Chapter 10

Information Flow Control: Advanced

There are various reasons that a λ-observer could have access to the variables in $V_{\sqsubseteq\lambda}$ while a program S is executing.

- Variables in $V_{\sqsubseteq\lambda}$ might be used to model the channels that a λ-observer reads while S is executing. Outputs produced during execution would be implemented as updates to those variables.
- 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 would become a λ-observer.

Therefore, we now consider noninterference policies for settings where the Batch and Asynchronous assumptions (page 253) that we had 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 S.

Termination Detection implies that a λ-observer seeing no change to the variables in $V_{\sqsubseteq\lambda}$ can determine whether the apparent lack of activity is because execution of S has terminated or because all of the variables that S is changing are from $V_{\not\sqsubseteq\lambda}$.

10.1 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 the sequence of intermediate states produced by a terminating or a non-terminating execution of S. So PSNI specifies limits on what could 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 $V_{\sqsubseteq\lambda}$ during an execution—even though nothing about the initial values of variables from $V_{\not\sqsubseteq\lambda}$ is revealed in the final values of variables in $V_{\sqsubseteq\lambda}$.

$$\begin{array}{l} \ell_1\colon y := 0;\ \ell_2\colon z := 0; \\ \ell_3\colon \texttt{if}\ x = 0\ \texttt{then}\ \ell_4\colon y := 1;\ \ell_5\colon z := 2 \\ \qquad\qquad\ \texttt{else}\ \ell_6\colon x := 4;\ \ell_7\colon z := 2;\ \ell_8\colon y := 1 \\ \quad \texttt{fi}\ \ell_9\colon \end{array} \tag{10.1}$$

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 intermediate states reveals information about the initial value of variable $x \in V_{\not\sqsubseteq\mathsf{L}}$. Yet program (10.1) does satisfy TINI, because the final values of y and z are the same for all initial values of x.

For formally specifying PSNI, we define a 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 values for the variables in $\mathcal{W}$. As an example, $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold if S is program (10.1), 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 assigned during another. Figure 10.1 illustrates by giving two *execution traces*, each specifying an initial state followed by the intermediate states produced during an execution of program (10.1). We write "$x \mapsto val$" to indicate that a state maps variable name x to a value val, where ? represents an unknown value. So execution trace σ describes the execution from an initial state satisfying $x = 0$, and execution trace σ' describes execution from an initial state satisfying $x = 1$.

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 can check this requirement by constructing *projected execution traces* $\langle \sigma|_{\{y,z\}} \rangle$ and $\langle \sigma'|_{\{y,z\}} \rangle$,

[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.

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

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

Figure 10.1: Sequences σ and σ' of intermediate states for executions of (10.1)

which give the sequences of changed values for y and z.

$$\langle \sigma|_{\{y,z\}} \rangle\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{10.2}$$

$$\langle \sigma'|_{\{y,z\}} \rangle\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 2 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{10.3}$$

The fourth states of (10.2) and (10.3) 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 that is monitoring y and z.

Construction of a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ involves two steps: (i) variables not in $\mathcal{V}$ are deleted from each state of σ and (ii) runs of identical states in the result are collapsed into a single state.[2] And predicate $\sigma =_{\mathcal{V}} \sigma'$ is defined to hold if de-stuttered trace projections $\langle \sigma|_{\mathcal{V}} \rangle$ and $\langle \sigma'|_{\mathcal{V}} \rangle$ form trace projections having the same length and having identical i^{th} states, for each i:

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

We collapse runs of identical states when forming a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ in order to obtain the sequence of states that would be seen by an observer monitoring $\mathcal{V}$, since state transitions are detectable by such an observer only

[2]We formalize these steps, as follows. For a sequence σ states, define $\sigma[i]$ to be its i^{th} state, $\sigma[i..]$ to be the suffix starting with $\sigma[i]$, and *trace projection* $\sigma|_{\mathcal{V}}$ to be the sequence of state projections for the states in σ:

$$\sigma|_{\mathcal{V}}\colon \ \sigma[1]|_{\mathcal{V}}\ \sigma[2]|_{\mathcal{V}}\ \ldots$$

The *de-stutter* operation $\langle \sigma \rangle$ collapses runs of identical states in σ is defined formally by:

$$\langle \sigma \rangle\colon \begin{cases} \sigma & \text{if } len(\sigma) = 1 \ \vee \ (\forall 1 < i \leq len(\sigma)\colon \sigma[1] = \sigma[i]) \\ \sigma[1]\ \langle \sigma[2..] \rangle & \text{if } len(\sigma) > 1 \ \wedge \ \sigma[1] \neq \sigma[2] \\ \langle \sigma[2..] \rangle & \text{otherwise} \end{cases}$$

if the value of some variable in $\mathcal{V}$ changes. Notice, a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ might have finite length even though σ has infinite length—this case arises with a non-terminating 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 defining PSNI, we are concerned with pairs of projected execution traces that start in initial states s and s' satisfying $s =_{\overline{\mathcal{V}}} s'$.

$$\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}\text{:}\quad (\forall s, s' \in \mathit{Init}_S\text{:}\ s =_{\overline{\mathcal{V}}} s' \ \Rightarrow\ [\![S]\!]^{\mathsf{tr}}(s) =_{\mathcal{W}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{10.4}$$

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 $\mathcal{W}$.

To illustrate the use of definition (10.4), we check whether $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ holds when S is program (10.1). $\overline{\mathcal{V}}$ is $\{y, z\}$ (since $\mathcal{V}$ is $\{x\}$), and $\mathcal{W}$ is $\{y, z\}$, resulting in:

$$(\forall s, s' \in \mathit{Init}_S\text{:}\ s =_{\{y,z\}} s' \ \Rightarrow\ [\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{10.5}$$

The earlier claim that $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold would be confirmed by showing that (10.5) does not hold. The initial states of σ and σ' in Figure 10.1 do satisfy antecedent $s =_{\{y,z\}} s'$, but consequent $[\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')$ of (10.5) does not hold, since (10.2) and (10.3) are different. As expected, we have shown 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 $V_{\not\in\lambda}$ 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.

