# Chapter 5

# Information Flow Control: TINI

This chapter and the next discuss the specification and enforcement of *information flow policies*. Such policies specify whether the initial values of variables in certain classes may directly or indirectly affect the values of variables in other classes and/or may affect program termination. The variables in an information flow policy might correspond to regions of memory, communications channels, or files. For enforcing confidentiality, an information flow policy would specify that the values of a variable not be affected by secrets; for enforcing integrity, it would specify that these values not be affected by values derived from untrusted sources. Information flow policies are said to be *end-to-end* because they restrict what an initial value (or input) may affect and, therefore, these policies limit all uses of derived values. In contrast, the authorization policies discussed elsewhere in this book just restrict access to containers, independent of contents.

## 5.1 Labels Specifying Information Flow Policies

An information flow policy for a program (i) gives a *label assignment* $\Gamma(\cdot)$ that associates a label $\Gamma(v)$ with each program variable $v$ and (ii) gives a partial order[1] $\sqsubseteq$ on the set $\Lambda$ of possible labels, where $\Lambda$ contains a minimal element $\bot_\Lambda$ satisfying $\bot_\Lambda \sqsubseteq \lambda$ for all $\lambda \in \Lambda$. We write $\not\sqsubseteq$ to indicate the complement of $\sqsubseteq$, and

[1] A *relation* $\rho$ on a set *Vals* is a subset of $\{\langle a, b\rangle \mid a, b \in \mathit{Vals}\}$. Its *complement* $\not\rho$ is the set $\{\langle a, b\rangle \mid a, b \in \mathit{Vals}\} - \rho$. A *partial order* $\rho$ on *Vals* is a relation on *Vals* that satisfies the following properties, where (as is conventional) infix notation $a\,\rho\,b$ is used for $\langle a, b\rangle \in \rho$.

> Reflexive: $a\,\rho\,a$ for all $a \in \mathit{Vals}$.
> Transitive: $a\,\rho\,b$ and $b\,\rho\,c$ implies $a\,\rho\,c$ for all $a, b, c \in \mathit{Vals}$.
> Antisymmetric: $a\,\rho\,b$ and $b\,\rho\,a$ implies $a = b$ for all $a, b \in \mathit{Vals}$.

Note that a partial order $\rho$ does not have to relate all pairs of elements $a, b \in \mathit{Vals}$, so if $a \not\rho\ b$ holds it is possible that neither $a\,\rho\,b$ nor $b\,\rho\,a$ holds.



January 26, 2026 Copyright Fred B. Schneider All rights reserved.

we write $\lambda \sqsubset \lambda'$ as an abbreviation for $\lambda \sqsubset \lambda' \wedge \lambda \neq \lambda'$. Relation $\sqsubseteq$ specifies the allowed and prohibited information flows during program executions.

- $\Gamma(v) \sqsubseteq \Gamma(w)$ specifies that the value of variable $v$ is allowed to affect the value of variable $w$.
- $\Gamma(v) \not\sqsubseteq \Gamma(w)$ specifies that the value of variable $v$ is <u>not</u> allowed to affect the value of variable $w$.

So each label $\lambda \in \Lambda$ divides the set $Vars(S)$ of variables in a program $S$ into two subsets according to partial order $\sqsubseteq$:

$$V_{\sqsubseteq\lambda}\text{:}\quad \{v \in Vars(S) \mid \Gamma(v) \sqsubseteq \lambda\} \qquad V_{\not\sqsubseteq\lambda}\text{:}\quad \{v \in Vars(S) \mid \Gamma(v) \not\sqsubseteq \lambda\}$$

And in executions that comply with $\sqsubseteq$, the value of no variable from $V_{\not\sqsubseteq\lambda}$ is allowed to affect the value of any variable from $V_{\sqsubseteq\lambda}$. This is useful for prohibiting leaks. Ordinary access control cannot prevent a program from copying $x$ to $y$ if that program is both authorized to read a secret variable $x$ and to write a public variable $y$. An information flow policy can prevent such leaks if $R(\lambda)$ is the set of subjects that are authorized to read a variable with label $\lambda$ and the following holds.

$$(\forall \lambda, \lambda' \in \Lambda\text{:}\ \ \lambda \sqsubseteq \lambda' \Rightarrow R(\lambda') \subseteq R(\lambda)). \tag{5.1}$$

This is because $\Gamma(x) \sqsubseteq \Gamma(y)$ must hold for a subject to write $y$ after reading $x$, so from (5.1) we conclude that $R(\Gamma(y)) \subseteq R(\Gamma(x))$ must hold if a subject is copying $x$ to $y$. Therefore, subjects authorized to read $y$ must also be authorized to read $x$. If $x$ stores a secret then $R(\Gamma(x))$ includes only those subjects that are allowed to read that secret. Since $R(\Gamma(y))$ does not allow additional subjects to read $y$, we conclude that $y$ cannot be public.

### 5.1.1 Labels for Expressions

$\Gamma(\cdot)$ gives labels to variables, but not to expressions. The label $\Gamma_{\mathcal{E}}(E)$ that we associate with an expression $E$ (i) will specify the variables and expressions that $E$ is allowed to affect and (ii) will specify the variables and expressions that are allowed to affect $E$. Since unary operators and infix binary operators can be viewed as syntactic sugar for function applications, no generality is lost if we limit consideration here to expressions that are constants, variables, and function applications $f(E_1, E_2, \ldots, E_n)$ having arguments $E_i$ that are themselves expressions. For simplicity, assume that evaluating an expression always produces some value.

*Labels for Constants.* The label $\Gamma_{\mathcal{E}}(c)$ that we associate with a constant $c$ should not preclude an assignment statement from storing $c$ into any variable. Therefore, we require that $\Gamma_{\mathcal{E}}(c) \sqsubseteq \Gamma(v)$ hold for any constant $c$ and for any variable $v$. That requirement leads to the definition:

$$\Gamma_{\mathcal{E}}(c)\text{:}\ \ \bot_{\Lambda} \text{ for any constant } c \tag{5.2}$$

 Copyright Fred B. Schneider All rights reserved.

*Labels for Variables.* The label $\Gamma_{\mathcal{E}}(v)$ we associate with an expression that is a variable $v$ should have the same restrictions as $\Gamma(v)$:

$$\Gamma_{\mathcal{E}}(v)\text{: } \Gamma(v) \text{ for any variable } v. \tag{5.3}$$

*Labels for Function Applications.* Whether the value of $f(E_1, E_2, \ldots, E_n)$ is affected by the value of its argument $E_i$ depends on $f$. The conservative choice for label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ would be a label that works for any function $f$. Such a label would allow (but not require) each argument $E_i$ to affect the value of $f(E_1, E_2, \ldots E_n)$:

$$\Gamma_{\mathcal{E}}(E_i) \sqsubseteq \Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n)) \quad \text{for } 1 \le i \le n. \tag{5.4}$$

So satisfying (5.4) is one goal for our definition of $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$.

A value that is at least as large as any member of a set is called an *upper bound* for that set; a *least upper bound* is an upper bound that is not larger than any other upper bound. One way to satisfy (5.4) is by defining label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ to be an upper bound of set $\{\Gamma_{\mathcal{E}}(E_1), \Gamma(E_2), \ldots, \Gamma_{\mathcal{E}}(E_n)\}$ of labels. Among the upper bounds, the least upper bound is the best choice for label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ because it allows the value of $f(E_1, E_2, \ldots E_n)$ to affect more variables.

Least upper bounds for finite subsets $\{\lambda_1, \lambda_2, \ldots, \lambda_n\} \subseteq \Lambda$ having partial orders $\sqsubseteq$ typically are specified by using an idempotent, commutative, and associative *join* operator $\sqcup$ that satisfies the axioms:

$$\lambda_i \sqsubseteq (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \qquad \text{for } 1 \le i \le n \tag{5.5}$$

$$(\lambda_1 \sqsubseteq \lambda \ \wedge\ \lambda_2 \sqsubseteq \lambda \ \wedge\ \cdots \ \wedge\ \lambda_n \sqsubseteq \lambda) \ \Rightarrow\ (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \sqsubseteq \lambda \tag{5.6}$$

Axiom (5.5) says that $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ is an upper bound for $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, and axiom (5.6) says that $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ is a least upper bound. We can ensure that $\Lambda$ contains least upper bound $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ for any subset $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ of $\Lambda$ simply (i) by adding to $\Lambda$ an element $\top_\Lambda$ that satisfies $\lambda \sqsubseteq \top_\Lambda$ for all $\lambda \in \Lambda$, and (ii) by defining $\lambda \sqcup \lambda'$ to equal $\top_\Lambda$ for every pair of labels $\lambda$ and $\lambda'$ where previously $\lambda \sqcup \lambda'$ was not a member of $\Lambda$. Figure 5.1 depicts a set $\Lambda$ of labels and some least upper bounds. Notice that not all labels in $\Lambda$ are related by $\sqsubseteq$—for example, neither $\lambda_1 \sqsubseteq \lambda_6$ nor $\lambda_6 \sqsubseteq \lambda_1$ holds.

