leak guard $x_{\mathsf{M}} = 0$ to a λ-observer?

Whether a λ-observer can read $\widetilde{y}$ in the final state is determined by λ and by label $\widetilde{\Gamma}[\widetilde{y}]$ in that final state. The table to the right gives a case analysis. The "no" in every entry of column L indicates that an L-observer cannot read $\widetilde{y}$ no matter what value guard $x_{\mathsf{M}} = 0$ has in the initial state. So an L-observer monitoring the state can learn nothing about the value of guard $x_{\mathsf{M}} = 0$—the guard does not leak to L-observers. An M-observer, by attempting to read $\widetilde{y}$ in the final state, can detect whether $x_{\mathsf{M}} = 0$ evaluated to *true* when the `if` started executing. This is because the M-observer reads 0 or "?" depending on whether the initial state satisfies $x_{\mathsf{M}} = 0$. However, $x_{\mathsf{M}} = 0$ is not being leaked to the M-observer, since $\widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{M}} = 0) \sqsubseteq \mathsf{M}$ holds. Finally, an H-observer is able to read $\widetilde{y}$ whether the `then` or the `else` alternative was executed, and the H-observer is also allowed to learn the value of guard $x_{\mathsf{M}} = 0$ because $\widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{M}} = 0) \sqsubseteq \mathsf{H}$ holds so, again, there is no leak.

state at `if`	$\widetilde{\Gamma}[\widetilde{y}]$ at `fi`	can λ-observer read $\widetilde{y}$ for λ = ?		
		L	M	H
$x_{\mathsf{M}} = 0$	H	no	no	yes
$x_{\mathsf{M}} \neq 0$	M	no	yes	yes

6.2.2 Dynamic Enforcement of F-PSNI

To implement a dynamic enforcement mechanism for F-PSNI, a runtime environment would maintain array $\widetilde{\Gamma}[\cdot]$. But Untaken Trajectory Label Compensation requires knowledge of assignment statements that could be executed in the future, and a reference monitor cannot have access to that information. So we must employ some form of hybrid enforcement.

Since F-PSNI is based on PSNI and $\mathcal{R}_{HPS}$ enforces PSNI, we obtain a dynamic enforcement mechanism $\mathcal{R}_{FHPS}$ for F-PSNI by starting with the actions (see Figure 6.3) that implement $\mathcal{R}_{HPS}$. Recall the following about the $\mathcal{R}_{HPS}$ actions:

- Programs must satisfy Guard Restrictions for PSNI (page 232). So the label for a `while` statement guard must be $\bot_{\Lambda}$, and a `while` statement may only be nested within control-flow statements that have guards with label $\bot_{\Lambda}$.
- $\mathcal{T}_S(\cdot)$ is a static analyzer. An invocation $\mathcal{T}_S(T)$ returns *true* if fragment T of program S is guaranteed to terminate because T contains (i) no `while` statements and (ii) no assignment statement that would be blocked because Δ_S-Safe Assignment Statements (page 231 is not satisfied.
- While executing the body of the most recently entered control-flow statement ℓ, the $\mathcal{R}_{HPS}$ stack *sps* satisfies

$$\mathtt{top}(sps) = \bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G).$$

upon S reaching •	action to be performed
• ℓ: `if` G `then` S' `else` S'' `fi`	`require`$(\widetilde{\Gamma}_{\mathcal{E}}(G) = \bot_\Lambda \vee (\mathcal{T}_S(S') \wedge \mathcal{T}_S(S'')))$ `push`$(sps, \texttt{top}(sps) \sqcup \widetilde{\Gamma}_{\mathcal{E}}(G))$ *forall* $\widetilde{x} \in F\text{-}tgts(\ell)$: $\widetilde{\Gamma}[\widetilde{x}] := \widetilde{\Gamma}[\widetilde{x}] \sqcup \texttt{top}(sps)$
... `fi` •	`pop`(sps)
• ℓ: `while` G `do` S `end`	`require`$(\widetilde{\Gamma}_{\mathcal{E}}(G) = \bot_\Lambda)$ *forall* $\widetilde{x} \in F\text{-}tgts(\ell)$: $\widetilde{\Gamma}[\widetilde{x}] := \widetilde{\Gamma}[\widetilde{x}] \sqcup \texttt{top}(sps)$
• $\widetilde{w} := expr$	$\widetilde{\Gamma}[\widetilde{w}] := \widetilde{\Gamma}_{\mathcal{E}}(expr) \sqcup \texttt{top}(sps)$
• $w := expr$	`require`$(\texttt{top}(sps) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w))$

