Chapter 9

Information Flow Control: Basics

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

9.1 Labels Specifying Information Flow Policies

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

[1]A *relation* ρ 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* ρ 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 $a,b \in \mathit{Vals}$.

A partial order ρ 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.

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

What is considered a "designated point" during an execution depends on the information flow policy. With some information flow policies, the designated points are the final states of terminating executions; with others, the designated points are all intermediate states produced during all executions.

For each label $\lambda \in \Lambda$, partial order $\sqsubseteq$ partitions the set $\mathit{Vars}(S)$ of variables in a program S into subsets

$$V_{\sqsubseteq\lambda}\colon\ \ \{v \in \mathit{Vars}(S) \mid \Gamma(v) \sqsubseteq \lambda\} \qquad V_{\not\sqsubseteq\lambda}\colon\ \ \{v \in \mathit{Vars}(S) \mid \Gamma(v) \not\sqsubseteq \lambda\}$$

where the initial value of no variable from $V_{\not\sqsubseteq\lambda}$ is allowed to affect the value of any variable from $V_{\sqsubseteq\lambda}$ at designated points during an execution. Letting $R(\lambda)$ denote the restrictions imposed on access to variables that have a label λ and letting $R(\lambda) \le R(\lambda')$ assert that compliance with $R(\lambda')$ implies compliance with $R(\lambda)$, we have:

$$(\forall \lambda, \lambda' \in \Lambda\colon\ \ \lambda \sqsubseteq \lambda' \Rightarrow R(\lambda) \le R(\lambda')) \tag{9.1}$$

So the restrictions being imposed on access to each variable v also apply to all variables storing values affected by v.

An obvious question is whether enforcing (9.1) is useful. Shouldn't ordinary access control suffice? It doesn't if our concern is the propagation of information. For example, ordinary access control cannot prevent a program that is authorized to read a secret variable x and to write a public variable y from copying x to y. But an information flow policy satisfying (9.1) could prohibit such leaks. Because $\Gamma(x) \sqsubseteq \Gamma(y)$ must hold for a principal to write y after reading x, from (9.1) we conclude that restrictions on x and y must satisfy $R(\Gamma(x)) \le R(\Gamma(y))$. Therefore, accesses to y also must comply with $R(\Gamma(x))$. If $R(\Gamma(x))$ specifies that only a select set of principals are allowed to read secrets and, thus, are allowed to read x, then $R(\Gamma(y))$ cannot allow additional principals to read y, which implies that y cannot be public.

9.1.1 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 uniary 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 constructed from 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.

Constants. The label $\Gamma_{\mathcal{E}}(c)$ that we associate with a constant c ought 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 any variable v. That requirement leads to the definition:

$$\Gamma_{\mathcal{E}}(c)\text{:}\ \ \bot_\Lambda \ \text{ for any constant } c \tag{9.2}$$

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

$$\Gamma_{\mathcal{E}}(v)\text{:}\ \ \Gamma(v) \ \text{ for any variable } v. \tag{9.3}$$

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 $f(E_1, E_2, \ldots E_n)$:

$$\Gamma_{\mathcal{E}}(E_i) \sqsubseteq \Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n)) \quad \text{for } 1 \le i \le n. \tag{9.4}$$

So satisfying (9.4) is the goal.

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 (9.4) is by defining label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ to be an upper bound of set $\{\Gamma_{\mathcal{E}}(E_1), \Gamma(E_2), \ldots, \Gamma_{\mathcal{E}}(E_n)\}$. 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 following axioms.

$$\lambda_i \sqsubseteq (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \qquad \text{for } 1 \le i \le n \tag{9.5}$$

$$(\lambda_1 \sqsubseteq \lambda \ \wedge\ \lambda_2 \sqsubseteq \lambda \ \wedge\ \cdots \ \wedge\ \lambda_n \sqsubseteq \lambda) \ \Rightarrow\ (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \sqsubseteq \lambda \tag{9.6}$$

Axiom (9.5) requires $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ to be an upper bound for $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, and axiom (9.6) requires $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ to be a least upper bound. We can ensure that Λ 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 Λ simply (i) by adding to Λ 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 λ and λ' where previously $\lambda \sqcup \lambda'$ was not a member of Λ. Figure 9.1 depicts a set Λ of labels and some least upper bounds. Notice that not all labels in Λ are related by $\sqsubseteq$—for example, $\lambda_1 \not\sqsubseteq \lambda_6$ holds.