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

10.1.1 PSNI Enforcement

The possibility of non-termination by a `while` statement can cause an implicit flow that violates PSNI. For 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
```
(10.6)

depends on which of $y > 0$ or $y \leq 0$ holds initially. 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 the variables in those guards to target x of assignment statement ℓ. If $\Gamma_{\mathcal{E}}(z = 0) \sqcup \Gamma_{\mathcal{E}}(y \neq 0) \not\sqsubseteq \Gamma(x)$ holds then this implicit flow also is an illicit flow—even though ℓ complies with Θ_S-Safe Assignment Statements condition (9.26). We conclude that restrictions beyond those required for enforcing TINI are needed for enforcing PSNI.

In general, there will be an implicit flow to any statement executed after a `while` statement. This implicit flow conveys information about the values of the variables in the `while` statment guard as well as the variables in the other guards that could affect whether that `while` statement is reached. To characterize this implicit flow to a statement ℓ in a program S, we form the set $\Delta_S(\ell)$ containing those guards that affect whether 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:

$$\Delta_S(\ell): \quad \Theta_S(\ell) \;\cup\; \bigcup_{\ell_W \in \mathcal{W}_\ell} (\{G_W\} \cup \Delta_S(\ell_W)) \tag{10.7}$$

Beware that for a statement ℓ in the body of a `while` statement, subset $\mathcal{W}_\ell$ used for (10.7) 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 account for illicit flows caused by guards affecting a `while` statement that does not have assignment statement ℓ in its body by replacing $\Theta_S(\ell)$ with $\Delta_S(\ell)$ in Θ_S-Safe Assignment Statements condition (9.26).

Δ_S-Safe Assignment Statements. Ensure that

$$\left(\Gamma_{\mathcal{E}}(expr) \;\sqcup\; \left(\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G)\right)\right) \;\sqsubseteq\; \Gamma(w) \tag{10.8}$$

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

Δ_S-Safe Assignment Statements condition (10.8) is more-stringent than Θ_S-Safe Assignment Statements condition (9.26), since $\Theta_S(\ell) \subseteq \Delta_S(\ell)$ holds according to definition (10.7) of $\Delta_S(\ell)$. This should not be surprising. TINI ignores non-terminating executions; PSNI does not. The additional guards in $\Delta_S(\ell)$ are those guards that affect whether ℓ cannot be reached due to non-termination, causing illicit flows ignored by TINI but not by PSNI.

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 $\Delta_S(\ell)$ becomes 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:[3] $\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$ □

So in programs satisfying restrictions (i) and (ii), Δ_S-Safe Assignment Statements condition (10.8) can be discharged by checking Θ_S-Safe Assignment Statements condition (9.26). The former is undecidable but the latter can be discharged using a straightforward static analysis of S.

Extending IMP to Relax Guard Restrictions. Guard Restrictions for PSNI requires that all iteration be controlled by variables with label $\bot_\Lambda$.[4] Some relaxation of the restrictions are possible, though, for looping that is guaranteed to terminate. Only loops that sometimes terminate can cause implicit flows.

[3] Here is that derivation. From definition (10.7) for $\Delta_S(\ell)$ we get:

$$\bigsqcup_{G \in \Delta_S(\ell)} \Gamma_{\mathcal{E}}(G) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \sqcup \bigsqcup_{\substack{\ell_W \in \mathcal{W}_\ell \\ G \in (\Delta_S(\ell_W) \cup \{G_W\})}} \Gamma_{\mathcal{E}}(G) \tag{10.9}$$

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 (10.9) (with $\mathcal{W}$ instantiated by $\mathcal{W}_\ell$) we get

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

[4] Compliance with this restriction on iteration also is necessary if we want to ensure that delays caused by execution of `while` statements reveal information only about the values of public variables. So if we are concerned with preventing side-channel attacks (as discussed in §12.2) then compliance with this restriction is going to be required, anyway.

A static analysis cannot determine whether a `while` statement will always terminate, due to the undecidability of the halting problem. However, a static analysis can identify iteration that always terminates if the statements for defining loops in the programming language are restricted. The `for`-loop statement

$$\texttt{for}\ v := \mathit{expr}\ \textbf{to}\ \mathit{expr}'\ \texttt{do}\ T\ \texttt{end} \tag{10.10}$$

would be an example of such a statement provided it is considered syntactically correct only if: (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 (10.10), body T is executed between zero and some bounded number of times. The guard for `for`-loop statement statement (10.10) is defined to be $\mathit{expr} \le v \le \mathit{expr}'$, since this predicate holds each time body T starts executing. So for every statement ℓ in body T of a `for`-loop statement appearing in a program S, $\mathit{expr} \le v \le \mathit{expr}' \in \Theta_S(\ell)$ will hold.

10.1.2 Enforcing PSNI with Typing Rules

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 10.2.[5] Each of theses typing rules derives a judgement that is valid 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 valid and, due to (i) in Valid Judgements for PSNI, the program will satisfy PSNI.

The typing rules in Figure 10.2 resemble the TINI typing rules in Figure 9.8, but with rule WHILE modified to ensure that every `while` statement ℓ_W in a type-correct program satisfies the requirements of Guard Restrictions for PSNI (page 280). In particular, the use of $\bot_\Lambda$ for the control context in the hypothesis

[5] Here is a typing rule in case IMP is being extended with `for`-loop statements.

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

where $\mathit{tgts}(S)$ denotes the set of variables that are the targets of assignment statements in S.

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

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

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

Figure 10.2: Typing rules for PSNI compliance

and conclusion of rule WHILE forces $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ to hold for all $G \in \Theta_S(\ell_W)$. Since every type-correct `while` statement ℓ_W is being restricted in this way, we can conclude that $\Gamma_{\mathcal{E}}(G) = \bot_\Lambda$ holds for all $G \in \Delta_S(\ell_W)$. Rule ASSIGN, which ensures that Θ_S-Safe Assignment Statements condition (9.26) holds for an assignment statement, then also ensures that Δ_S-Safe Assignment Statements condition (10.8) holds for that assignment statement.

10.1.3 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, 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.

Detecting that execution is blocked can result in an illicit implicit flow, as the following IMP program illustrates if $0 \le x_{\mathsf{H}}$, $\Gamma(i_{\mathsf{L}}) = \mathsf{L}$, $\Gamma(x_{\mathsf{H}}) = \mathsf{H}$, and $\mathsf{H} \not\sqsubseteq \mathsf{L}$ 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
```
(10.11)