Figure 6.5: Hybrid enforcement $\mathcal{R}_{HPS}$ for F-PSNI

because $\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$ holds due to Guard Restrictions for PSNI.

Figure 6.5 gives the actions that implement $\mathcal{R}_{FHPS}$. They augment the actions $\mathcal{R}_{HPS}$ invokes when execution reaches a control-flow statement, and they include a new action to be invoked when execution reaches an assignment statement to a flexible variable. In particular, the actions for the start of control-flow statements assume that $F\text{-}tgts(\ell)$ returns the set of flexible variables that are targets of assignment statements in the control-flow statement with label ℓ. This static analysis thus identifies the flexible variables needed to implement Untaken Trajectory Label Compensation in the action for assignment statements to flexible variables.

6.2.3 Typing Rules to Enforce F-PSNI

The typing rules to enforce F-PSNI are based on the typing rules to enforce PSNI given in Figure 6.4. To enforce Δ_S-Safe Assignment Statements for updates to flexible variables, the F-PSNI typing rules require labels that comply with Flexible Variable Label Update. But instead of employing Untaken Trajectory Label Compensation and allowing differences in the labels that a flexible variable can have when execution reaches a given control point, the F-PSNI typing rules associate a single label assignment with each control point. For programs that satisfy this stronger restriction, the label for a flexible variable may still be changed from one control point to the another during an execution—but only in limited ways. Moreover, there will be programs that comply with F-PSNI but are not type correct, because the label given to a flexible variable at a given control point depends on earlier execution.

A *label annotated program* gives the text for a program S along with a single label assignment for each control point in S. One way that we will represent a

label annotated program is with the notation $S\,/\,\widetilde{\Gamma}^*$ where S is a program and $\widetilde{\Gamma}^*$ maps each control point ℓ in S to an array $\widetilde{\Gamma}_\ell[\cdot]$ of labels indexed by variable names. So upon reaching control point ℓ during an execution, $\widetilde{\Gamma}^*(\ell)$ gives a value for array $\widetilde{\Gamma}[\cdot]$ in the current state.

Another representation for a label annotated program is to insert a *label annotation* at each control point in the text of the program. The label annotation $\{\widetilde{\Gamma}_\ell\}$ appearing at a control point ℓ in the program text indicates that array $\widetilde{\Gamma}[\cdot]$ in the current state equals $\widetilde{\Gamma}_\ell[\cdot]$ whenever control point ℓ is reached during an execution. IMP programs have a control point before and after each statement, so a label annotated IMP program can be represented by inserting a label annotation before and after each statement in the program text. Here is an example of a label annotated IMP program, assuming that the S_i are `skip` or assignment statements and the $\widetilde{\Gamma}_i$ are label assignments.[7]

$$
\begin{array}{l}
\{\widetilde{\Gamma}_1\}\ \ell_1\colon S_1\ \{\widetilde{\Gamma}_2\} \\
\ell_2\colon \texttt{if}\ G\ \texttt{then}\ \{\widetilde{\Gamma}_3\}\ \ell_3\colon S_2; \{\widetilde{\Gamma}_4\} \\
\qquad\quad \texttt{else}\ \{\widetilde{\Gamma}_5\}\ \ell_4\colon S_3\ \{\widetilde{\Gamma}_6\} \\
\quad \texttt{fi}\ \{\widetilde{\Gamma}_7\}
\end{array}
$$