Axiom (9.5) suggests that a definition for $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ satisfying (9.4) can be constructed with $\sqcup$. It is

$$\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))\text{:}\ \ \bigsqcup_{1 \le i \le n} \Gamma_{\mathcal{E}}(E_i) \tag{9.7}$$

$$\begin{tikzcd}
 & \top_\Lambda & \\
\lambda_4 \arrow[ur] & \lambda_5 \arrow[u] & \lambda_6 \arrow[ul] \\
\lambda_1 \arrow[u] \arrow[ur] & \lambda_2 \arrow[ul] \arrow[ur] & \lambda_3 \arrow[ul] \arrow[u] \\
 & \bot_\Lambda \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

$$\lambda_4 = \lambda_1 \sqcup \lambda_2$$
$$\lambda_5 = \lambda_1 \sqcup \lambda_3$$
$$\lambda_6 = \lambda_2 \sqcup \lambda_3$$
$$\top_\Lambda = \lambda_1 \sqcup \lambda_2 \sqcup \lambda_3$$

$\lambda \longrightarrow \lambda'$ denotes $\lambda \sqsubset \lambda'$

Figure 9.1: Examples of $\sqcup$ for $\Lambda = \{\bot_\Lambda, \lambda_1, \ldots, \lambda_6, \top_\Lambda\}$

where we define

$$\bigsqcup_{i \in \mathcal{I}} \lambda_i \colon \begin{cases} \bot_\Lambda & \text{if } \mathcal{I} = \varnothing \\ \lambda_{i_1} \sqcup \lambda_{i_2} \sqcup \cdots \sqcup \lambda_{i_n} & \text{if } \mathcal{I} = \{i_1, i_2, \ldots, i_n\} \end{cases} \tag{9.8}$$

By combining (9.2), (9.3), and (9.7), we then get the following definition for the label $\Gamma_\mathcal{E}(E)$ given to an expression E.

$$\Gamma_\mathcal{E}(E) \colon \begin{cases} \bot_\Lambda & \text{if } E \text{ is a constant } c \\ \Gamma(v) & \text{if } E \text{ is a variable } v \\ \bigsqcup\limits_{1 \le i \le n} \Gamma_\mathcal{E}(E_i) & \text{if } E \text{ is } f(E_1, E_2, \ldots, E_n) \end{cases} \tag{9.9}$$

Useful corollaries of (9.9) include the following, where $\mathit{Vars}(\mathit{expr})$ is the set of variables referenced in *expr*.

$$\Gamma_\mathcal{E}(E) = \bigsqcup_{v \in \mathit{Vars}(\mathit{expr})} \Gamma(v) \tag{9.10}$$

$$(\Gamma_\mathcal{E}(E) \not\sqsubseteq \Gamma(w)) \;\Rightarrow\; (\exists v \in \mathit{Vars}(E) \colon \Gamma(v) \not\sqsubseteq \Gamma(w)) \tag{9.11}$$

9.2 Λ_{LH}: A Simple Label Scheme

The set $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$ of labels, along with the partial order $\sqsubseteq$ and join $\sqcup$ defined in Figure 9.2, are often used as examples when discussing information flow policies.

- For specifying confidentiality, variables storing public information are given label L, and variables storing secret information are given label 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 L and variables storing untrusted information are given label H.