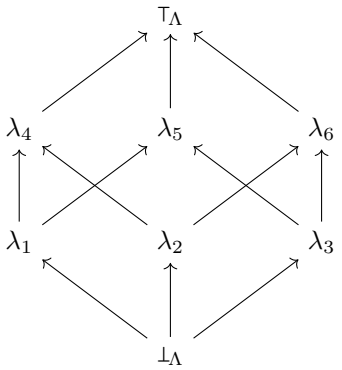
Axiom (5.5) suggests that a definition for $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ satisfying (5.4) can be constructed with $\sqcup$. It is

$$\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))\text{: } \bigsqcup_{1 \le i \le n} \Gamma_{\mathcal{E}}(E_i) \tag{5.7}$$

where we define

$$\bigsqcup_{i \in \mathcal{I}} \lambda_i\text{: } \begin{cases} \bot_\Lambda & \text{if } \mathcal{I} = \varnothing \\ \lambda_{i_1} \sqcup \lambda_{i_2} \sqcup \cdots \sqcup \lambda_{i_n} & \text{if } \mathcal{I} = \{i_1, i_2, \ldots, i_n\} \end{cases} \tag{5.8}$$

 Copyright Fred B. Schneider All rights reserved.




$$\lambda_1 \sqcup \lambda_2 = \lambda_4$$
$$\lambda_1 \sqcup \lambda_3 = \lambda_5$$
$$\lambda_2 \sqcup \lambda_3 = \lambda_6$$
$$\lambda_1 \sqcup \lambda_2 \sqcup \lambda_3 = \top_\Lambda$$
$$\lambda \longrightarrow \lambda' \text{ denotes } \lambda \sqsubset \lambda'$$

Figure 5.1: Examples of $\sqcup$ for $\Lambda = \{\bot_\Lambda, \lambda_1, \ldots, \lambda_6, \top_\Lambda\}$

By combining (5.2), (5.3), and (5.7), we then get the definition for the label $\Gamma_{\mathcal{E}}(E)$ that we give to an expression $E$.

$$\Gamma_{\mathcal{E}}(E)\text{:} \quad \begin{cases} \bot_\Lambda & \text{if } E \text{ is a constant } c \\ \Gamma(v) & \text{if } E \text{ is a variable } v \\ \bigsqcup\limits_{1 \le i \le n} \Gamma_{\mathcal{E}}(E_i) & \text{if } E \text{ is } f(E_1, E_2, \ldots, E_n) \end{cases} \tag{5.9}$$

Some useful corollaries of (5.9) include the following, where *Vars*(*expr*) is the set of variables referenced in *expr*.

$$\Gamma_{\mathcal{E}}(E) \;=\; \bigsqcup_{v \in Vars(expr)} \Gamma(v) \tag{5.10}$$

$$(\Gamma_{\mathcal{E}}(E) \not\sqsubseteq \Gamma(w)) \;\Rightarrow\; (\exists v \in Vars(E)\text{: } \Gamma(v) \not\sqsubseteq \Gamma(w)) \tag{5.11}$$

## 5.2 $\Lambda_{\mathsf{LH}}$: A Simple Label Scheme

The set $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$ of labels, along with the partial order $\sqsubseteq$ and join $\sqcup$ depicted in Figure 5.2, are often used when discussing information flow policies.

- For specifying confidentiality, variables storing public information are given label $\mathsf{L}$, and variables storing secret information are given label $\mathsf{H}$. Because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds, secret values are then prohibited from affecting public values.
- For specifying integrity, variables storing trusted information are given label $\mathsf{L}$ and variables storing untrusted information are given label $\mathsf{H}$. Because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds, untrusted values are prohibited from affecting trusted values.

Many people find it counterintuitive to be using the same label ($\mathsf{H}$) both for untrusted values and for secret values. Here is a way to reconcile these

 Copyright Fred B. Schneider All rights reserved.

| $\sqsubseteq$ | L | H |
|---|---|---|
| L | $\sqsubseteq$ | $\sqsubseteq$ |
| H | $\not\sqsubseteq$ | $\sqsubseteq$ |

(a) Definition of $\sqsubseteq$

| $\sqcup$ | L | H |
|---|---|---|
| L | L | H |
| H | H | H |

(b) Definition of $\sqcup$

Figure 5.2: Definitions of $\sqsubseteq$ and $\sqcup$ for $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$, where $\bot_{\Lambda_{\mathsf{LH}}}$ is L

interpretations. In both cases, the presumed readers of variables with label H is a subset of the presumed readers for variables with label L due to (5.1). For confidentiality, this restriction on readers limits the propagation of secrets; for integrity, the restriction limits the propagation of untrusted information.

Some find it helpful to think of an information flow from a variable with label L (Low) to a variable with label H (High) as information flowing "up", and they think of an information flow from a variable with label H to a variable with label L as information flowing "down". According to that view, two formulations of the information flows that are prohibited by $\sqsubseteq$ in Figure 5.2(a) are:

> *No read up.* A value read from a variable with label H cannot be used to update a variable with label L.
>
> *No write down.* A variable with label L cannot be updated using a value read from a variable with label H.

More succinctly put: "no read up; no write down".

## 5.3 Termination Insensitive Noninterference

*Noninterference policies* prevent a $\lambda$-*observer* (for any $\lambda \in \Lambda$) from learning about the initial values of variables in $V_{\not\sqsubseteq\lambda}$ by reading variables in $V_{\sqsubseteq\lambda}$ at certain designated points during executions. What is considered a "designated point" during an execution depends on what capabilities we attribute to attackers, since attackers are $\lambda$-observers. Some noninterference policies assume that $\lambda$-observers only have access to the initial and final states of those executions that terminate; other noninterference policies assume that $\lambda$-observers can access intermediate states of terminating and non-terminating executions. And some noninterference policies assume that $\lambda$-observers are also capable of detecting that an execution is non-terminating or that an execution has been blocked by an enforcement mechanism.

*Termination insensitive noninterference* (TINI) policies prohibit the values of variables from $V_{\not\sqsubseteq\lambda}$ in initial states from affecting the values of variables from $V_{\sqsubseteq\lambda}$ in final states of terminating executions, for all labels $\lambda \in \Lambda$. So if TINI is being enforced then the initial and final values of variables from $V_{\sqsubseteq\lambda}$ in terminating executions will reveal nothing about the initial values of variables from $V_{\not\sqsubseteq\lambda}$.[2] TINI policies are intended for settings where attackers have limited system access, so the following assumptions are satisfied.

[2]TINI policies thus ignore leaks that occur if an observer can deduce that some execution

 Copyright Fred B. Schneider All rights reserved.

*Batch.* For a program $S$ with variables $V$, a $\lambda$-observer can read variables in $V_{\sqsubseteq\lambda}$ before and after, but not during, terminating executions of $S$.

*Asynchronous.* A $\lambda$-observer cannot distinguish a non-terminating execution from a terminating execution that has not yet terminated.

TINI policies can be formally defined by using a predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ that holds if, for terminating executions by $S$, the initial values of variables in the set $\mathcal{V}$ do not affect the final values of variables in the set $\mathcal{W}$.