It will be helpful to combine the two different representations for label annotated programs, writing

$$\{\widetilde{\Gamma}'\}\ S\,/\,\widetilde{\Gamma}^*\ \{\widetilde{\Gamma}''\}$$

to indicate that label assignment $\widetilde{\Gamma}'$ is associated with the entry control point for S, label assignment $\widetilde{\Gamma}''$ is associated with the exit control point for S, and the label assignments given by $\widetilde{\Gamma}^*$ are used for all other control points in S. Omission of the initial label annotation $\{\widetilde{\Gamma}'\}$ or the final label annotation $\{\widetilde{\Gamma}''\}$ means that $\widetilde{\Gamma}^*$ is providing those label assignments.

Judgements and Typing Rules for F-PSNI. To use type-checking for enforcing F-PSNI, we employ typing rules that derive judgements $\gamma \vdash_{\widetilde{\mathsf{pS}}} S\,/\,\widetilde{\Gamma}^*$ for label annotated programs $S\,/\,\widetilde{\Gamma}^*$. Validity of these judgements is defined as follows.

[7] The placement of label annotation $\{\widetilde{\Gamma}_2\}$ immediately after S_1 and before `if` statement ℓ_2 illustrates that IMP sequential compositions $S; S'$ have a single control point serving both as the exit control point of S and as the entry control point of S'. Also notice that the exit control point for an IMP `if` statement is distinct from the exit control points for its `then` alternative and its `else` alternative. Therefore, the exit control point for an `if` statement always will have two direct predecessor control points.

$$\text{SKIP:}\ \frac{}{\gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \texttt{skip}\ \{\widetilde{\Gamma}\}} \qquad \text{SEQ:}\ \frac{\gamma \vdash_{\widetilde{\mathsf{ps}}} S_1 / \widetilde{\Gamma}_1^*\ \{\widetilde{\Gamma}\},\ \ \gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ S_2 / \widetilde{\Gamma}_2^*}{\gamma \vdash_{\widetilde{\mathsf{ps}}} S_1 / \widetilde{\Gamma}_1^*; \{\widetilde{\Gamma}\}\ S_2 / \widetilde{\Gamma}_2^*}$$

$$\text{A-ASSIGN:}\ \frac{\gamma \sqcup \widetilde{\Gamma}_{\mathcal{E}}(expr) \sqsubseteq \widetilde{\Gamma}[w]}{\gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ x := expr\ \{\widetilde{\Gamma}\}} \qquad \text{F-ASSIGN:}\ \frac{}{\gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \widetilde{v} := expr\ \{\widetilde{\Gamma}[\widetilde{v} \leftarrow \gamma \sqcup \widetilde{\Gamma}_{\mathcal{E}}(expr)]\}}$$

$$\text{IF:}\ \frac{\widetilde{\Gamma}_{\mathcal{E}}(expr) = \lambda,\ \ \gamma \sqcup \lambda \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ S_1 / \widetilde{\Gamma}_1^*\ \{\widetilde{\Gamma}'\},\ \ \gamma \sqcup \lambda \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ S_2 / \widetilde{\Gamma}_2^*\ \{\widetilde{\Gamma}'\}}{\gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \texttt{if}\ expr\ \texttt{then}\ \{\widetilde{\Gamma}\}\ S_1 / \widetilde{\Gamma}_1^*\ \{\widetilde{\Gamma}'\}\ \texttt{else}\ \{\widetilde{\Gamma}\}\ S_2 / \widetilde{\Gamma}_2^*\ \{\widetilde{\Gamma}'\}\ \texttt{fi}\ \{\widetilde{\Gamma}'\}}$$