Assignment statement ℓ does not comply with Δ_S-Safe Assignment Statements condition (10.8), because ℓ appears in the body of an `if` statement with guard $x_{\mathsf{H}} = i_{\mathsf{L}}$ and $\Gamma_{\mathcal{E}}(x_{\mathsf{H}} = i_{\mathsf{L}}) \not\sqsubseteq \Gamma(x_{\mathsf{L}})$ holds. Moreover, a dynamic enforcement mechanism that blocks execution upon reaching ℓ would not prevent an L-observer from deducing the initial value of x_{H} by monitoring i_{L}—the last value of i_{L} that

the L-observer reads before detecting that execution has blocked is the initial value of x_{H}.

We conclude that a dynamic enforcement mechanism for PSNI must not only prevent illicit explicit and implicit flows caused by assignment statements. It must also prevent illicit *detection flows*, wherein detecting that an execution has blocked allows a λ-observer to learn information about the initial value of a variable in $V_{\not\sqsubseteq\lambda}$ based on the values assigned to 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) $\Gamma(v) \not\sqsubseteq \lambda$ holds for some variable $v \in \mathit{Vars}(G)$ and some guard $G \in \Delta_S(\ell)$ that was evaluated prior to execution becoming blocked. For example, program (10.11) 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, then, would be sufficient to ensure that a reference monitor enforces PSNI without causing illicit detection flows.

(i) To avoid illicit flows from an assignment statement ℓ: $w := \mathit{expr}$, execution of the monitored program must be blocked before reaching ℓ if ℓ does not satisfy Δ_S-Safe Assignment Statements condition (10.8).

(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 at a control point that was reached before a problematic statement is executed. Even given this flexibility to block execution early, requirement (iii) presents an implementation challenge, since 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 to the implementation challenge for requirement (iii) 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 cause execution to be blocked for satisfying 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

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

$\mathcal{R}_{PS} \rhd S$ satisfies Guard Restrictions for PSNI (page 280) and, therefore Δ_S-Safe Assignment Statements condition (10.8) 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 10.3.

By design, any program that reference monitor $\mathcal{R}_{PS}$ executes without blocking is type-correct according to the the typing rules in Figure 10.2. A type-correct program might be blocked by $\mathcal{R}_{PS}$, though. As an illustration, if $\Gamma(x_{\mathsf{H}}) \neq \bot_{\Lambda}$ holds then $\mathcal{R}_{PS}$ would block execution by

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{H}} := x_{\mathsf{H}} + 1 \texttt{ else skip fi} \tag{10.12}$$

even though this program is type-correct. This establishes that $\mathcal{R}_{PS}$ is less permissive than type-checking is for PSNI enforcement.

Moreover, no reference monitor can be more permissive than $\mathcal{R}_{PS}$, because (by definition) the statements in the body of a control flow statement cannot be known by any reference monitor when that control flow statement is reached. So, for example, if $x_{\mathsf{H}} = 0$ holds in initial states, then (by definition) a reference monitor cannot distinguish between an execution of (10.12), a program that satisfies PSNI, and an execution of

```
if x_H = 0 then x_H := x_H + 1
           else while true do skip end
fi
```

a program that violates PSNI. For enforcing PSNI, the improved permissiveness that is the goal of dynamic enforcement is not achieved by using a reference monitor.

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. Results from the static analyzer enable reference monitor actions to take into account code that could be executed in the future and code that was an alternative to code had been executed.

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 *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 by $\mathcal{R}_{HPS}$.

Figure 10.4 details the actions comprising $\mathcal{R}_{HPS}$. Execution of control-flow statements are blocked by $\mathcal{R}_{HPS}$, as follows.

- A `while` statement with guard G_W is blocked if $\Gamma_{\mathcal{E}}(G_W) \neq \bot_\Lambda$ holds.
- An `if` statement with guard G_{IF} and body T is blocked if $\Gamma_{\mathcal{E}}(G_{\text{IF}}) \neq \bot_\Lambda$ holds and T contains a `while` statement that $\mathcal{R}_{HPS}$ would block or T contains an assignment statementthat $\mathcal{R}_{HPS}$ would block.

This blocking prevents illicit detection flows, because a blocked execution reveals no information about results obtained by evaluating any guard G that satisfies $\Gamma_{\mathcal{E}}(G) \neq \bot_\Lambda$. In addition, $\mathcal{R}_{HPS}$ ensures compliance with Guard Restrictions for PSNI, which allows Δ_S-Safe Assignment Statements condition (10.8) to be checked for an assignment statement ℓ by using a stack where `top()` satisfies (9.30). $\mathcal{R}_{HPS}$ implements such a stack with 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, because the guard label will be $\bot_\Lambda$, so (9.30) 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 $\mathcal{R}_{HPS}$. For example, $\mathcal{R}_{HPS}$ does not block execution of (10.12) which, as we observed earlier, $\mathcal{R}_{PS}$ blocks. So $\mathcal{R}_{HPS}$ is more permissive than $\mathcal{R}_{PS}$. Moreover, since any static analysis can be formulated as a type system, 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 would violate PSNI but cannot be reached. No static analyzer can perform such an analysis, since it requires solving an undecidable problem.