$\sqsubseteq$	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 9.2: Definitions of $\sqsubseteq$ and $\sqcup$ for $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$, where $\bot_{\Lambda_{\mathsf{LH}}}$ is L

Because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds, untrusted values are prohibited from affecting trusted values.

You may find it counterintuitive to be using the same label (H) both for untrusted values and for secret values. Here is a way to reconcile these interpretations. According to (9.1), uses of variables with label H are more restricted than uses of variables with label L. For confidentiality, the added restrictions limit the propagation of secrets; for integrity, the added restrictions limit 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 consequences of the information flows that are prohibited by $\sqsubseteq$ as defined in Figure 9.2(a) are:

No read up. A value derived from a variable with label H cannot be written to a variable with label L.

No write down. A value written to a variable with label L cannot be derived from a variable with label H.

So the prohibitions that Λ_{LH} in Figure 9.2 specifies might be succinctly described by "no read up; no write down".

9.3 Multilevel Security Labels

When enterprises ran using paper documents, an employee was allowed access to a document only if (i) the document's contents were considered relevant to that person's job and (ii) there was reason to believe that the person would not cause significant harm by divulging or corrupting the contents of the document. If each document instead is stored in a separate file then we can achieve this *need-to-know* by using an information flow policy and assigning suitable labels to files and to users. Access authorizations and prohibitions are then achieved through the definition of partial order $\sqsubseteq$ for users U assigned labels $\Gamma(U)$ and files F assigned labels $\Gamma(F)$:

- $\Gamma(F) \sqsubseteq \Gamma(U)$ must hold for a file F to affect user U and, therefore, it specifies that U is allowed to learn whether file F exists and to learn the contents of F.

- $\Gamma(U) \sqsubseteq \Gamma(F)$ must hold for a user U to affect file F and, therefore, it specifies that U is allowed to create, delete, and/or update the contents of file F.

9.3.1 Confidentiality

For enforcing the confidentiality requirements of need-to-know, we use labels that authorize a user to access a file based on (i) the file's content, (ii) whether the access would facilitate the content being leaked, and (iii) the damage such a leak might cause. Each of these *multilevel security* (MLS) labels comprises a pair $\langle \mathcal{T}, \mathcal{C} \rangle$, where $\mathcal{T}$ and $\mathcal{C}$ have different interpretations for files than for users.

> *File Labels.* MLS labels for files are assigned by *classification authorities*—individuals who are both knowledgeable about the subjects covered in the file and understand the broader context necessary for predicting possible damage from leaking the contents of the file.
>
> - $\mathcal{T}$ is a set of topic names describing the information that the file contains. Topic names come from a catalog that has been adopted by the community using these labels.[2]
> - $\mathcal{C}$ is the file's *sensitivity* and categorizes the potential damage if the file contents are leaked. Figure 9.3 gives the sensitivities that the U.S. Department of Defense uses and their definitions.
>
> *User Labels.* These MLS labels might be assigned by the user's employer or by some external agency that performs assessments on behalf of a community.
>
> - $\mathcal{T}$ is a set of topics that describe content already known to this user as well as topics relevant to the user's current position or task assignments.
> - $\mathcal{C}$ is the user's *clearance* and categorizes the extent to which this individual is believed to be trustworthy.[3] The clearance is presumed to predict whether an individual will leak information. Figure 9.3 gives the categories that the U.S. Department of Defense uses for clearances and how each is interpreted.

We assume that MLS labels initially assigned to files and users are accurate. We preserve this accuracy by allowing an information flow only if the MLS

[2]Topic names might be self-explanatory (e.g., chem/bio, crypto, or nuclear) or obscure (e.g., Ultra or Umbra). Obscure names are used so that people who see a label but do not have a need to know are kept in the dark about what the name describes. With the labels used by the U.S. Department of Defense, names whose meanings are secret are called *codewords*. For example, the codeword Ultra was used during World War II to label information that the Allies obtained by decrypting intercepts of German communications, and Umbra is a more recent (but also now-retired) codeword for the most-sensitive kinds communications intercepts.