$$\text{WHILE:}\ \frac{\widetilde{\Gamma}_{\mathcal{E}}(expr) = \bot_\Lambda,\ \ \bot_\Lambda \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ S / \widetilde{\Gamma}^*\ \{\widetilde{\Gamma}\}}{\bot_\Lambda \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \texttt{while}\ expr\ \texttt{do}\ \{\widetilde{\Gamma}\}\ S / \widetilde{\Gamma}^*\ \{\widetilde{\Gamma}\}\ \texttt{end}\ \{\widetilde{\Gamma}\}}$$

$$\text{RELAB:}\ \frac{\gamma' \sqsubseteq \gamma,\ \ \widetilde{\Gamma}'_1 \sqsubseteq \widetilde{\Gamma}_1,\ \ \widetilde{\Gamma}_2 \sqsubseteq \widetilde{\Gamma}'_2,\ \ \gamma \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}_1\}\ S / \widetilde{\Gamma}^*\ \{\widetilde{\Gamma}_2\}}{\gamma' \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}'_1\}\ S / \widetilde{\Gamma}^*\ \{\widetilde{\Gamma}'_2\}}$$

Notation:

$\widetilde{\Gamma}[\widetilde{x} \leftarrow \Gamma_{\mathcal{E}}(expr)]$: $\widetilde{\Gamma}$, except that $\widetilde{\Gamma}[\widetilde{x}]$ equals label $\widetilde{\Gamma}_{\mathcal{E}}(expr)$

$\widetilde{\Gamma} \sqsubseteq \widetilde{\Gamma}'$: $(\forall v, \widetilde{v} \in Vars(S)\colon\ \widetilde{\Gamma}[v] = \widetilde{\Gamma}'[v]\ \wedge\ \widetilde{\Gamma}[\widetilde{v}] \sqsubseteq \widetilde{\Gamma}'(\widetilde{v}))$

Figure 6.6: Typing rules for F-PSNI compliance

Valid Judgements for FSNI. Judgement $\gamma \vdash_{\widetilde{\mathsf{ps}}} S / \widetilde{\Gamma}^*$, where S is a deterministic program, is *valid* if and only if

(i) $(\forall \lambda \in \Lambda\colon\ \widetilde{V}_{\not\sqsubseteq \lambda}(\cdot) \overset{S}{\not\longrightarrow}_{\mathsf{ps}} \widetilde{V}_{\sqsubseteq \lambda}(\cdot))$.

(ii) $\gamma \sqsubseteq \widetilde{\Gamma}_\ell[w]$ holds for the target w and the exit control point ℓ of each assignment statement that appears in S.[8] □

A label annotated program $S / \widetilde{\Gamma}^*$ is considered type-correct if the typing rules in Figure 6.6 can derive the judgement $\bot_\Lambda \vdash_{\widetilde{\mathsf{ps}}} S / \widetilde{\Gamma}^*$. By design, these typing rules only derive valid judgements from valid hypotheses. So a type-correct label annotated program $S / \widetilde{\Gamma}^*$ will satisfy F-PSNI, due to (i) in the above definition of validity for $\bot_\Lambda \vdash_{\widetilde{\mathsf{ps}}} S / \widetilde{\Gamma}^*$.

To enforce F-PSNI, the typing rules reject programs that would allow λ-observers to distinguish executions starting from initial values that differ only

[8] For assignments to fixed variables, the value of $\widetilde{\Gamma}[w]$ is the same at all control points. So requirement (ii) is equivalent to what is required by Valid Judgements for PSNI (page 236). With assignments to flexible variables, however, it is the value of $\widetilde{\Gamma}[w]$ after the assignment statement executes that determines whether w can be read by a λ-observer.

in the values of fixed or flexible variables v satisfying $\widetilde{\Gamma}[v] \not\in \lambda$. The PSNI typing rules in Figure 6.4 achieve this for programs that only have fixed variables. The F-PSNI typing rules in Figure 6.6 can be seen as generalizing these PSNI typing rules in order to accomodate flexible variables. Specifically, judgements in the F-PSNI typing rules have label annotated programs in place of the programs appearing in the judgements of the PSNI typing rules, the F-PSNI typing rules ensure that a single label assignment is associated with each control point, and rule F-ASSIGN ensures that an assignment statement to a flexibile variable does not violate Δ_S-Safe Assignment Statements because the rule generates a label assignment as if Flexible Variable Label Update had been performed.