10.2 Flow-Sensitive Noninterference

We now consider programs in which some variables have labels that vary. A *fixed variable* has the same fixed label throughout all executions; a *flexible variable*,

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

which we distinguish with tilde-topped names (e.g., $\widetilde{y}$), has a label that can be changed during an execution. Changing the label of a flexible variable allows that variable to be used for different purposes during an execution.

To specify label assignments for both fixed and flexible variables, program states come with a function-valued variable $\widehat{\Gamma}(\cdot)$. The label that a program state s gives to a fixed variable v is $s.\widehat{\Gamma}(v)$, the label s gives a flexible variable $\widetilde{v}$ is $s.\widehat{\Gamma}(\widetilde{v})$, and the label s gives to an expression E is $s.\widehat{\Gamma}_{\mathcal{E}}(E)$ defined by

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

We write just $\widehat{\Gamma}(v)$ or $\widehat{\Gamma}_{\mathcal{E}}(E)$ when the state is clear from the context or when discussing fixed variables (since they have the same label in all states).

For defining flow-sensitive noninterference (F-PSNI), we generalize state projections $s|_{\mathcal{W}}$ and then use $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$. Definition (10.4) for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$, employs state projections $s|_{\mathcal{W}}$ for comparing values of a fixed set $\mathcal{W}$ of variables in corresponding program states in certain pairs of executions. For F-PSNI, the variables that are compared depends on their label assignments in the states being compared. We materialize that dependence by using state projections $s|_{F(\cdot)}$, where $F(\cdot)$ is a function giving the set $F(s)$ of variables to use for constructing the state projection. The value of an anchored or flexible variable v in a state projection $s|_{F(\cdot)}$ is thus defined by:

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

Then, to define F-PSNI, instead of using sets $V_{\sqsubseteq\lambda}$ and $V_{\not\sqsubseteq\lambda}$ for the state projections in PSNI formal definition (10.4), we use the following functions $\widehat{V}_{\sqsubseteq\lambda}(\cdot)$ and $\widehat{V}_{\not\sqsubseteq\lambda}(\cdot)$ from program states to sets of fixed and flexible variables:

$$\widehat{V}_{\sqsubseteq\lambda}(s)\colon\ \{v \in \mathit{Vars}(S) \mid s.\widehat{\Gamma}(v) \sqsubseteq \lambda\} \qquad \widehat{V}_{\not\sqsubseteq\lambda}(s)\colon\ \{v \in \mathit{Vars}(S) \mid s.\widehat{\Gamma}(v) \not\sqsubseteq \lambda\}$$

We get a definition for F-PSNI by combining these elements:

> **F-PSNI.** For a deterministic program S having states that assign values to fixed variables, to flexible variables, and to the function $\widehat{\Gamma}$ mapping variable names to labels from a set Λ with a partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\colon\ \widehat{V}_{\not\sqsubseteq\lambda}(\cdot) \overset{S}{\not\rightarrow}_{\mathsf{ps}} \widehat{V}_{\sqsubseteq\lambda}(\cdot))$$ □

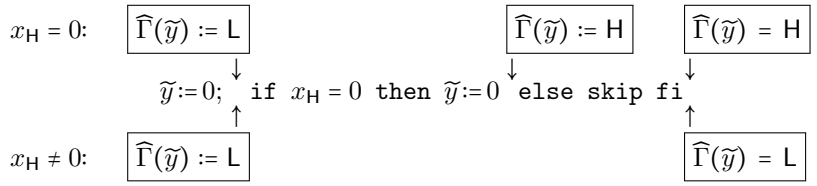
10.2.1 Flexible Variables and Enforcement

We might expect that an update $\ell\colon \widetilde{y} := \mathit{expr}$ (say) to a flexible variable $\widetilde{y}$ would not violate F-PSNI if executing ℓ also gives $\widetilde{y}$ a new label:

$$\widehat{\Gamma}(\widetilde{y}) \ :=\ \widehat{\Gamma}_{\mathcal{E}}(\mathit{expr}) \ \sqcup \bigsqcup_{G\in\Delta_S(\ell)} \widehat{\Gamma}_{\mathcal{E}}(G). \tag{10.13}$$

However, as we shall see, such a label update can cause an F-PSNI violation.

The problem can be seen in a simple IMP program, where the set of labels is Λ_{LH} and $\widehat{\Gamma}(x_{\mathsf{H}}) = \mathsf{H}$ holds. Two executions of such a program are depicted in the following diagram—one execution where $x_{\mathsf{H}} = 0$ holds initially and the other where it doesn't.



(10.14)

Both executions terminate with the same value for $\widetilde{y}$ but different labels. Letting s and s' denote those two final states, $s|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)} = s'|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)}$ does not hold, since only one of $s|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)}$ and $s'|_{\widetilde{V}_{\sqsubseteq \mathsf{L}}(\cdot)}$ gives a value to $\widetilde{y}$. So, by definition, F-PSNI does not to hold. The value of guard $x_{\mathsf{H}} = 0$ is being leaked through the different label assignments.

A straightforward way to avoid such leaks is to associate a single label assignment Γ_ℓ with each control point ℓ by adjusting the exit control points for control-flow statements, as follows.

> **Preventing Variation in Label Assignments.** For each flexible variable $\widetilde{x}$, give $\widetilde{x}$ a more restrictive label if needed to eliminate variation in label $\widehat{\Gamma}(\widetilde{x})$ when the exit control point ℓ for a control-flow statement is reached. □

In (10.14), for example, variation in the label for $\widetilde{y}$ at the exit control point for the `if` statement would be eliminated by changing that label from L to H before or after the `skip`. Executions from any initial values of x_{H} then would terminate satisfying $\widehat{\Gamma}(\widetilde{y}) = \mathsf{H}$ (without also granting L-observers additional access).