**Termination Insensitive Noninterference (TINI).** For a deterministic program $S$ where the variables have labels from a set $\Lambda$ with partial order $\sqsubseteq$:

$$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}) \qquad \square$$

The formal definition for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ uses a function $[\![S]\!](s)$ that characterizes the relevant effects of executing program $S$ from a state $s$.

$$[\![S]\!](s)\colon \begin{cases} s' & \text{if execution of } S \text{ in state } s \text{ terminates in state } s' \\ \Uparrow & \text{if execution of } S \text{ in state } s \text{ is non-terminating} \end{cases} \tag{5.12}$$

The formal definition of $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ also uses a predicate on pairs of states that is satisfied if these states are indistinguishable to an observer that can read only the variables in some given set $\mathcal{V}$. For a state $s$, we write $s.v$ to denote the value of a variable $v$ in a state $s$, and we define the value of a variable $v$ in *state projection* $s|_{\mathcal{V}}$ as follows, where "?" represents an unknown value.

$$s|_{\mathcal{V}}.v\colon \begin{cases} s.v & \text{if } v \in \mathcal{V} \\ ? & \text{otherwise} \end{cases} \tag{5.13}$$

State projections provide a straightforward way to assert that two states give the same values to the variables in $\mathcal{V}$.

$$s =_{\mathcal{V}} s'\colon\ \ s|_{\mathcal{V}} = s'|_{\mathcal{V}}$$

We then have the following formal definition for predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$, where $Init_S$ is the set of initial states of program $S$ and $\overline{\mathcal{V}}$ denotes the set $Vars(S) - \mathcal{V}$.

$$\begin{aligned} \mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}\colon\ \ (\forall s, s' \in Init_S\colon\ & s =_{\overline{\mathcal{V}}} s' \wedge [\![S]\!](s) \neq \Uparrow \wedge [\![S]\!](s') \neq \Uparrow \\ & \Rightarrow [\![S]\!](s) =_{\mathcal{W}} [\![S]\!](s')) \end{aligned} \tag{5.14}$$

---

is non-terminating. For example, deducing that an execution of

```
while x = 0 do skip end
```

is non-terminating implies that $x = 0$ was *true* in the initial state. *Termination sensitive noninteference* (TSNI) strengthens TINI to account for $\lambda$-observers that can detect that an execution is non-terminating.

 Copyright Fred B. Schneider All rights reserved.

Predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ thus specifies a requirement on the states produced by terminating executions of $S$ if those executions start in states $s$ and $s'$, give different values to one or more variables in $\mathcal{V}$, but give the same values to variables not in $\mathcal{V}$. The requirement is that final states $[\![S]\!](s)$ and $[\![S]\!](s')$ must satisfy $[\![S]\!](s) =_{\mathcal{W}} [\![S]\!](s')$ and, therefore, the final values of variables in $\mathcal{W}$ have not been affected by any differences in starting states $s$ and $s'$. Since initial states $s$ and $s'$ differ only in the values for variables in $\mathcal{V}$, we have made a counterfactual argument[3] that the different initial values for variables in $\mathcal{V}$ did not affect the final values of variables in $\mathcal{W}$.

### 5.3.1 TINI in Action

Consider the TINI policy specified by set $\Lambda_{\mathsf{LH}}$ of labels given in Figure 5.2. For a deterministic program $S$ with variables $V$, replacing $\lambda$ with its possible values $\mathsf{L}$ and $\mathsf{H}$ in $V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}$ from the above formal definition of TINI results in

$$V_{\not\sqsubseteq\mathsf{L}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{L}} \quad \wedge \quad V_{\not\sqsubseteq\mathsf{H}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{H}}. \tag{5.15}$$

By expanding $\overset{S}{\not\rightarrow}_{\mathsf{ti}}$ according to definition (5.14), we obtain the following restrictions on the initial and final states of the terminating executions by $S$:

$$\begin{array}{lll} & (\forall s, s' \in \mathit{Init}_S\colon & s =_{\overline{V_{\not\sqsubseteq\mathsf{L}}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ & & \quad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq\mathsf{L}}} [\![S]\!](s')) \\ \wedge & (\forall s, s' \in \mathit{Init}_S\colon & s =_{\overline{V_{\not\sqsubseteq\mathsf{H}}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ ;[\![S]\!](s') \neq \Uparrow \\ & & \quad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq\mathsf{H}}} [\![S]\!](s')) \end{array} \tag{5.16}$$

Writing $V_\lambda$ to denote the set of variables having label $\lambda$, the following hold

$$\overline{V_{\not\sqsubseteq\mathsf{L}}} = V_{\mathsf{L}} \qquad V_{\sqsubseteq\mathsf{L}} = V_{\mathsf{L}} \qquad \overline{V_{\not\sqsubseteq\mathsf{H}}} = V_{\mathsf{L}} \cup V_{\mathsf{H}} \qquad V_{\sqsubseteq\mathsf{H}} = V_{\mathsf{L}} \cup V_{\mathsf{H}}$$

and therefore (5.16) is equivalent to:

$$\begin{array}{lll} & (\forall s, s' \in \mathit{Init}_S\colon & s =_{V_{\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ & & \quad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}}} [\![S]\!](s')) \\ \wedge & (\forall s, s' \in \mathit{Init}_S\colon & s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ & & \quad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} [\![S]\!](s')) \end{array} \tag{5.17}$$

We have that $V_{\mathsf{L}} \cup V_{\mathsf{H}} = V$ holds, since every variable is assigned a label from $\Lambda_{\mathsf{LH}}$. Therefore, predicate $s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s'$ in (5.17) is equivalent to predicate $s = s'$. So the second quantified formula of (5.17) is always satisfied due to the assumption that $S$ is deterministic—terminating executions of deterministic program that

[3] With a *counterfactual argument*, multiple hypothetical starting points or sets of assumptions are the basis for justifying a conclusion.

 Copyright Fred B. Schneider All rights reserved.

| $\Gamma(in)$ | $\Gamma(out)$ | $V_{\sqsubseteq \mathsf{L}}$ | $V_{\not\sqsubseteq \mathsf{L}}$ | $V_{\sqsubseteq \mathsf{H}}$ | $V_{\not\sqsubseteq \mathsf{H}}$ | $out := in$? |
|---|---|---|---|---|---|---|
| L | L | $\{in, out\}$ | $\varnothing$ | $\{in, out\}$ | $\varnothing$ | $\surd$ |
| L | H | $\{in\}$ | $\{out\}$ | $\{in, out\}$ | $\varnothing$ | $\surd$ |
| H | L | $\{out\}$ | $\{in\}$ | $\{in, out\}$ | $\varnothing$ | $\times$ |
| H | H | $\varnothing$ | $\{in, out\}$ | $\{in, out\}$ | $\varnothing$ | $\surd$ |

Figure 5.3: Possible information flow policies for $out := in$

start from the same states produce the same final states. Thus, (5.17) simplifies to:

$$(\forall s, s' \in Init_S\colon\ (s =_{V_\mathsf{L}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow) \\ \Rightarrow\ [\![S]\!](s) =_{V_\mathsf{L}} [\![S]\!](s'))$$

States satisfying $s =_{V_\mathsf{L}} s'$ may differ in the values of variables in $V_\mathsf{H}$ but must agree on the values of variables in $V_\mathsf{L}$. So (5.17) is specifying that the values of variables in $V_\mathsf{H}$ in initial states are not allowed to affect on the values of variables in $V_\mathsf{L}$ in final states or, equivalently, that the values of variables with label H are prohibited from affecting the values of variables with label L. Enforce this TINI policy and variables with label H can store information that must not leak to variables with label L.

We illustrate with the simple program: $out := in$. This program is deterministic, it always terminates, and the value of $in$ in initial states affects the value of $out$ in final states. Figure 5.3 summarizes whether program $out := in$ satisfies the TINI policy defined by the labels that a row gives for variables $in$ and $out$. There is a $\surd$ in the final column if the TINI policy defined by the row is satisfied by executions of $out := in$. The third row has an $\times$ in the final column, because TINI is violated. This violation should not be surprising—TINI prohibits executions where a variable having label H affects a variable having label L, and for the third row $in$ has label H, the initial value of $in$ affects the final value of $out$, but $out$ has label L.

Some implications of various specific TINI policies might be surprising, though. Consider variables $x_\mathsf{L}$ and $x_\mathsf{H}$, with $\Gamma(x_\mathsf{L}) = \mathsf{L}$ and $\Gamma(x_\mathsf{H}) = \mathsf{H}$. The following program shows that TINI can be violated by assignment statements where the expressions are constants, even though constants have label L.

$$\texttt{if } x_\mathsf{H} = 0 \texttt{ then } x_\mathsf{L} := 1 \texttt{ else } x_\mathsf{L} := 2 \texttt{ fi} \tag{5.18}$$

The next program slightly changes the `else` alternative.

$$\texttt{if } x_\mathsf{H} = 0 \texttt{ then } x_\mathsf{L} := 1 \texttt{ else } x_\mathsf{L} := 1 \texttt{ fi} \tag{5.19}$$

TINI is not violated by (5.19) because the same assignment to $x_\mathsf{L}$ is executed for any value of $x_\mathsf{H}$.

Two final programs illustrate that TINI policies are not necessarily violated if assignment statements store values into variables labeled L from variables

 Copyright Fred B. Schneider All rights reserved.