Figure 6.7 illustrates a use of the F-PSNI typing rules, giving a derivation that shows program (6.14) is type-correct and, therefore, satisfies F-PSNI.[9] Steps 4 and 5 derive judgements for the `then` and `else` alternatives. These judgements have exit control points giving different labels for flexible variable $\widetilde{y}$, necessitating step 7 to change label $\widetilde{\Gamma}[\widetilde{y}]$ at the exit control point of the `else` alternative (step 5) to match the label assignment for $\widetilde{y}$ at the exit control point of the `then` alternative (step 4). The hypotheses for rule IF then is discharged using the judgements in steps 4 and 7.

6.3 *Other Noninterference Policies

Noninterference policies for a program S all have the same general form

$$(\forall s, s' \in \mathit{Init}_S\colon\ s \sim s' \ \Rightarrow\ [\![S]\!]^{\eta}(s) \approx [\![S]\!]^{\eta}(s')) \tag{6.17}$$

where Init_S is the set of initial states for S, predicate $s \sim s'$ is satisfied by any pair of states s and s' that are indistinguishable to attackers, function $[\![S]\!]^{\eta}(s)$ evaluates to a description of the effects when S begins execution in state s, and predicate $\xi \approx \xi'$ is satisfied if ξ and ξ' are descriptions of execution effects that would be indistinguishable to attackers. Different choices for the predicates and the function result in the different noninterference policies discussed in this book and elsewhere.

The descriptions of the execution effects that are surfaced by function $[\![S]\!]^{\eta}(\cdot)$ reflect any assumptions we are making about programs, about the runtime environment, and about what an attacker can monitor and detect. We have assumed thus far that programs are deterministic. For TINI in §5.3, we assumed that attackers only can monitor the final states of terminating executions, so for $[\![S]\!]^{\eta}(\cdot)$ we used function $[\![S]\!](\cdot)$ from an initial state to a final state; for PSNI in §6.1, we assumed that an attacker can monitor intermediate states of all executions, so for $[\![S]\!]^{\eta}(\cdot)$ we used function $[\![S]\!]^{\mathrm{tr}}(\cdot)$ from an initial state to an execution trace.

Nondeterministic programs and concurrent programs can be handled by instantiating $[\![S]\!]^{\eta}(\cdot)$ with a function that evaluates to a set.

[9] The set of labels used by this program is $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$. Therefore $\bot_{\Lambda_{\mathsf{LH}}}$ is L, so type-correctness requires derivation of judgement $\mathsf{L} \vdash_{\widetilde{\rho s}} S \,/\, \widetilde{\Gamma}^*$.

1. $\mathsf{L} \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \widetilde{y} := 0;\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}$
   ... instance of rule F-ASSIGN.

2. $\mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}\ ]\}\ \widetilde{y} := 0\ \{(\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}\ ])[\widetilde{y} \leftarrowtail \mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \sqcup \mathsf{L}]\}$
   ... instance of rule F-ASSIGN.

3. $(\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}\ ])[\widetilde{y} \leftarrowtail \mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \sqcup \mathsf{L}]) \;=\; \widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]$
   ... substitution and simplifcation.

4. $\mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \ \widetilde{y} := 0\ \ \{(\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}])\}$
   ... instance of rule RELAB with 2, 3.

5. $\mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \texttt{skip}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}$
   ... instance of rule SKIP.

6. $\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}] \;\sqsubseteq\; \widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]$
   ... definition of $\sqsubseteq$.