Leaks also can be avoided if there are different label assignments when the exit control point for a control-flow statement is reached provided the following holds.[6]

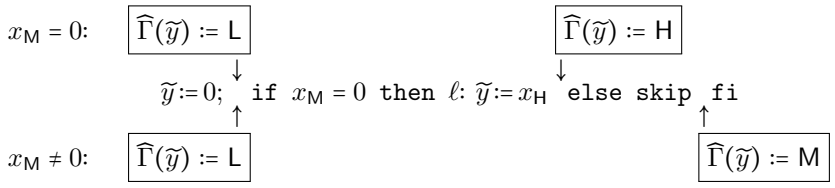
> **Matching Outcomes to Guards.** If the body of a control-flow statement S with guard G_S contains an assignment statement to a flexible variable $\widetilde{x}$ then make label $\widehat{\Gamma}(\widetilde{x})$ more restrictive, if necessary, to ensure that $\widehat{\Gamma}_{\mathcal{E}}(G_S) \sqsubseteq \widehat{\Gamma}(\widetilde{x})$ holds when the exit control point for S is reached. □

[6] To see that Matching Outcomes to Guards stops a label $\widehat{\Gamma}(\widetilde{y})$ from leaking the value of guard G_S to a λ-observer, we analyze scenerios in which $\widehat{\Gamma}_{\mathcal{E}}(G_S) \not\sqsubseteq \lambda$ holds and, therefore, learning G_S is a leak. Consider two executions of a control-flow statement S, where guard G_S was evaluated to different values and the executions terminated in states giving different labels λ_1 and λ_2 to $\widetilde{y}$. For a λ-observer to detect a difference in these labels then either $\lambda_1 \sqsubseteq \lambda$ and $\lambda_2 \not\sqsubseteq \lambda$ must hold or $\lambda_2 \sqsubseteq \lambda$ and $\lambda_1 \not\sqsubseteq \lambda$ must hold. Without loss of generality, we consider the former and show that it is not possible because it results in a contradiction. Matching Outcomes to Guards requires that $\widehat{\Gamma}_{\mathcal{E}}(G_S) \sqsubseteq \lambda_1$ holds. So, by transitivity, we conclude $\widehat{\Gamma}_{\mathcal{E}}(G_S) \sqsubseteq \lambda$, but that contradicts the earlier assumption that $\widehat{\Gamma}_{\mathcal{E}}(G_S) \not\sqsubseteq \lambda$ holds.

Note that in implementing Matching Outcomes to Guards, an execution path that itself has no assignment statement to a flexible variable $\widetilde{y}$ (say) might nevertheless require an update to label $\widehat{\Gamma}(\widetilde{y})$ if alternative execution paths by that control-flow statement do have assignment statements where $\widetilde{y}$ is the target. For IMP programs, this implies:

- An `if` statement's `then` or `else` alternative might have to update the label of a flexible variable because that variable is a target in the other alternative of that `if` statement.
- A `while` statement S that does not execute its body (because its guard was initially *false*) might still have to update the label of a flexible variable that is a target in the body of S.

To illustrate an application of Matching Outcomes to Guards, 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} in this program satisfy $\widehat{\Gamma}(x_{\mathsf{M}}) = \mathsf{M}$ and $\widehat{\Gamma}(x_{\mathsf{H}}) = \mathsf{H}$. Two executions are depicted.



(10.15)

The updates to label $\widehat{\Gamma}(\widetilde{y})$ inserted after each assignment statement in (10.15) are to comply with (10.13); the update after the `skip` statement is to comply with Matching Outcomes to Guards. To see that the update after the `skip` is necessary, observe that guard $x_{\mathsf{M}} = 0$ affects label $\widehat{\Gamma}(\widetilde{y})$ by affecting whether assignment statement ℓ is executed. So $\widehat{\Gamma}_{\mathcal{E}}(x_{\mathsf{M}} = 0) \sqsubseteq \widehat{\Gamma}(\widetilde{y})$ needs to hold when the `if` statement terminates—whether the `then` alternative or the the `else` alternative has been taken.

Another way to justify the update to $\widehat{\Gamma}(\widetilde{y})$ after the `skip` in (10.15) is by analyzing what a λ-observer can learn when the `fi` is reached. The table to the right lists that $\widehat{\Gamma}(\widetilde{y})$ is either M or H at that control point. Consequently (as listed in the L column of the table), an L-observer cannot read $\widetilde{y}$ no matter what value guard $x_{\mathsf{M}} = 0$ had. We conclude that an L-observer can learn nothing about the guard evaluation from the different labels for $\widetilde{y}$ when execution reaches `fi`. But an M-observer can detect whether $x_{\mathsf{M}} = 0$ evaluated to *true* by attempting to read $\widetilde{y}$, since $\widehat{\Gamma}(\widetilde{y}) = \mathsf{M}$ holds if guard $x_{\mathsf{M}} = 0$ evaluated to *false* but $\widehat{\Gamma}(\widetilde{y}) = \mathsf{H}$ holds if the guard evaluated to *true*. This difference does not cause a leak, though, because $\widehat{\Gamma}_{\mathcal{E}}(x_{\mathsf{M}} = 0) = \mathsf{M}$ holds, so an

state at `if`	$\widehat{\Gamma}(\widetilde{y})$ at `fi`	λ-observer read $\widetilde{y}$? L	M	H
$x_{\mathsf{M}} = 0$	H	no	no	yes
$x_{\mathsf{M}} \neq 0$	M	no	yes	yes

M-observer is allowed to learn the value of guard $x_{\mathsf{M}} = 0$. 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 read guard $x_{\mathsf{M}} = 0$.

10.2.2 Enforcing F-PSNI with Typing Rules

Typing rules can ensure compliance with Preventing Variation in Label Assignments. In that case, label assignments for the control points in a program S could be conveyed by a *label annotated program*—the program text for S along with a label assignment Γ_ℓ for each control point ℓ in S.[7]

One way to represent a label annotated program is by using the notation $S\,/\,\widehat{\Gamma}^*$ where S identifies a program and $\widehat{\Gamma}^*$ is a function that maps each control point ℓ in S (and perhaps other control points) to a label assignment. So upon reaching control point ℓ, function-valued variable $\widehat{\Gamma}(\cdot)$ would be set to $\widehat{\Gamma}^*(\ell)$.