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \texttt{skip} \\
& & \mid \mathit{var} := \mathit{expr} \\
& & \mid \texttt{if } \mathit{expr} \texttt{ then } \mathit{stmt} \texttt{ else } \mathit{stmt} \texttt{ fi} \\
& & \mid \texttt{while } \mathit{expr} \texttt{ do } \mathit{stmt} \texttt{ end} \\
& & \mid \mathit{stmt};\ \mathit{stmt}
\end{array}
$$

Figure 5.4: Syntax for IMP programs

labeled H. In program

$$x_{\mathsf{L}} := x_{\mathsf{H}};\ x_{\mathsf{L}} := 63; \tag{5.20}$$

TINI is satisfied, since the final value of $x_{\mathsf{L}}$ is not affected by the initial value of $x_{\mathsf{H}}$. This next program satisfies TINI if Boolean expression $B$ does not mention $x_{\mathsf{L}}$ or $x_{\mathsf{H}}$, even though a variable with label H affects a variable with label L in the body of the `while`.

$$\texttt{while } B \texttt{ do } x_{\mathsf{L}} := x_{\mathsf{H}} \texttt{ end} \tag{5.21}$$

If $B$ is initially *true* then $B$ will remain *true* (because the only variable changed in the loop body is not mentioned in $B$), so the `while` never terminates. TINI is then satisfied because TINI imposes no restrictions on non-terminating executions. If $B$ is initially *false*, then TINI is satisfied because the loop body is never executed, so problematic assignment statement $x_{\mathsf{L}} := x_{\mathsf{H}}$ is never executed.

### 5.3.2 TINI Enforcement

To be concrete in our discussions about how to enforce TINI and other noninterference policies, Figure 5.4 gives the grammer for IMP, a simple imperative programming language. Instead of including variable declarations, an IMP program will be accompanied by a label assignment $\Gamma(\cdot)$ giving a fixed label $\Gamma(v)$ for each variable $v$. Expressions *expr* in IMP programs are constructed from constants, variables, operators, and functions, as discussed in §5.1.1. Finally, we write "$\ell_i$: $S$" to indicate that a *statement label* $\ell_i$ names the control point associated with the start of statement $S$. Statement labels will also be used to refer to the statement at the control point. No statement label will appear more than once in a program, and statement labels are disjoint from the labels in $\Lambda$.

Assignment statements $\mathit{var} := \mathit{expr}$ are the way an IMP program changes the value of a variable; *var* is called the *target*, and *expr* is called the *source*. IMP provides two kinds of *control-flow statements*: `if` statements and `while` statements. Each control-flow statement has a *guard* and a *body*. The guard is a Boolean expression; the body comprises one or more statements. With an `if` statement, the body comprises a `then` alternative and an `else` alternative; the value of the guard determines which alternative is executed. With a `while`

 Copyright Fred B. Schneider All rights reserved.

```
ℓ1: S1
ℓ2: if G2 then ℓ3: S3
           else ℓ4: while G4 do
                       ℓ5: S5
                       end;
                ℓ6: S6
   fi;
ℓ7: S7
```

| $\ell_i$ | $\Theta_S(\ell_i)$ |
|---|---|
| $\ell_1$ | $\varnothing$ |
| $\ell_2$ | $\varnothing$ |
| $\ell_3$ | $\{G_2\}$ |
| $\ell_4$ | $\{G_2\}$ |
| $\ell_5$ | $\{G_2, G_4\}$ |
| $\ell_6$ | $\{G_2\}$ |
| $\ell_7$ | $\varnothing$ |

Figure 5.5: $\Theta_S(\ell_i)$ for a program $S$

statement, the value of the guard determines whether the body is executed for another iteration or, instead, execution of the `while` statement terminates.

For each statement label $\ell$ in a program $S$, define $\Theta_S(\ell)$ to be the set containing the guards for those control-flow statements having a body that includes statement $\ell$. So the guards in $\Theta_S(\ell)$ could affect whether statement $\ell$ will be reached during some terminating execution of $S$. Figure 5.5 gives $\Theta_S(\cdot)$ for an example program. Notice, $\Theta_S(\ell)$ contains multiple guards when $\ell$ is nested within multiple control-flow statements. Guards in $\Theta_S(\ell)$, however, are not the only guards that can affect whether statement $\ell$ will be reached during executions of a program $S$. In Figure 5.5, for example, $G_4 \notin \Theta_S(\ell_7)$ holds even though $G_4$ could affect whether $\ell_7$ will be reached—$G_4$ affects whether `while` statement $\ell_4$ terminates, and if that `while` statement does not terminate then $\ell_7$ will not be reached.

By definition, an execution of a program $S$ that violates TINI must be terminating and it must execute some assignment statement. There are two ways that executing an assignment statement $\ell$: $w := expr$ could violate TINI because $\ell$ causes an *illicit flow*. With an illicit *explicit flow*, the illicit flow is from some variable in $expr$. An illicit explicit flow cannot occur during execution of $\ell$: $w := expr$ if the following holds.

$$\Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w) \tag{5.22}$$

With an illicit *implicit flow*, the illicit flow is from some guard $G$ that does not satisfy $\Gamma_{\mathcal{E}}(G) \sqsubseteq \Gamma(w)$ but affects whether $\ell$: $w := expr$ is executed. Such an illicit explicit flow cannot occur if

$$\left( \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \;\sqsubseteq\; \Gamma(w). \tag{5.23}$$

holds, since then all guards affecting whether $\ell$ gets executed satisfy $\Gamma_{\mathcal{E}}(G) \sqsubseteq \Gamma(w)$. Therefore, the following condition ensures that executing an assignment statement $\ell$: $w := expr$ does not cause an illicit explicit flow or an illicit implicit flow.

 Copyright Fred B. Schneider All rights reserved.

**$\Theta_S$-Safe Assignment Statements.** Ensure that

$$\left(\Gamma_{\mathcal{E}}(expr) \;\sqcup\; \left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)\right)\right) \;\sqsubseteq\; \Gamma(w) \tag{5.24}$$

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

$\Theta_S$-Safe Assignment Statements is conservative—programs that comply will satisfy TINI, but programs that do not comply might also satisfy TINI. One reason that a program could be speciously rejected is that definition (5.9) for $\Gamma_{\mathcal{E}}(\cdot)$ ignores the semantics of expressions. If, for example, variables $v$ and $w$ satisfy $\Gamma(v) \not\sqsubseteq \Gamma(w)$ then the program $w := v - v$ does not satisfy $\Theta_S$-Safe Assignment Statements because $\Gamma_{\mathcal{E}}(v - v) = \Gamma(v)$ and, therefore, $\Gamma_{\mathcal{E}}(v - v) \sqsubseteq \Gamma(w)$ does not hold. However, program $w := v - v$ does satisfy TINI, since the final value of $w$ is the same for all initial values of $v$.

A second reason that programs could be speciously rejected is that $\Theta_S$-Safe Assignment Statements ignores context. Program (5.19) is rejected by $\Theta_S$-Safe Assignment Statements due to the label of `if` statement guard $x_{\mathsf{H}} = 0$. Yet this program satisfies TINI, because the `then` and the `else` alternatives each store the same value into $x_{\mathsf{L}}$ for any initial value of $x_{\mathsf{H}}$. A different effect of context is seen in program (5.20), where an assignment statement $x_{\mathsf{L}} := x_{\mathsf{H}}$ that does not satisfy $\Theta_S$-Safe Assignment Statements is followed by an assignment statement $x_{\mathsf{L}} := 63$ that overwrites the illicit update.

### 5.3.3 Dynamic Enforcement of TINI

A *reference monitor*[4] is invoked in response to certain specified events that occur as some *monitored program* executes. Once invoked, the reference monitor may update its state and, based on its state, either block further execution by the monitored program or allow execution of the monitored program to continue. So when a reference monitor is present, each execution of a monitored program is blocked, terminating, or non-terminating. Also, the decision to block further execution of a monitored program must be made without any knowledge of program statements in the monitored program that have not yet executed.

A reference monitor to enforce TINI blocks further progress and deletes the program state when a monitored program is about to perform an action that would violate TINI. Therefore, blocked executions are indistinguishable from non-terminating executions. Since TINI imposes no constraints on non-terminating executions, it would seem sensible for TINI to impose no constraints on other executions that are indistinguishable from blocked executions. So the definition of TINI as imposing constraints only on terminating executions remains unchanged.

The only way for a terminating execution of an IMP program $S$ to violate TINI is by executing an assignment statement that does not satisfy $\Theta_S$-Safe

[4] Chapter 12 gives a detailed treatment of reference monitors.