7. $\mathsf{L} \sqcup \widetilde{\Gamma}_{\mathcal{E}}(x_{\mathsf{H}} = 0) \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \texttt{skip}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   ... instance of rule RELAB with 5, 6.

8. $\mathsf{L} \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}$
   $\texttt{if}\ x_{\mathsf{H}} = 0\ \texttt{then}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \widetilde{y} := 0\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   $\texttt{else}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \texttt{skip}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   $\texttt{fi}$
   $\{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   ... instance of rule IF with 4, 7.

9. $\mathsf{L} \vdash_{\widetilde{\mathsf{ps}}} \{\widetilde{\Gamma}\}\ \widetilde{y} := 0;\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}$
   $\texttt{if}\ x_{\mathsf{H}} = 0\ \texttt{then}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \widetilde{y} := 0\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   $\texttt{else}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{L}]\}\ \texttt{skip}\ \{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   $\texttt{fi}$
   $\{\widetilde{\Gamma}[\widetilde{y} \leftarrowtail \mathsf{H}]\}$
   ... instance of rule SEQ with 1, 8.

Figure 6.7: Example F-PSNI derivation

- With a nondeterministic program S, each element in the set $[\![S]\!]^{\eta}(s)$ would describe the execution effects resulting from one of the possible sequences of outcomes for the nondeterministic choices made during an execution that starts in initial state s.

- With a concurrent program S, each element in the set $[\![S]\!]^{\eta}(s)$ would describe the execution effects produced by one of the possible interleavings of atomic actions by the processes that comprise S, starting from initial state s. Which interleavings are considered possible would depend on the scheduler, capacity bounds on resources, and the semantics of any synchronization mechanisms that processes use.

For cases where function $[\![S]\!]^{\eta}(\cdot)$ evaluates to sets, a straightforward definition for predicate $\mathcal{X} \approx \mathcal{X}'$ is $\mathcal{X} = \mathcal{X}'$. This definition is inadequate, however, for attackers that learn by monitoring execution of the system. First, any single execution might be in the sets $[\![S]\!]^{\eta}(s)$ for multiple initial states s. Second, even if an attacker could instigate executions from a given initial state s, there is no guarantee that attacker would be able to generate the full contents of $[\![S]\!]^{\eta}(s)$, because nondeterministic choices, by definition, reflect aspects of the runtime environment that attackers cannot control.

If executions are affected by nondeterministic choices that an attacker cannot control, then we might want to ascertain whether an attacker who observes executions from some initial state s can rule out having witnessed an execution from a different initial state s'. That noninterference policy can be formalized by using the following weaker definition for $\mathcal{X} \approx \mathcal{X}'$:

$$\mathcal{X} \approx \mathcal{X}' \colon \ (\forall \xi \in \mathcal{X} \colon \ (\exists \xi' \in \mathcal{X}' \colon \ \xi \approx \xi'))$$

Noninterference policies where $\approx$ involves an existential quantifier are known as *possibilistic noninterference*.

With nondeterministic and with concurrent programs, we might also want to impose limits on what an observer might learn from observing a set of executions. The set could allow an attacker to approximate the likelihood of an initial state given how often various executions are observed. As an example, consider the nondeterministic program

$$\begin{array}{lll} S\colon \texttt{if } x_{\mathsf{H}} > 0 \texttt{ then } & [x_{\mathsf{L}} := 1 & \blacksquare_{.99} \ \ x_{\mathsf{L}} := 2] \\ \qquad\qquad\quad \texttt{else } & [x_{\mathsf{L}} := 1 & \blacksquare_{.5} \ \ x_{\mathsf{L}} := 2] \\ \texttt{fi} & & \end{array}$$

where $[S \blacksquare_p S']$ is the syntax for a statement that executes S or S', choosing S with probability p and choosing S' with probablilty $1 - p$. An L-observer who instigates multiple executions of S should be able to predict whether $x_{\mathsf{H}} > 0$ holds based on the distribution of the final values observed for x_{L} in those executions—if $x_{\mathsf{L}} = 1$ holds often then $x_{\mathsf{H}} > 0$ holds with high probability. We would want to use a version of noninterference where $[\![S]\!]^{\eta}(\cdot)$ produces a probability distribution for possible executions and where $[\![S]\!]^{\eta}(s) \approx [\![S]\!]^{\eta}(s')$ holds

if the probability distributions $[\![S]\!]^\eta(s)$ and $[\![S]\!]^\eta(s')$ are indistinguishable to attackers.