Another representation is to insert a *label annotation* at each control point in the program text. The label annotation "$\{\Gamma_\ell\}$" inserted at a control point ℓ indicates that function-valued variable $\widehat{\Gamma}(\cdot)$ be set to Γ_ℓ 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 IMP assignment or `skip` statements and the Γ_i are label assignments.[8]

$\{\Gamma_1\}\ \ell_1\text{: } S_1\ \{\Gamma_2\}$
$\ell_2\text{: }$ `if` G `then` $\{\Gamma_3\}\ \ell_3\text{: } S_2; \{\Gamma_4\}$
`else` $\{\Gamma_5\}\ \ell_4\text{: } S_3\ \{\Gamma_6\}$
`fi` $\{\Gamma_7\}$

It sometimes is helpful to combine the two kinds of representations for label annotated programs, as follows.

$\{\Gamma\}\ S\,/\,\widehat{\Gamma}^*$: label assignment Γ is associated with the entry control point for S, and the label assignments given by $\widehat{\Gamma}^*$ are used for all other control points in S.

$S\,/\,\widehat{\Gamma}^*\ \{\Gamma'\}$: label assignment Γ' is associated with the exit control point for S, and the label assignments given by $\widehat{\Gamma}^*$ are used for all other control points in S.

[7] Flexible variables might have to be given more restrictive labels than required by Matching Outcomes to Guards. So there will be programs that satisfy F-PSNI but are not type-correct. Program (10.15) is an example. The label update $\widehat{\Gamma}(\widetilde{y}) := \mathsf{M}$ performed after the `skip` statement is less restrictive from the label update $\widehat{\Gamma}(\widetilde{y}) := \mathsf{H}$ that would be required by Preventing Variation in Label Assignments.

[8] The placement of label annotation "$\{\Gamma_2\}$" 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 `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.

$\{\Gamma\}\ S/\widehat{\Gamma}^*\ \{\Gamma'\}$: label assignment Γ is associated with the entry control point for S, label assignment Γ' is associated with the exit control point for S, and the label assignments given by $\widehat{\Gamma}^*$ are used for all other control points in S.

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

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

A label annotated program $S/\widehat{\Gamma}^*$ is considered type-correct if the typing rules in Figure 10.5 can derive the judgement $\bot_\Lambda \vdash_{\widehat{\mathsf{pss}}} S/\widehat{\Gamma}^*$. By design, those typing rules only derive valid judgements from valid hypotheses. So a type-correct label annotated program $S/\widehat{\Gamma}^*$ will satisfy F-PSNI, due to (i) in the definition validity for $\bot_\Lambda \vdash_{\widehat{\mathsf{pss}}} S/\widehat{\Gamma}^*$.

The typing rules for control-flow statements are the key to having a label annotated program be the conclusion of each typing rule, thereby ensuring that different executions reaching the same control point give the same labels to the flexible variables. Different label assignments arise when a given control point is reached only if different execution paths are taken. That requires executing a control-flow statement. But rule IF mandates the same label assignment for the exit control point of the `then` alternative as for the `else` alternative, and rule WHILE mandates the same label assignment before each evaluation of the `while` statement guard. So in type-correct programs, different label assignments cannot arise when a given control point is reached.

To enforce F-PSNI, the typing rules also must reject programs that would allow λ-observers to distinguish executions starting from initial values that differ only in the values of anchored or flexible variables v satisfying $\widehat{\Gamma}(v) \not\sqsubseteq \lambda$. The PSNI typing rules in Figure 10.2 achieve this for programs that only have fixed variables. The F-PSNI typing rules in Figure 10.5 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 include label annotations giving the mandates discussed above regarding label updates due to control-flow statements, and rule F-ASSIGN has been added for characterizing the label assignment produced by assignment statements that have flexible variables as targets.

Figure 10.6 illustrates the use of the F-PSNI typing rules, giving a derivation to show that program (10.14) is type-correct and, therefore, satisfies F-PSNI.[9]

[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 $\mathsf{L} \vdash_{\widehat{\mathsf{pss}}} S/\widehat{\Gamma}^*$.

$$\textsc{skip:}\ \frac{}{\gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ \texttt{skip}\ \{\Gamma\}} \qquad \textsc{seq:}\ \frac{\gamma \vdash_{\widetilde{\mathsf{pS}}} S_1 / \widehat{\Gamma}_1^*\ \{\Gamma\},\ \ \gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S_2 / \widehat{\Gamma}_2^*}{\gamma \vdash_{\widetilde{\mathsf{pS}}} S_1 / \widehat{\Gamma}_1^*; \{\Gamma\}\ S_2 / \widehat{\Gamma}_2^*}$$

$$\textsc{a-assign:}\ \frac{\gamma \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)}{\gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ x := expr\ \{\Gamma\}} \qquad \textsc{f-assign:}\ \frac{}{\gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ \widetilde{v} := expr\ \{\Gamma[\widetilde{v} \leftarrowtail \gamma \sqcup \Gamma_{\mathcal{E}}(expr)]\}}$$

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

$$\textsc{while:}\ \frac{\Gamma_{\mathcal{E}}(expr) = \bot_\Lambda,\ \ \bot_\Lambda \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S / \widehat{\Gamma}^*\ \{\Gamma\}}{\bot_\Lambda \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ \texttt{while}\ expr\ \texttt{do}\ \{\Gamma\}\ S / \widehat{\Gamma}^*\ \{\Gamma\}\ \texttt{end}\ \{\Gamma\}}$$

$$\textsc{relab:}\ \frac{\gamma' \sqsubseteq \gamma,\ \ \Gamma_1' \sqsubseteq \Gamma_1,\ \ \Gamma_2 \sqsubseteq \Gamma_2',\ \ \gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma_1\}\ S / \widehat{\Gamma}^*\ \{\Gamma_2\}}{\gamma' \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma_1'\}\ S / \widehat{\Gamma}^*\ \{\Gamma_2'\}}$$

Notation:

$\Gamma[\widetilde{x} \leftarrowtail \Gamma_{\mathcal{E}}(expr)]$: Γ, except that $\widetilde{x}$ has label $\Gamma_{\mathcal{E}}(expr)$

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

Figure 10.5: Typing rules for F-PSNI compliance

Steps 4 and 5 derive judgements for the `then` and `else` alternatives. These judgements have exit control points that mandate different labels for flexible variable $\widetilde{y}$, necessitating step 7 to change label $\widehat{\Gamma}(\widetilde{y})$ 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 can be discharged using the judgements in steps 4 and 7.