[3]Either an employer or an external agency would make this assessment. Clearances are typically granted after a person has submitted to a background investigation that seeks to identify character flaws or exploitable personal circumstances. A background investigation might range from a short interview to a polygraph test conducted over multiple days.

$\mathcal{C}$	file's sensitivity: potential damage	individual's clearance: belief of trustworthiness
TS (<u>T</u>op <u>S</u>ecret)	exceptionally grave	strong
S (<u>S</u>ecret)	serious	moderate
C (<u>C</u>onfidential)	some	somewhat
U (<u>U</u>nclassified)	none	unknown

where $\mathsf{U} < \mathsf{C} < \mathsf{S} < \mathsf{TS}$ and $\mathsf{X} \le \mathsf{Y}$ denotes $\mathsf{X} = \mathsf{Y} \vee \mathsf{X} < \mathsf{Y}$.

Figure 9.3: Interpretations for $\mathcal{C}$ in an MLS label $\langle \mathcal{T}, \mathcal{C} \rangle$

labels involved will remain accurate. Since an information flow from P to P' is allowed only if $\Gamma(P) \sqsubseteq \Gamma(P')$ holds, preserving the accuracy of MLS labels is the basis for the definition of partial order $\sqsubseteq$.

$$\langle \mathcal{T}, \mathcal{C} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{C}' \rangle\text{:}\quad \mathcal{T} \subseteq \mathcal{T}' \wedge \mathcal{C} \le \mathcal{C}' \tag{9.12}$$

To show that this definition preserves the accuracy of all MLS labels for any information flow that $\langle \mathcal{T}, \mathcal{C} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{C}' \rangle$ allows, we establish that an information flow allowed by $\langle \mathcal{T}, \mathcal{C} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{C}' \rangle$

(i) never transfers information about topics not covered by $\mathcal{T}'$ and

(ii) never increases the information known to a user deemed less trustworthy than $\mathcal{C}$ and never increases the information that has affected a file that such a user is allowed to read.[4]

An information flow allowed by $\langle \mathcal{T}, \mathcal{C} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{C}' \rangle$ occurs because a user U reads a file F or because a user U updates a file F. Assume $\langle \mathcal{T}_F, \mathcal{C}_F \rangle$ is the label on a file F and $\langle \mathcal{T}_U, \mathcal{C}_U \rangle$ is the label for a user U

U reads F. Since $\Gamma(F) \sqsubseteq \Gamma(U)$ must hold if U is authorized to read F, we conclude that $\mathcal{T}_F \subseteq \mathcal{T}_U$ and $\mathcal{C}_F \le \mathcal{C}_U$ hold due to definition (9.12) for $\sqsubseteq$. All information in F is covered by $\mathcal{T}_F$. From $\mathcal{T}_F \subseteq \mathcal{T}_U$, we conclude that information also is covered by topic list $\mathcal{T}_U$, satisfying requirement (i). From $\mathcal{C}_F \le \mathcal{C}_U$, we conclude that U is believed to be at least as trustworthy as the least trustworthy user that is authorized to read F. So the information known to a user deemed less trustworthy is not increased, satisfying requirement (ii).

U updates F. Since $\Gamma(U) \sqsubseteq \Gamma(F)$ must hold for U to be authorized to update F, we conclude that $\mathcal{T}_U \subseteq \mathcal{T}_F$ and $\mathcal{C}_U \le \mathcal{C}_F$ hold due to definition (9.12) for $\sqsubseteq$. From $\mathcal{T}_U \subseteq \mathcal{T}_F$ we have that any information U writes is

[4]This presumes that increasing the number of people who know a secret does not bring an increased risk of leaks. That is an unsound assumption about the general population. As Benjamin Franklin is reported to have written: "Three can keep a secret if two of them are dead". However, the assumption could be true enough for the population of individuals who have been vetted by background investigations.