Changes to the value of a program variable are not the only observable effects of execution, though. For example, the elapsed time between changes to a variable while a given program executes can reveal information about the values of other variables. One reason is that the time required for performing certain operations can depend on the values of the operands. Another reason is that access to a variable may depend on whether that variable was recently accessed and, therefore, resides in a cache. So from information about execution timing, an attacker sometimes can make inferences about the values of variables that the attacker cannot read directly. A noninterference policy could be formulated that prohibits such leaks, but its states would have to provide information about execution times, cache contents, and any other hardware or operating system resources that affect execution timings.

Finally, we should acknowledge that attackers monitor system interfaces to learn about executions. The noninterference policies we have been discussing concern states. So we would have to model the interfaces—what is visible and what can be changed—using states. The domain and range for states then would embody assumptions about the interfaces that attackers access. For example, the assumption that attackers have access to an interface could then be implemented by identifying certain variables with the interface state and choosing labels for those variables so access by attackers is allowed.

Notes and Reading for Chapter 6

Success with enforcing TINI in sequential programs prompted researchers to investigate defending against more-capable attackers, developing enforcement mechanisms that would be more permissive, and supporting programning languages that had nondeterminacy and concurrency.

New typing rules in Volpano and Smith [18] for enforcing (what later became known as) termination-sensitive noninterference (TSNI) were the first step. With TSNI, attackers are assumed able to distinguish between terminating and nontermination executions. TSNI, therefore, defends against more-capable attackers than TINI, since TINI ignores nonterminating executions.

Another significant step towards a more realistic characterization of attackers was the advent of an interactive model of computation in Askarov et al. [2] to replace the batch model used for TINI and TSNI. In this interactive model, attackers could observe outputs and/or intermediate states during an execution. The additional leaks that became possible were captured by new variants of noninterference, later named by Askarov and Myers [3] *progress insensitive noninterference* (PINI) and *progress sensitive noninterference* (PSNI). Askarov et al. [2] also shows that that the certification conditions in Denning [5] to enforce TINI in the batch model of computing suffice for enforcing PINI in this interactive model of computing, which explains why the typing rules in Figure 6.4 resemble the typing rules in Figure 5.7. Finally, Askarov et al. [2] debunks

a widely-held belief that only a single secret bit could be leaked if observers can detect that an execution has been terminated by a dynamic enforcement mechanism.

The design of more-permissive enforcement mechanisms has also attracted considerable attention from researchers. Most type systems had employed flow-insensitive analyses. Hunt and Sands [7] gives typing rules that implement a flow-sensitive analysis.[10] Those typing rules (the basis for the typing rules in Figure 6.6) associate a label assignment with each control point, where labels assigned to variables are allowed to change as execution proceeds, a flexibility Denning [5] allows but the subsequent type system formalization in Volpano, Smith, and Irvine [17] does not. Moreover, Hunt and Sands [7] observes that the label assignments generated by their typing rules is an abstract characterization of dependencies, leading to a surprising result: if assignments statements to fresh variables can be added then a flow-sensitive analysis of a program having flexible variables can be replaced by a flow-insensitive analysis of an equivalent program where all variables have a fixed label assignment.