 Copyright Fred B. Schneider All rights reserved.

| upon $S$ reaching • | action to be performed |
|---|---|
| • $w := expr$ | `require(` $\texttt{top}(sti) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$ `)` |
| • `if` $G$ `then` ... | $\texttt{push}(sti, \texttt{top}(sti) \sqcup \Gamma_{\mathcal{E}}(G))$ |
| ... `fi` • | $\texttt{pop}(sti)$ |
| • `while` $G$ `do` ... | $\texttt{push}(sti, \texttt{top}(sti) \sqcup \Gamma_{\mathcal{E}}(G))$ |
| ... `end` • | $\texttt{pop}(sti)$ |

Figure 5.6: Reference monitor $\mathcal{R}_{TI}$ for TINI

Assignment Statements condition (5.24). So a reference monitor for enforcing TINI should be invoked and check this condition whenever an assignment statement is about to execute. To perform this check for an assignment statement $\ell$: $w := expr$ in some monitored program $S$, the reference monitor needs labels $\Gamma(w)$, $\Gamma_{\mathcal{E}}(expr)$, and $\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$. The values of these labels can be determined using $\Gamma(\cdot)$, as follows.

- $\Gamma(w)$ and $\Gamma_{\mathcal{E}}(expr)$ can be determined by the reference monitor if (i) reaching an assignment statement $\ell$: $w := expr$ is an event that causes the reference monitor to be invoked, and (ii) the name of target $w$ and the names of variables referenced in $expr$ are delivered to the reference monitor with that event.

- $\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$ can be calculated by the reference monitor if (i) reaching or exiting `if` and `while` statements are events that cause the reference monitor to be invoked and (ii) a reason (`if` $G$, `fi`, `while` $G$, or `end`) for the event is available to the reference monitor.

Figure 5.6 gives the actions for such a reference monitor $\mathcal{R}_{TI}$. A $\texttt{require}(B)$ statement is used there in describing those reference monitor actions. Execution of $\texttt{require}(B)$ evaluates $B$. If $B$ evaluates to *false* then the reference monitor terminates the monitored program that was being executed when the reference monitor was invoked and also deletes the state of that program; if $B$ evaluates to *true* then the monitored program is allowed to proceed.

$\mathcal{R}_{TI}$ is invoked and checks $\Theta_S$-Safe Assignment Statements condition (5.24) whenever an assignment statement is reached in monitored program $S$. To facilitate this checking, $\mathcal{R}_{TI}$ is also invoked to update a stack[5] $sti$ whenever $S$ reaches or exits a control-flow statement. (Assume that a new instance of stack $sti$ is allocated and initialized to empty for each monitored program $S$.) The

[5] We use operations $\texttt{push}(sti, v)$ to insert value $v$ onto stack $sti$; $\texttt{pop}(sti)$ to remove the most recently added value from stack $sti$; and a function $\texttt{top}(sti)$ that returns the value currently at the top of stack $sti$.

 Copyright Fred B. Schneider All rights reserved.

updates to this stack ensure that

$$\texttt{top}(sti) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \tag{5.25}$$

holds whenever an assignment statement (say) $\ell$: $w := expr$ is about to execute in the monitored program. Therefore, the value of $\texttt{top}(sti)$ along with $\Gamma(w)$ and $\Gamma_{\mathcal{E}}(expr)$, can be used by the reference monitor to check $\Theta_S$-Safe Assignment Statements condition (5.24) for that assignment statement.

**What $\mathcal{R}_{TI}$ Enforces.** The result of executing a program $S$ with $\mathcal{R}_{TI}$ present is a combined program, which we represent using notation $\mathcal{R}_{TI} \rhd S$. By construction, terminating executions of $\mathcal{R}_{TI} \rhd S$ are terminating executions of $S$ where $\Theta_S$-Safe Assignment Statements condition (5.24) holds for every assignment statement that was executed.

For $\mathcal{R}_{TI} \rhd S$ to satisfy TINI, the following must hold.

$$(\forall \lambda \in \Lambda\colon\ V_{\not\in\lambda} \overset{\mathcal{R}_{TI} \rhd S}{\not\longrightarrow_{\mathsf{ti}}} V_{\sqsubseteq\lambda}) \tag{5.26}$$

If (5.26) does not hold then, according to definition (5.14) of $\mathcal{V} \overset{S}{\not\longrightarrow_{\mathsf{ti}}} \mathcal{W}$, there would be initial states $s$ and $s'$ of terminating executions that agree on the initial values of all variables in $V_{\sqsubseteq\lambda}$ but do not agree on the final values of those variables. We prove that this scenerio is impossible by assuming that such a problematic pair of terminating executions exists and deriving a contradiction.

If two executions of $\mathcal{R}_{TI} \rhd S$ do not have the same final values for some variables in $V_{\sqsubseteq\lambda}$, then there must be an earliest state where the values for one or more of those variables disagree. The disagreement must be caused by an assignment statement that was affected by some variable outside of $V_{\sqsubseteq\lambda}$, since the two executions agreed on values for variables from $V_{\sqsubseteq\lambda}$ in all previous states. However, such an assignment statement would have violated $\Theta_S$-Safe Assignment Statements condition (5.24), so execution would be blocked before performing that assignment statement, which contradicts the assumption that we started with terminating executions. We conclude that $\mathcal{R}_{TI} \rhd S$ satisfies TINI.

However, the matter of leaks is more nuanced. If $S$ does not also satisfy TINI then there must be terminating executions of $S$ that become blocked executions of $\mathcal{R}_{TI} \rhd S$. The initial states of the resulting smaller set of terminating executions for $\mathcal{R}_{TI} \rhd S$ must then exhibit additional correlations over the correlations present in the initial states of the terminating executions for $S$ without $\mathcal{R}_{TI}$ present. But if variables in initial states are correlated then the value of one can be used to predict the values of the others, potentially compromising confidentiality. That suggests $\mathcal{R}_{TI} \rhd S$ might exhibit a leak.

To make this concrete, here is an example. Assume that $\Gamma(x_{\mathsf{L}}) = \mathsf{L}$ and $\Gamma(x_{\mathsf{H}}) = \mathsf{H}$ hold.

$$\texttt{if } even(x_{\mathsf{H}}) \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else skip fi} \tag{5.27}$$

 Copyright Fred B. Schneider All rights reserved.

$\mathcal{R}_{TI}$ blocks executions of (5.27) that start in states where $even(x_{\mathsf{H}})$ is *true*; $\mathcal{R}_{TI}$ does not block executions that start in states where $even(x_{\mathsf{H}})$ is *false*. So an $\mathsf{L}$-observer of a terminating execution of (5.27) when $\mathcal{R}_{TI}$ is present learns something about the initial value of $x_{\mathsf{H}}$—that initially $x_{\mathsf{H}}$ was odd. Arguably, that's a leak. Yet TINI holds, because differences in the initial values for $x_{\mathsf{H}}$ in terminating executions are not visible to an $\mathsf{L}$-observer reading $x_{\mathsf{L}}$ when execution terminates. Moreover, an $\mathsf{L}$-observer cannot detect that an execution is blocked and, therefore, cannot determine that the initial value of $x_{\mathsf{H}}$ is even—apparently, there is no leak.

### 5.3.4 Typing Rules to Enforce TINI

A type-safe programming language will come with *typing rules* that derive the set of *type-correct* programs. The typing rules will have been formulated to ensure that all executions of type-correct programs are guaranteed to satisfy certain properties. You are doubtless familiar with typing rules to ensure that only the right kinds of values are stored into specific program variables or appear as arguments to certain operations. Such typing rules, for example, reject programs that perform arithmetic operations on variables storing character strings. In this section, we give typing rules that ensure type-correct programs satisfy TINI.

To assert that a program or statement $S$ is type-correct, we use *judgements*

$$\Gamma, \gamma \vdash_{\mathsf{ti}} S \tag{5.28}$$

where *typing context* $\Gamma$ is a label assignment, and *control context* $\gamma$ is a label from $\Lambda$.[6] Judgements that satisfy certain constraints are defined to be *valid*.

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

Requirement (i) ensures that type-safe programs comply with TINI. Requirement (ii) ensures that $\Theta_S$-Safe Assignment Statements is not violated if $S$ is put in the body of a control-flow statement that has a guard $G$ satisfying $\Gamma_{\mathcal{E}}(G) \sqsubseteq \gamma$. So requirement (ii) allows valid judgements for a compound statement to be derived from valid judgements for its component statements. A derivation below will illustrate.

**Typing Rules.** Each typing rule R is specified as a *schema*

$$\text{R:}\ \frac{H_1,\ H_2,\ \ldots,\ H_n}{\Gamma, \gamma \vdash_{\mathsf{ti}} S}$$

[6] Consistent with the IMP syntax given in Figure 5.4, "statement" and "program" are used interchangeably in the following discussions.