$\mathcal{I}$	file's credibility: integrity of content	individual's risk level: amount of knowledge
0	high	expert
1	moderate	somewhat
2	none	clueless

Figure 9.4: Interpretations for $\mathcal{I}$ in an MLS integrity label $\langle \mathcal{T}, \mathcal{I} \rangle$

covered by topic list $\mathcal{T}_F$ because it is covered by $\mathcal{T}_U$, so requirement (i) is satisfied. To discharge (ii), observe that a reader U_R of F must have a clearance $\mathcal{C}_R$ satisfying $\mathcal{C}_F \leq \mathcal{C}_R$. Thus, by transitivity with $\mathcal{C}_U \leq \mathcal{C}_F$ from $\Gamma(U) \sqsubseteq \Gamma(F)$, we conclude that $\mathcal{C}_U \leq \mathcal{C}_R$ must hold. So the information known to a user deemed less trustworthy is not increased, satisfying requirement (ii).

9.3.2 Integrity

Need-to-know is not only concerned with leaks. It also is concerned with limiting access in order to prevent users from corrupting file contents. *Multilevel integrity* (MLI) labels address this need by providing a way to specify that low-integrity information and ignorant users not be allowed to contaminate high-integrity information. Each MLI label comprises a pair $\langle \mathcal{T}, \mathcal{I} \rangle$, where $\mathcal{T}$ and $\mathcal{I}$ are given different interpretations for files than for users.

File Labels.
- $\mathcal{T}$ is a set of topic names describing the information that the file contains.
- $\mathcal{I}$ is the file's *credibility* and categorizes the integrity of the file's content. Figure 9.4 gives a list of possible categories along with their definitions.

User Labels.
- $\mathcal{T}$ gives the set of topic names where this user is knowledgeable to some degree.
- $\mathcal{I}$ is a *risk level* and categorizes the credibility for content produced when this user makes an update. Figure 9.4 gives a list of possible risk levels.

The definition of partial order $\sqsubseteq$ on MLI labels is based on preserving the accuracy of these labels after there has been an information flow that was allowed because $\langle \mathcal{T}, \mathcal{I} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{I}' \rangle$ holds. Such an information flow must

(i) never transfer information about topics not covered by $\mathcal{T}'$, and

(ii) never reduce a file's credibility because an update is made using less credible data or is made by a less knowledgeable individual.

Requirement (i) is equivalent to $\mathcal{T} \subseteq \mathcal{T}'$, and requirement (ii) is equivalent to $\mathcal{I} \leq \mathcal{I}'$. So we obtain the following definition for partial order $\sqsubseteq$ on MLI labels.

$$\langle \mathcal{T}, \mathcal{C} \rangle \sqsubseteq \langle \mathcal{T}', \mathcal{C}' \rangle\text{:}\quad \mathcal{T} \subseteq \mathcal{T}' \ \wedge\ \mathcal{I} \leq \mathcal{I}' \tag{9.13}$$

9.3.3 Case Study: File System Authorization

A file system typically provides operations for reading, writing, creating, and deleting files. If invocations of file system operations can be attributed to users, then we can use an information flow policy to specify permitted and prohibited information flows between users and files. For files F having label $\Gamma(F)$ and users U having label $\Gamma(U)$:

- $\Gamma(F) \sqsubseteq \Gamma(U)$ holds if U is allowed to learn whether file F exists and learn about its contents.
- $\Gamma(U) \sqsubseteq \Gamma(F)$ holds if user U is allowed to create file F, delete it, and/or update its contents.

The labels could be MLS labels, MLI labels, or any other set of labels accompanied by a partial order $\sqsubseteq$ and join $\sqcup$ operator.

The above interpretations of $\Gamma(F) \sqsubseteq \Gamma(U)$ and $\Gamma(U) \sqsubseteq \Gamma(F)$ define rules for when a file system operation should be allowed to proceed. The rules concerning read and write operations are straightforward:

No Read Up. $\Gamma(F) \sqsubseteq \Gamma(U)$ must hold for a user U to read a file F.