Dynamic enforcement mechanisms—reference monitors and hybrid enforcement mechanisms—were also seen by researchers as a promising avenue for achieving increased permissiveness. Hedin and Sabelfeld [6, §4.3] surveys dynamic enforcement mechanisms found in the literature prior to 2012. These mechanisms differ in the information flow policy they enforce, the events they intercept, the actions they take to prevent policy violations, the analyses they use, and whether they deliver increased permissiveness over a type system. Space does not permit a detailed enumeration here of those results.

We might expect that a dynamic enforcement mechanism would have to analyze the code that will not be executed, since implicit flows can be caused by assignment statements in the untaken alternative of an `if` statement or in an unexecuted body of a `while` statement. However, Sabelfeld and Russo [14] shows that a reference monitor like $\mathcal{R}_{TI}$ can enforce TINI without analyzing assignment statements that are not being executed. The relationship between the permissiveness of type systems (which typically analyze all statements in a program) and reference monitors (which only analyze the statements that execute) is subtle. For flow-sensitive labels and programs that produce output, Russo and Sabelfeld [12] shows that a reference monitor to enforce PINI will not accept all executions of programs that are considered type-correct by the Hunt and Sands [7] flow-sensitive type system. So neither enforcement mechanism is more permissive. However, Russo and Sabelfeld [12] also shows that a hybrid enforcement mechanism can be more permissive than the typing rules in Hunt and Sands [7].

To learn more about information flow control, good starting points are (i) the Sabelfeld and Myers survey [13] of language-based approaches and (ii) the

[10] Hunt and Sands [7] is not the first flow-sensitive analysis for checking noninterference. It is preceded by Amtoft and Banerjee [1], which gives a Hoare-style logic that implements a flow-sensitive analysis for verifying that certain variables are independent. The first Hoare-style logic for reasoning about information flow policies of sequential and concurrent programs appears in Reitman and Andrews [11], but that analysis was not flow-sensitive.

Hedin and Sabelfeld tutorial [6] on how various forms of noninterference can be enforced. See Kozyri et al. [8] for an in-depth exploration of the various kinds of information flow policies. Sabelfeld and Sands [15] is considered the authoritative treatment of declassification. Also, consider experimenting with a programming language that uses types to enforce information flow control. Jif [9, 10] extends Java and has been used to build non-trivial applications. One such application that is well documented is the Civitas [4] coercion-resistant, voter verifiable electronic voting system. Flow Caml [16] is a prototype that extends the Caml language.

Bibliography

[1] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Proceedings 11th International Symposium on Static Analysis (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115, Berlin, Heidelberg, August 2004. Springer-Verlag.

[2] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, October 2008. Springer-Verlag.

[3] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In Andrew D. Gordon, editor, *Programming Languages and Systems*, ESOP'10, pages 64–84, Berlin, Heidelberg, March 2010. Springer-Verlag.

[4] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, May 2008.

[5] Dorothy E. Denning. *Secure Information Flow in Computer Ssystems*. PhD thesis, Purdue University, USA, 1975.

[6] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security – Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[7] Sebastian Hunt and David Sands. On flow-sensitive security types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90. ACM, January 2006.

[8] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. Expressing information flow properties. *Foundations and Trends Privacy and Security*, 3(1):1–102, 2022.

[9] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241. ACM, January 1999.

[10] Andrew C. Myers and et al. Jif: Java information flow (software release), July 2001. http://www.cs.cornell.edu/jif.

[11] Richard P. Reitman and Gregory R. Andrews. Certifying information flow properties of programs: An axiomatic approach. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, POPL '79, pages 283–290. ACM, January 1979.

[12] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199. IEEE Computer Society, July 2010.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[14] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In Amir Pnueli, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference (PSI)*, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer, June 2009.

[15] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[16] Vincent Simonet. The Flow Caml system (version 1.00): Documentation and user's manual. http://cristal.inria.fr/~simonet/soft/flowcaml/manual/index.html, July 2003.

[17] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[18] Dennis M. Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop (CSFW '97)*, pages 156–169. IEEE Computer Society, June 1997.