 Copyright Fred B. Schneider All rights reserved.

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

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

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

Figure 5.7: Typing rules for TINI compliance

that gives a procedure for deriving the rule's *conclusion* $\Gamma, \gamma \vdash_{\mathsf{ti}} S$ by mechanically transforming some or all of the rule's *hypotheses* $H_1, H_2, \ldots, H_n$. By design, the conclusion of a typing rule will be a valid judgement if each of the rule's hypotheses is valid.

Figure 5.7 gives a set of typing rules for enforcing TINI in IMP programs. An IMP program $S$ is considered type-correct if judgement $\Gamma, \bot_\Lambda \vdash S$ can be derived using these typing rules, because having $\Gamma, \bot_\Lambda \vdash S$ be valid implies that $S$ satisfies TINI. So, TINI is enforced if IMP programs that are type-correct are allowed to execute but other programs are not allowed to execute.

The typing rules in Figure 5.7 ensure that assignment statements in type-correct programs do not violate $\Theta_S$-Safe Assignment Statements condition (5.24). TINI then follows. An example is a good way to see how the rules prevent violations of $\Theta_S$-Safe Assignment Statements condition (5.24). Consider the following possible conclusion of rule IF, where $S'$ denotes an IMP statement.

$$\Gamma, \bot_\Lambda \ \vdash_{\mathsf{ti}} \ S\colon \texttt{if } B \texttt{ then } \ell\colon w := expr \texttt{ else } S' \texttt{ fi} \tag{5.29}$$

To derive this judgement requires having a derivation for each hypothesis of rule IF. Substituting $\Gamma_{\mathcal{E}}(B)$ for $\lambda$ due to the first hypothesis, the second hypothesis requires a derivation of the following.

$$\Gamma, \ \bot_\Lambda \sqcup \Gamma_{\mathcal{E}}(B) \ \vdash_{\mathsf{ti}} \ \ell\colon \ w := expr$$

Rule ASSIGN must be used to derive this judgement, and the required hypothesis for that derivation is satisfied provided the following holds

$$\bot_\Lambda \sqcup \Gamma_{\mathcal{E}}(B) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w),$$

which is equivalent to $\Gamma_{\mathcal{E}}(B) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$. For program $S$, we have that $\Theta_S(\ell)$ is $\{B\}$ and, therefore, the following holds.

$$\Gamma_{\mathcal{E}}(B) \ = \ \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$$

 Copyright Fred B. Schneider All rights reserved.

1. $\Gamma_{\mathcal{E}}(0) = \mathsf{L}$ ... defn (5.9) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\bot_{\Lambda_{\{\mathsf{L},\mathsf{H}\}}} = \mathsf{L}$ .
2. $\Gamma(m) = \mathsf{H}$ ... assumption.
3. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{L}) \sqsubseteq \mathsf{H}$ ... defns of $\sqcup$ and $\sqsubseteq$ in Figure 5.2.
4. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := 0$ ... ASSIGN with 1, 2, 3.
5. $\Gamma_{\mathcal{E}}(y) = \mathsf{H}$ ... defn (5.9) of $\Gamma_{\mathcal{E}}(\cdot)$, given assumption $\Gamma(y) = \mathsf{H}$.
6. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{H}) \sqsubseteq \mathsf{H}$ ... defns of $\sqcup$ and $\sqsubseteq$ in Figure 5.2.
7. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := y$ ... ASSIGN with 5, 2, 6.
8. $\Gamma_{\mathcal{E}}(x \le y) = \mathsf{H}$ ... defn (5.9) of $\Gamma_{\mathcal{E}}(\cdot)$, since
   $\Gamma_{\mathcal{E}}(x \le y) = (\Gamma(x) \sqcup \Gamma(y)) = (\mathsf{L} \sqcup \mathsf{H}) = \mathsf{H}$.
9. $\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if } x \le y \texttt{ then } m := 0 \texttt{ else } m := y \texttt{ fi}$ ... IF with 8, 4, 7.

Figure 5.8: Example of Hilbert-style proof format

So we have showed that the derivation of (5.29) implies

$$\left( \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \sqcup \Gamma_{\mathcal{E}}(expr) \quad \sqsubseteq \quad \Gamma(w).$$

This is exactly what $\Theta_S$-Safe Assignment Statements condition (5.24) requires for assignment statement $\ell$: $w := expr$, since whether $\ell$ executes is affected by guard $B$ of the `if` statement (and by no other guards).

**Proof Formats.** Various formats are available for presenting the derivation of a judgement to establish that some given IMP program is type-correct. Each has advantages and disadvantages. To illustrate the different formats, we use each to give the type-correctness derivation for the following judgement

$$\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if } x \le y \texttt{ then } m := 0 \texttt{ else } m := y \texttt{ fi} \tag{5.30}$$

assuming $\Gamma(x) = \mathsf{L}$, $\Gamma(y) = \mathsf{H}$, and $\Gamma(m) = \mathsf{H}$ hold, $\Lambda$ is $\{\mathsf{L}, \mathsf{H}\}$, and the rules for evaluating expressions involving $\sqsubseteq$ and $\sqcup$ are those given in Figure 5.2.

*Hilbert-Style Proof Format.* Figure 5.8 gives a type-correctness derivation as a list of sequentially numbered *steps*. Each step comprises a formula $F$ (often, a judgement) and a rationale $R$. When $F$ is a judgement, $R$ names a typing rule and lists the numbers for earlier steps that discharge hypotheses needed to derive $F$ by using that rule. Sometimes the validity of a hypothesis is given as part of the justification rather than by referencing an earlier step. Such inline justifications are used by steps 1, 2, 3, 5, and 8 of Figure 5.8.

*Derivation-Tree Proof Format.* Figure 5.9 gives the type-correctness derivation as a series of derivation trees. A *derivation tree* vertically stacks instances of typing rules, positioning the conclusion of one rule to appear as a hypothesis for another rule. Three derivation trees appear in Figure 5.9. Tags (DT1 and DT2) on the first two derivation trees allow their conclusions to be used for

 Copyright Fred B. Schneider All rights reserved.

$$\text{ASSIGN:}\ \frac{((\mathsf{L}\sqcup\mathsf{H})\sqcup\Gamma_{\mathcal{E}}(0))\ \sqsubseteq\ \mathsf{H}}{\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=0} \tag{DT1}$$

$$\text{ASSIGN:}\ \frac{((\mathsf{L}\sqcup\mathsf{H})\sqcup\Gamma_{\mathcal{E}}(y))\ \sqsubseteq\ \mathsf{H}}{\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=y} \tag{DT2}$$

$$\text{IF:}\ \frac{\Gamma_{\mathcal{E}}(x\le y)=\mathsf{H},\quad \text{DT1:}\ \dfrac{\cdots}{\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=0},\quad \text{DT2:}\ \dfrac{\cdots}{\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=y}}{\Gamma,\mathsf{L}\vdash_{\mathsf{ti}} \texttt{if}\ x\le y\ \texttt{then}\ m:=0\ \texttt{else}\ m:=y\ \texttt{fi}}$$

Figure 5.9: Example of derivation tree proof format

1. $\Gamma,\mathsf{L}\vdash_{\mathsf{ti}} \texttt{if}\ x\le y\ \texttt{then}\ m:=x\ \texttt{else}\ m:=y\ \texttt{fi}$ IF with 1.1, 1.2, and 1.3.
   - 1.1. $\Gamma_{\mathcal{E}}(x\le y)=\mathsf{H}$ — $\Gamma_{\mathcal{E}}(x\le y)=(\Gamma(x)\sqcup\Gamma(y))=(\mathsf{L}\sqcup\mathsf{H})=\mathsf{H}$.
   - 1.2. $\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=0$ — ASSIGN with 1.2.1 and 1.2.2.
     - 1.2.1. $\Gamma_{\mathcal{E}}(0)=\mathsf{L}$ — Definition (5.9) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\bot_{\{\mathsf{L},\mathsf{H}\}}=\mathsf{L}$.
     - 1.2.2. $\Gamma(m)=\mathsf{H}$ — Assumption.
     - 1.2.3. $((\mathsf{L}\sqcup\mathsf{H})\sqcup\mathsf{L})\sqsubseteq\mathsf{H}$ — Definitions of $\sqcup$ and $\sqsubseteq$ in Figure 5.2.
   - 1.3. $\Gamma,\mathsf{L}\sqcup\mathsf{H}\vdash_{\mathsf{ti}} m:=y$ — ASSIGN with 1.3.1, 1.3.2, and 1.3.3.
     - 1.3.1 $\Gamma_{\mathcal{E}}(y)=\mathsf{H}$ — Assumption.
     - 1.3.2. $\Gamma(m)=\mathsf{H}$ — Assumption.
     - 1.3.3. $((\mathsf{L}\sqcup\mathsf{H})\sqcup\mathsf{H})\sqsubseteq\mathsf{H}$ — Definitions of $\sqcup$ and $\sqsubseteq$ in Figure 5.2.