No Write Down. $\Gamma(U) \sqsubseteq \Gamma(F)$ must hold for a user U to write a file F.

Together, these rules prevent a user U from copying information in a file F that another user U' cannot read to some file F' that U' can read.[5] As such, the rules prevent *Trojan horse* attacks, where a program is invoked because it appears useful but the program also implements hidden and nefarious functionality.[6] An example of a Trojan horse attack that the rules block would be a game program that copies information from a file that an attacker cannot read to a file that the attacker could read.

[5]We show, by contradiction, that $\Gamma(F') \not\sqsubseteq \Gamma(U')$ necessarily holds and, therefore, No Read Up prevents U' from reading F'. For U to read F requires $\Gamma(F) \sqsubseteq \Gamma(U)$, for U to write F' requires $\Gamma(U) \sqsubseteq \Gamma(F')$ and presumes $\Gamma(F') \sqsubseteq \Gamma(U')$ so that U' could read F'. By transitivity we conclude $\Gamma(F) \sqsubseteq \Gamma(U')$ holds. However, $\Gamma(F) \sqsubseteq \Gamma(U')$ contradicts the initial assumption that $\Gamma(F) \not\sqsubseteq \Gamma(U')$ holds.

[6]Greek mythology recounts how the 10-year Greek siege of Troy was ended by a clever subterfuge. The Greeks built a huge wooden horse, hid a small force of warriors inside, placed the horse outside the gates of Troy, and then appeared to abandon the siege by sailing out of sight. With the Greek force gone, the Trojans opened the city gates and moved the horse—thought to be a tribute marking the end of the siege—inside. The city residents celebrated. But once the sun had set, the Greek fleet turned around and headed back to Troy. At midnight, the Greek warriors inside the horse emerged, killed the Trojan guards, and opened the city gates. The Greek force, which by then had returned, entered the open gates and destroyed the city, thereby ending the war.

The rules for read and write, however, do allow a user U to update a file that U cannot read. Such an update is known as a *blind write*. There is no way to check that a blind write has been correctly performed, so allowing blind writes is a bad idea. To prohibit blind writes, it suffices to require that both $\Gamma(F) \sqsubseteq \Gamma(U)$ (to allow the read) and $\Gamma(U) \sqsubseteq \Gamma(F)$ (to allow the write) hold in order for a user U to update a file F.

No Blind Writes. $\Gamma(U) = \Gamma(F)$ must hold for a user U to write a file F without creating blind writes.

Rules for authorizing file creation and deletion must prevent users from using the existence of a file as a means for communicating a bit of information. A user that creates or deletes a file is affecting that file, so our interpretation of $\sqsubseteq$ leads to the following rules, where the first is a variant of No Blind Writes and the second is a variant of No Write Down.

File Creation. $\Gamma(F)$ is initialized to $\Gamma(U)$ for any file F user U creates.

File Deletion. $\Gamma(U) \sqsubseteq \Gamma(F)$ must hold for a user U to delete a file F.

If only these rules are followed, though, file creation or deletion could be abused to create a channel from user a U_S to a user U_R having labels that satisfy $\Gamma(U_S) \not\sqsubseteq \Gamma(U_R)$. The channel implementation we show uses file creation; an implementation using file deletion is similar. Assume that files named F_1 and F_2 do not exist, U_S and U_R agree on those file names, and an attempt to read or write a non-existent file returns a different error message than an attempt to violate any of the rules given above.

- *Sending a bit.* By using a convention that U_R also knows, U_S chooses between creating file F_1 and creating file F_2 based on the value of bit b. Thus, one of F_1 and F_2 remains non-existent and the other file F (say) satisfies $\Gamma(F) \not\sqsubseteq \Gamma(U_R)$ due to the File Creation Rule and $\Gamma(U_S) \not\sqsubseteq \Gamma(U_R)$.
- *Receiving a bit.* User U_R repeatedly attempts to read both F_1 and F_2. These operation attempts eventually will return a non-existant file error message for only one of the files. U_R then infers the value of b based on the convention that U_S used to decide between creating file F_1 and creating file F_2.