When using the F-PSNI typing rules to derive a label annotated program for an `if` statement, it is not unusual to start by deriving judgements for the `then` and `else` alternatives, but with different label assignments mandated for their exit control points:

$$\gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S_1 / \widehat{\Gamma}_1^*\ \{\Gamma_1\} \qquad \gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S_2 / \widehat{\Gamma}_2^*\ \{\Gamma_2\}$$

Because labels are elements of a lattice, there will always exist a a label assignment Γ_3 where $\Gamma_3(\widetilde{x})$ is $\Gamma_1(\widetilde{x}) \sqcup \Gamma_2(\widetilde{x})$ for each flexible variable $\widetilde{x}$ (and $\Gamma_3(x)$ is $\Gamma_1(x)$ for each fixed variable x). By construction, Γ_3 satisfies $\Gamma_1 \sqsubseteq \Gamma_3$ and $\Gamma_2 \sqsubseteq \Gamma_3$. Rule RELAB then can be used to derive

$$\gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S_1 / \widehat{\Gamma}_1^*\ \{\Gamma_3\} \qquad \gamma \vdash_{\widetilde{\mathsf{pS}}} \{\Gamma\}\ S_2 / \widehat{\Gamma}_2^*\ \{\Gamma_3\}$$

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

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

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

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

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

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

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

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

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

Figure 10.6: Example F-PSNI derivation

which are the hypotheses needed for rule IF.

10.2.3 Dynamic Enforcement of F-PSNI

A reference monitor alone cannot ensure compliance with Preventing Variation in Label Assignments or with Matching Outcomes to Guards. This is because reference monitor actions based on untaken execution paths would be necessary for that compliance. For example, with the programs in (10.14) and (10.15), an `if` statment executes either the `then` alternative or the `else` alternative but the label updates required when that `if` statment terminates depend on the assignment statement targets in both the `then` and the `else` alternatives.

With hybrid enforcement mechanisms, a reference monitor is augmented with the capability to analyze code that has not yet been executed. Seeking greater permissiveness than allowed by Preventing Variation in Label Assignments (required for type-correctness), we develop a hybrid enforcement mechanism for compliance with Matching Outcomes to Guards. It uses an analyzer $\mathit{ftgts}(S)$ that provides the set of flexible variables that are targets of assignment statements in a code fragment S. Compliance with Matching Outcomes to Guards is then maintained by executing

$$\textbf{forall } \widetilde{x} \in \mathit{ftgts}(S) \texttt{ do } \widehat{\Gamma}(\widetilde{x}) := \widehat{\Gamma}(\widetilde{x}) \sqcup \widehat{\Gamma}_{\mathcal{E}}(G_S) \texttt{ end}$$

whenever execution reaches the end of a control-flow statement S with guard G_S.

A dynamic enforcement mechanism to implement these label adjustments is given in Figure 10.7. It employs a stack sf containing pairs $\langle V, \lambda_G \rangle$, where V is a set of flexible variables with labels to be adjusted and λ_G is the label to use for the adjustment. To enforce F-PSNI, this mechanism would be combined with $\mathcal{R}_{HPS}$ in Figure 10.4. Or, it could be combined with Reference monitor $\mathcal{R}_{TI}$ in Figure 9.12 to enforce TINI in programs that have fixed and flexible variables.

10.3 *Other Noninterference Policies

Noninterference policies have the form

$$(\forall s, s' \in \mathit{Init}_S\colon\ s \sim_\eta s' \ \Rightarrow\ [\![S]\!]^\eta(s) \approx_\eta [\![S]\!]^\eta(s')) \tag{10.16}$$

where predicate $s \sim_\eta s'$ is satisfied by initial states s and s' that are indistinguishable to an attacker, function $[\![S]\!]^\eta(s)$ evaluates to a description of the execution effects produced when S is started in state s, and predicate $\xi \approx_\eta \xi'$ is satisfied if ξ and ξ' are descriptions of execution effects that would be indistinguishable to attackers. Different choices for predicate $\sim_\eta$, function $[\![S]\!]^\eta(\cdot)$, and predicate $\approx_\eta$ result in the various different noninterference policies discussed in this chapter as well as many others.

The execution effects that $[\![S]\!]^\eta(\cdot)$ captures reflect the assumptions we are making about what can be monitored by attackers. For TINI, we assumed

upon S reaching $\bullet$	action to be performed
$\bullet$ `if` G `then` S'`else` S'' `fi`	`push`$(sf,\ \langle ftgts(S') \cup ftgts(S''),\ \widehat{\Gamma}_{\mathcal{E}}(G)\rangle\ )$
... `fi` $\bullet$	$\langle \widetilde{V},\ \lambda_G\rangle$:= `top`(sf) **forall** $\widetilde{x} \in \widetilde{V}$ `do` $\widehat{\Gamma}(\widetilde{x}) := \widehat{\Gamma}(\widetilde{x}) \sqcup \lambda_G$ `end`; `pop`(sf)
$\bullet$ `while` G `do` S `end`	`push`$(sf,\ \langle ftgts(S),\ \widehat{\Gamma}_{\mathcal{E}}(G)\rangle\ )$
`while` G `do` S `end` $\bullet$	$\langle \widetilde{V},\ \lambda_G\rangle$:= `top`(sf) **forall** $\widetilde{x} \in \widetilde{V}$ `do` $\widehat{\Gamma}(\widetilde{x}) := \widehat{\Gamma}(\widetilde{x}) \sqcup \lambda_G$ `end`; `pop`(sf)

Figure 10.7: Compliance for matching outcomes to guards

that attackers can monitor only the final states of terminating executions, so $[\![S]\!]^{\eta}(\cdot)$ is a function (*viz.* $[\![S]\!](\cdot)$) from initial states to final states; for PSNI, we assumed that an attacker can monitor intermediate states of all executions, so $[\![S]\!]^{\eta}(\cdot)$ is a function (*viz.* $[\![S]\!]^{\mathsf{tr}}(\cdot)$) from initial states to execution traces.