Figure 5.10: Example of hierarchically presented proof format

discharging hypotheses in the third derivation tree. Many people prefer reading derivation trees over reading the Hilbert-style proof format, because derivation trees graphically show dependencies between steps. Derivation trees are a natural format when working with pen and paper, but few text formatters facilitate their construction.

*Hierarchical Proof Format.* A combination of Hilbert-style proofs and derivation trees is to present a list of judgements, but do so hierarchically. This format is illustrated in Figure 5.10. Here, each hypothesis (with a rationale) needed to infer the judgement of step $n$ is listed after step $n$, indented, and numbered by appending sequence numbers to $n$ to get $n.1$, $n.2$, etc. Arbitrary levels of nesting are permitted. With this format, indentation helps readers to see the steps that support a conclusion.

 Copyright Fred B. Schneider All rights reserved.

### 5.3.5 Comparison of TINI Enforcement Mechanisms

Checking type-correctness entails overhead before a program is executed but incurs no runtime overhead. With a reference monitor like $\mathcal{R}_{TI}$, there is no overhead before a program executes. But transfers of control to reference monitor actions incur the overhead of context switches, and the execution of reference monitor actions brings more overhead. The reference monitor, however, only checks an assignment statement when that statement is reached during an execution, so potentially fewer assignment statements need to be checked (although the same assignment statement would be checked each time it is executed).

Permissiveness is often an important difference between type-correctness and a reference monitor. Type-checking rejects any program $S$ containing a statement $T$ that would violate TINI if $T$ is executed in isolation—even if $T$ could never be reached during any terminating execution of $S$. $\mathcal{R}_{TI}$ can be more permissive, as program (5.27) illustrates. Program (5.27) is not type-correct, so type-checking would not allow its execution, but $\mathcal{R}_{TI}$ does not block its executions that start in states where $even(x_{\mathsf{H}})$ is *false*.

Could different typing rules enable substantial improvements in premissiveness when we use type-correctness for enforcement? If the typing rules could identify and ignore unreachable assignment statements then more programs would be type-correct. However, to determine that a statement is reachable would require that the typing rules determine whether `while` statements are guaranteed to terminate and whether the guards for a collection of `if` statements all could hold during one execution. The undecidability of the halting problem implies no algorithm can make such inferences. Since typing rules are actually just defining an algorithm—its what the type-checker executes—we must conclude that inferences about statement reachability cannot be incorporated into typing rules.

## 5.4 Trusted Code and Weaker Policies

Public outputs from many real systems are affected by secret inputs. Examples of such public outputs include encryption of a secret for transmission or storage, redaction[7] of a document for wider disclosure, and transmission of an acknowledgement message to confirm receipt of a request involving secret values. Systems also sometimes can benefit from having trusted outputs be affected by untrusted inputs. Digital signature verification and the use of Byzantine agreement algorithms are examples where the output can be trusted but the input is not. We conclude that noninterference policies may need to be relaxed in parts of real systems.

To avoid these problems, many systems incorporate statements or routines that, by fiat, are allowed to violate noninterference. This is variously known

[7] *Redaction* deletes or obscures parts of a document, producing a version that complies with a given confidentiality policy. Redaction often will be used to delete a name or other personally identifiable information from a document.

 Copyright Fred B. Schneider All rights reserved.

as *trusted code* or, for larger components, *trusted subjects*. During execution of trusted code, the value of any variable is allowed to affect the value of selected variables, independent of labels. The system implementors either verify or simply posit that the trusted code will have the effects that it should, not do things that it shouldn't, and cannot be subverted. An alternative approach, however, is to enforce a security policy that is not as stringent as noninterference. Such an approach would depend on the weaker properties that the trusted code satisfies.

***Uncertainty-Based Security.** An example of such a weaker security policy is *uncertainty-based security*. This security policy asserts that $\lambda$-observers are able to conclude only that the variables in $V_{\not\sqsubseteq\lambda}$ could have many possible initial values and, therefore, the specific initial values of those variables remains confidential.

> **Uncertainty-Based Security.** Despite knowing the values of variables in $V_{\sqsubseteq\lambda}$, a $\lambda$-observer still has sufficient uncertainty about possible initial values for the variables in $V_{\not\sqsubseteq\lambda}$. □

Uncertainty-Based Security notably does not prohibit the values of variables in $V_{\sqsubseteq\lambda}$ from being affected by the values of variables in $V_{\not\sqsubseteq\lambda}$. So Uncertainty-Based Security is weaker than noninterference. But Uncertainty-Based Security does prevent a $\lambda$-observer from learning the initial values of variables in $V_{\not\sqsubseteq\lambda}$. The confidentiality examples mentioned at the beginning of this subsection—encryption, redaction, and transmission of acknowledgments—satisfy Uncertainty-Based Security but they violate noninterference.

We use an example—a secret ballot election—to illustrate how compliance with Uncertainty-Based Security might be established. In a secret ballot election, each voter $i$ stores into a ballot $b_i$ the name of some candidate from a set $C$; the winner $m$ of the election is the candidate named in a majority[8] of the ballots:

$$S\colon \quad m := maj(b_1, \ldots, b_n) \tag{5.31}$$

We assume that only voter $i$ ever has access to ballot $b_i$, but $m$ can be read by all voters.

The voters in a secret ballot election expect compliance with *ballot confidentiality*. This security policy stipulates that the value of ballot $b_i$ and the value of winner $m$ does not allow a voter $i$ to rule out any possible value for a ballot $b_j$ if $i \neq j$ holds. It is implied by an instance of Uncertainty-Based Security where, for an initial system state $s$ and each candidate $c \in C$, there will be an initial system state $s'$ that satisfies $b_j = c$ and $s'$ is indistinguishable to voter $i$ from $s$.

$$\begin{array}{l}(\forall i, j,\ i \neq j\colon\ (\forall c \in C\colon \\ \qquad (\forall s\colon\ (\exists s'\colon\ s =_{\{b_i\}} s'\ \wedge\ s'.b_j = c\ \wedge\ [\![S]\!](s) =_{\{b_i, m\}} [\![S]\!](s')\ )))))\end{array} \tag{5.32}$$

[8] To simplify the discussion, assume that a majority always exists.

 Copyright Fred B. Schneider All rights reserved.

To establish compliance with (5.32), it suffices to exhibit a function $\mathcal{SK}(i, j, c, s)$ for producing a state $s'$ that satisfies[9]

$$s =_{\{b_i\}} s' \quad \wedge \quad s'.b_j = c \quad \wedge \quad [\![S]\!](s) =_{\{b_i, m\}} [\![S]\!](s'). \tag{5.33}$$

provided $i \neq j$ holds. So $\mathcal{SK}(i, j, c, s)$ is producing witnesses $s'$ for demonstrating that the variables a voter $i$ can access before and after an execution from initial state $s$ will have the same values as for an execution from an initial state $s'$ in which $b_j = c$ could hold for any candidate $c$. The values of the variables that voter $i$ can read thus rule out no possible value of $b_j$.

Of course, there is no guarantee that a function $\mathcal{SK}(i, j, c, s)$ satisfying these requirements exists. But if we can give a construction fir $\mathcal{SK}(i, j, c, s)$, then we establish that (5.32) holds and, therefore, ballot confidentiality is being enforced. Function $\mathcal{SK}(i, j, c, s)$ produces a state, so a construction for $\mathcal{SK}(i, j, c, s)$ would show how to produce a mapping from variables to values. The construction we give assumes that $i \neq j$ holds and is formulated as a set of terms "$var \mapsto val$" that each indicate a value $val$ that the state is giving to a variable $var$; a value that is unknown or uninitialized is represented with ?.

$$\mathcal{SK}(i, j, c, s)\text{:} \quad \left[\begin{array}{l} b_i \mapsto s.b_i \\ b_j \mapsto c \\ b_k \mapsto \underset{1 \leq h \leq n}{maj}\,(s.b_h) \quad \text{for } 1 \leq k \leq n \ \wedge \ k \neq i \ \wedge \ k \neq j \\ m \mapsto ? \end{array}\right]$$

If there are 2 candidates and at least 5 voters, it is straightforward to establish that (5.33) is satisfied when $s'$ is replaced by this definition for $\mathcal{SK}(i, j, c, s)$.[10]

However, (5.33) is not satisfied for elections with 2 candidates and only 3 voters, Moreover, there is no definition for $\mathcal{SK}(i, j, c, s)$ that produces states satisfying (5.33) for such elections—with too few voters, knowing the values of $b_i$ and $m$ sometimes will completely eliminate a voter $i$'s uncertainty about $b_j$. Here is an example. Suppose $C$ is $\{c_1, c_2\}$, and we are concerned about voter 1 learning the value of $b_3$. Consider an initial state $s$

$$s\text{:} \quad [m \mapsto ?,\ b_1 \mapsto c_1,\ b_2 \mapsto c_2,\ b_3 \mapsto c_2],$$

[9] We are proving an existentially-quantified formula $(\exists x\text{:}\ P(x))$ by identifying an expression $E$ that satisfies $P(E)$ and, therefore, generates a witness to the existence of $x$. This reasoning is embodied in a standard Predicate Logic inference rule: $\frac{P(E)}{(\exists x\text{:}\ P(x))}$. Expression $E$ is called a *Skolem function*.