To prevent such channel implementations, it suffices to have error messages for attempts to access non-existent files be indistinguishable from other error messages (i.e., a attempt to create a file that already exists or an attempt to delete a file that does not exist) . One possible scheme is to terminate execution when any error occurs; another possible scheme is to return a single value.

9.4 Termination Insensitive Noninterference

Noninterference policies prevent a so-called λ-*observer* (for any $\lambda \in \Lambda$) from learning about the initial values of variables in $V_{\not\sqsubseteq\lambda}$ by reading variables in $V_{\sqsubseteq\lambda}$.

The capabilities of different threats are modeled by making different assumptions about when λ-observers can access the variables in $V_{\sqsubseteq\lambda}$. Some noninterference policies assume that λ-observers only have access to the initial and final states of a terminating execution; other noninterference policies assume that λ-observers can access intermediate states of terminating and non-terminating executions. And some noninterference policies assume that λ-observers are also capable of detecting that an execution is non-terminating or that an execution have 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 reveal nothing about the initial values of variables from $V_{\not\sqsubseteq\lambda}$.[7] TINI policies are intended for settings that satisfy the following.

> *Batch.* For a program S with variables V, a λ-observer can read variables in $V_{\sqsubseteq\lambda}$ before and after, but not during, terminating executions.
>
> *Asynchronous.* A λ-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 Λ with partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}) \qquad \Box$$

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

$$[\![S]\!](s)\colon \begin{cases} s' & \text{if execution of } S \text{ in state } s \text{ terminates in state } s' \\ \Uparrow & \text{if execution of } S \text{ in state } s \text{ is non-terminating} \end{cases} \tag{9.14}$$

The formal definition of $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ also uses a predicate that is satisfied when two states give the same values to variables in some set $\mathcal{V}$. For a state s, we

[7] TINI policies thus ignore leaks that occur because an observer can reliably ascertain that some execution 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 observers that can detect that an execution is non-terminating.

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.

$$s|_{\mathcal{V}}.v\colon\ \begin{cases} s.v & \text{if } v \in \mathcal{V} \\ ? & \text{otherwise} \end{cases} \tag{9.15}$$

State projections provide a straightforward way to define relations for asserting that two states give the same values to the variables in $\mathcal{V}$ or to the variables in complement $\overline{\mathcal{V}}$ comprising the variables in $Vars(S) - \mathcal{V}$.

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

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 a program S.

$$\begin{aligned} \mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}\colon\quad (\forall s, s' \in Init_S\colon\ & s =_{\overline{\mathcal{V}}} s' \wedge [\![S]\!](s) \neq \Uparrow \wedge [\![S]\!](s') \neq \Uparrow \\ & \Rightarrow [\![S]\!](s) =_{\mathcal{W}} [\![S]\!](s')) \end{aligned} \tag{9.16}$$

Predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ thus specifies a requirement on the states produced by terminating executions of S, started in initial states s and s', where s and s' can give different values to one or more variables in $\mathcal{V}$ but give the same values to variables not in $\mathcal{V}$. That 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 $\mathcal{V}$, a counterfactual argument[8] has established that the different values for variables in $\mathcal{V}$ did not affect the values of variables in $\mathcal{W}$.