But thus far we have considered only deterministic programs. Nondeterministic programs and concurrent programs can be handled, too. We use a function $[\![S]\!]^{\eta}(\cdot)$ that evaluates to sets of execution effects.

- With a nondeterministic program S, each element in set $[\![S]\!]^{\eta}(s)$ would represent 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 set $[\![S]\!]^{\eta}(s)$ would represent 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.

The straightfoward definition for relation $\mathcal{X} \approx_{\eta} \mathcal{X}'$ in (10.16) is $\mathcal{X} = \mathcal{X}'$. However, in situations where an attacker can account for differences due to nondeterminacy or scheduling choices, a weaker definition is more suitable:

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

With this weaker definition, $[\![S]\!]^{\eta}(s) \approx_{\eta} [\![S]\!]^{\eta}(s')$ holds if an attacker observing the execution effects from an initial state s cannot rule out having witnessed the execution effects from a different initial state s'. Noninterference policies where $\approx_{\eta}$ 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 by running multiple experiments. Such a set of experiments could allow the attacker to approximate the likelihood of an initial state given how often certain execution effects are produced.[10] For those settings, we would want to use a version of noninterference where $[\![S]\!]^\eta(\cdot)$ produces a probability distribution for possible execution effects and $[\![S]\!]^\eta(s) \approx_\eta [\![S]\!]^\eta(s')$ holds if the probability distributions $[\![S]\!]^\eta(s)$ and $[\![S]\!]^\eta(s')$ are indistinguishable to attackers.

Finally, we should acknowledge that an attacker must monitor some system interface to learn about execution effects. The noninterference policies discussed in this chapter avoided the details of those interfaces by using states to describe the information that is available to observers. The domain and range for those states reflect assumptions about what execution effects an attacker is and is not able to monitor. For example, TINI and PSNI are defined in terms of states that map variables to values and labels, because these policies assume that attackers are limited to reading (certain) variables.

Changes to variables are not the only observable execution effects, though. The passage of time is an execution effect, and measuring the time that elapses 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.

Notes and Reading for Chapter 10

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 programming languages that had nondeterminacy and concurrency.

The first step was new typing rules in Volpano and Smith [18] for enforcing (what later became known as) termination-sensitive noninterference (TSNI). With TSNI, attackers are assumed able to distinguish between terminating and

[10] As an example, consider the nondeterministic program

$$S\colon\ \texttt{if}\ x_{\mathsf{H}} > 0\ \texttt{then}\ [x_{\mathsf{L}} := 1\ \blacksquare_{.99}\ x_{\mathsf{L}} := 2]\ \texttt{else}\ [x_{\mathsf{L}} := 1\ \blacksquare_{.5}\ x_{\mathsf{L}} := 2]\ \texttt{fi}$$

where $[S\ \blacksquare_p\ S']$ is the syntax for a statement that executes S or S', choosing S with probability $1 - p$ and choosing S' with probablilty 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.

nontermination executions. Therefore, TSNI defends against more-capable attackers than TINI, since TINI ignores nonterminating executions.

Another significant step towards a more realistic model for 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 *progress insensitive noninterference* (PINI) and *progress sensitive noninterference* (PSNI) by Askarov and Myers [3]. 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 10.2 resemble the typing rules in Figure 9.8. Finally, Askarov et al. [2] debunks a widely-held belief that only a single secret bit would be leaked when 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. Most type systems had employed flow-insensitive analyses. So the program $x_{\mathsf{L}} := x_{\mathsf{H}}$; $x_{\mathsf{L}} := 63$, which satisfies TINI, would be rejected because it contains a subprogram (*viz.* $x_{\mathsf{L}} := x_{\mathsf{H}}$) that—in isolation—violates TINI. Hunt and Sands [7] gives typing rules that implement a flow-sensitive analysis, thereby avoiding such specious rejections.[11] Those typing rules (the basis for the typing rules in Figure 10.5) associate assertions with control points; each assertion gives a label assignment. 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] did not. Moreover, Hunt and Sands [7] observes that the label assignments generated by their typing rules is an abstract characterization of dependencies, leading to the surprising result: 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 if assignments statements to fresh variables can be added.

Dynamic enforcement mechanisms—reference monitors and hybrid enforcement mechanisms—were seen as another 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 a policy violation, the analyses they use, and whether they deliver increased permissiveness over a type system. Space does not permit giving a detailed enumeration here.

We might have expected that a dynamic enforcement mechanism would have to analyze code that will not be executed, since implicit flows can be caused

[11] 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.

by assignment statements in the untaken alternative of an `if` statement or 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 untaken code. 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 complex. 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 than the other. 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 the Sabelfeld and Myers survey [13] of language-based approaches and the 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 the authoritative treatment of declassification. Also, consider experimenting with a programming language like Jif [10, 9] or Flow Caml [16], where types enforce information flow control. Jif extends Java and has been used to build non-trivial applications. One such application that is well documented is the Civitas [4] coercion-resistant, universally and voter verifiable electronic voting system. Flow Caml 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] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow (software release), July 2001. http://www.cs.cornell.edu/jif.

[10] 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.

[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.