[10] We show that each conjunct of (5.33) holds. The first conjunct is $s =_{\{b_i\}} \mathcal{SK}(i, j, c, s)$, and it is satisfied because $\mathcal{SK}(i, j, c, s)$ is constructed using $b_i \mapsto s.b_i$. The second conjunct, which is $\mathcal{SK}(P, j, c, s).b_j = c$, is satisfied because $\mathcal{SK}(i, j, c, s)$ is constructed using $b_j \mapsto c$.

The final conjunct is $[\![S]\!](s) =_{\{b_i, m\}} [\![S]\!](s')$. State $\mathcal{SK}(i, j, c, s)$ gives all but 2 ballots—$b_i$ and $b_j$—a value $w$ (say) equal to $maj(s.b_1, \ldots, s.b_n)$. So at least $n - 2$ ballots in state $\mathcal{SK}(P, j, c, s)$ have value $w$. Because $n \geq 5$ holds, $n - 2$ ballots having value $w$ constitues a majority. Therefore, states $s$ and $\mathcal{SK}(i, j, c, s)$ have the same majority, which means executing $S$ either from initial state $s$ or from initial state $\mathcal{SK}(i, j, c, s)$ will assign the same value to $m$, as required for the final conjunct to hold.

 Copyright Fred B. Schneider All rights reserved.

so $c_2$ is the majority. $\mathcal{SK}(1, 3, c, s)$ would have to produce a state where $b_3 = c_1$ holds and the majority remains $c_2$. However, no value for $b_2$ result in having $c_2$ still be the majority. So there is no function $\mathcal{SK}(i, j, c_1, s)$ that produces a state satisfying (5.33). The requirement for at least 5 voters when there are 2 candidates often surprises people who have used informal reasoning and ignored edge cases. That is a lesson about the use of informal assurance arguments for trusted code.

## Notes and Reading for Chapter 5

Dorothy Denning was the first to suggest that security policies ought to specify restrictions on information flow. This work is summarized in two papers [4, 7], which are based on her Ph.D. dissertation [3]. An interview [6] with Denning explores what motivated and influenced this work. Denning's dissertation introduces the terms "explicit flow" and "implicit flow" for distinguishing the information flows caused by control structures.[11] Her dissertation also discusses both fixed and flow-sensitive variables, certification conditions for a static analysis to enforce security policies, and the undecidability of determining whether a program satisfies an information flow policy.

Denning's dissertation characterizes program statements that could cause an information flow but it does not give a formal definition for information flow per se. Her later textbook [5, chptr 5] does give a formal definition. That definition is formulated in terms of entropy as defined by Shannon [20] and, therefore, involves probabilities that executions will enter given states. The need to have those probabilities makes Denning's definition difficult to use in practice.

The formal definitions widely used today for information flow are based on noninterference.[12] Often, Gougen and Messequer [9] will be cited, because that paper introduced and formalized *noninterference assertions*, which specify that actions performed by one group of users do not affect outputs seen by another group of users. It is just a small step from noninterference assertions to an information flow definition that involves checking whether changes to the values of one set variables affects the values of another set, and the term "noninterference" is a suggestive way to describe that situation. Gougen and Messequer [9] was not the first paper to suggest such a counterfactual definition, though. Cohen [2] had previously introduced *strong dependency*, which defined information flow from $x$ to $y$ as variation in $x$ that results in variation in $y$. But

[11] Denning was not the first or the only researcher to have investigated information flows arising from control structures. Fenton [8] had earlier discussed how to prevent such information flows in connection with implementing memoryless subsystems. At the SOSP conference where Denning gave a preliminary version of her paper [4], Jones and Lipton [10] also presented a paper that uses the term "negative inference" to describe leaks caused by control structures.

[12] Alternative definitions that have been suggested, include constraints [16], nondeducibility [21], generalized non-interference [13], restrictiveness [14], selective interleavings [15], trace closure properties [23], and the modular assembly kit [11, 12]. None has attracted a large following.

January 26, 2026 Copyright Fred B. Schneider All rights reserved.

the theory given in Cohen [2] uses inscrutable notation, making the paper hard to understand. Also, strong dependency was the negation of what was sought for security.

With noninterference generally accepted as the formal definition for information flow, all of the pieces were present to define a type system for ensuring compliance with the certification conditions in Denning [3]. Volpano, Smith, and Irvine combine these pieces in a paper [22] that, for programs having a fixed label assignment, gives typing rules to enforce what Sabelfeld and Sands [19] later call termination-insensitive noninterference (TINI). The soundness proof in Volpano, Smith, and Irvine [22] for that type system is the first formal account of the connection between Denning's static analysis and a noninterference policy.

The design of runtime enforcement mechanisms for TINI also attracted attention. Reference monitors were seen as a promising way to achieve increased permissiveness. Sabelfeld and Russo [18] explores the differences in permissiveness and shows that a reference monitor like our $\mathcal{R}_{TI}$ not only enforces TINI but is more permissive than a type system. However, reference monitors do not always lead to increased permissiveness, as will be seen in Chapter 6.

Trusted subjects were introduced in Bell and LaPadula [1] when the requirement that labels form a partial order was found to be too restrictive. The use of uncertainty-based security to avoid those difficulties is proposed in Saatcioğlu and Schneider [17]

# Bibliography

[1] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report EDS-TR-75-306, Electronic Systems Division (AFSC), March 1976.

[2] Ellis S. Cohen. Information transmission in computational systems. In *Proceedings of the Sixth Symposium on Operating System Principles*, SOSP '77, pages 133–139. ACM, November 1977.

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

[4] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[5] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[6] Dorothy E. Denning. Oral history interview with Dorothy E. Denning. Retrieved from the University Digital Conservancy, April 2013.

[7] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

 Copyright Fred B. Schneider All rights reserved.

[8] Jeffrey S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[9] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[10] Anita K. Jones and Richard J. Lipton. The enforcement of security policies for computation. In *Proceedings of the Fifth Symposium on Operating System Principles*, SOSP '75, pages 197–206. ACM, November 1975.

[11] Heiko Mantel. Possibilistic definitions of security – an assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, CSFW '00, pages 185–199. IEEE Computer Society Press, July 2000.

[12] Heiko Mantel. The framework of selective interleaving functions and the modular assembly kit. In *Proceedings of the 3rd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE)*, pages 53–62, Alexandria , VA, USA, November 2005.

[13] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, April 1987.

[14] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, April 1988.

[15] John McLean. A general theory of composition of trace sets closed under selective interleaving functions. In *Proceedings 1994 IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society Press, 1994.

[16] Jonathan K. Millen. Constraints. Part II: Constraints and multilevel security. In Richard A DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*. Academic Press, 1978.

[17] Göktuğ Saatcioğglu and Fred B. Schneider. Assurance for observable declassifications. Submitted for publication.

[18] 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.

[19] Andrei Sabelfeld and David Sands. A Per model of secure information flow in sequential programs. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European*

January 26, 2026 Copyright Fred B. Schneider All rights reserved.

*Joint Conferences on the Theory and Practice of Software, ETAPS'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 40–58. Springer, March 1999.

[20] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3,4):379–423, 623–656, July, Ocober 1948.

[21] David Sutherland. A model of information. In *Proceedings of 9th National Computer Security Conference*, pages 175–183. National Institute of Standards and Technology, National Computer Security Center, September 1986.

[22] 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.

[23] Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 94–102. IEEE Computer Society Press, 1997.

 Copyright Fred B. Schneider All rights reserved.