9.4.1 TINI in Action

Consider the TINI policy specified by set Λ_{LH} of labels given in Figure 9.2. For a deterministic program S with variables V, this information flow policy specifies the following, obtained by replacing λ with the possible values: L and H in $V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}$ from the above formal definition of TINI:

$$V_{\not\sqsubseteq\mathsf{L}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{L}} \quad\wedge\quad V_{\not\sqsubseteq\mathsf{H}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{H}} \tag{9.17}$$

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

By expanding $\overset{S}{\not\rightarrow}_{\mathsf{ti}}$ according to definition (9.16), we obtain the following restrictions on the initial and final states of the terminating executions by S:

$$\begin{array}{ll} & (\forall s, s' \in \mathit{Init}_S\colon\ s =_{\overline{V_{\not\sqsubseteq \mathsf{L}}}} s' \ \wedge\ [\![S]\!](s) \neq\Uparrow \ \wedge\ [\![S]\!](s') \neq\Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq \mathsf{L}}} [\![S]\!](s')) \\ \wedge & (\forall s, s' \in \mathit{Init}_S\colon\ s =_{\overline{V_{\not\sqsubseteq \mathsf{H}}}} s' \ \wedge\ [\![S]\!](s) \neq\Uparrow \ \wedge\ ;[\![S]\!](s') \neq\Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq \mathsf{H}}} [\![S]\!](s')) \end{array} \tag{9.18}$$

Because the following hold

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

where V_λ is the set of variables having label λ, (9.18) is equivalent to:

$$\begin{array}{ll} & (\forall s, s' \in \mathit{Init}_S\colon\ s =_{V_{\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq\Uparrow \ \wedge\ [\![S]\!](s') \neq\Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}}} [\![S]\!](s')) \\ \wedge & (\forall s, s' \in \mathit{Init}_S\colon\ s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s' \ \wedge\ [\![S]\!](s) \neq\Uparrow \ \wedge\ [\![S]\!](s') \neq\Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} [\![S]\!](s')) \end{array} \tag{9.19}$$

We have that $V_{\mathsf{L}} \cup V_{\mathsf{H}} = V$ holds, since every variable is assigned a label from Λ_{LH}. Therefore, predicate $s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s'$ in (9.19) is equivalent to predicate $s = s'$. So the second quantified formula of (9.19) is satisfied due to the assumption that S is deterministic—terminating executions of deterministic program that start from the same states produce the same final states. Consequently, the second quantified formula of (9.19) is equivalent to *true*, and we conclude that (9.19) simplifies to:

$$\begin{array}{l} (\forall s, s' \in \mathit{Init}_S\colon\ (s =_{V_{\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq\Uparrow \ \wedge\ [\![S]\!](s') \neq\Uparrow\ ) \\ \qquad\qquad \Rightarrow\ \ [\![S]\!](s) =_{V_{\mathsf{L}}} [\![S]\!](s')) \end{array}$$

States satisfying $s =_{V_{\mathsf{L}}} s'$ may differ in the values of variables in V_{H} but must agree on the values of variables in V_{L}. That means (9.19) implies that the values of variables in V_{H} in initial states may not affect on the values of variables in V_{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. So if this TINI policy is enforced, then variables with label H can store information that we do not want leaked to variables with label L.

We illustrate with the simple program: $\mathit{out} := \mathit{in}$. This program is deterministic, it always terminates, and the value of *in* in initial states affects the value of *out* in final states. Each entry in the final column of Figure 9.5 summarizes whether program $\mathit{out} := \mathit{in}$ satisfies the TINI policy defined by the labels that row gives for variables *in* and *out*. There is a ✓ in the final column if the TINI policy defined by the row is satisfied by execution of $\mathit{out} := \mathit{in}$. The third row has an × in the final column. This violation should not be surprising—TINI

$\Gamma(in)$	$\Gamma(out)$	$V_{\sqsubseteq \mathsf{L}}$	$V_{\not\sqsubseteq \mathsf{L}}$	$V_{\sqsubseteq \mathsf{H}}$	$V_{\not\sqsubseteq \mathsf{H}}$	$out := in$?
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 9.5: Possible information flow policies for $out := in$

prohibits executions where a variable having label H affects a variable having label L, and here *in* has label H but *out* has label L.

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

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else } x_{\mathsf{L}} := 2 \texttt{ fi} \tag{9.20}$$

The next program slightly changes the `else` alternative.

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else } x_{\mathsf{L}} := 1 \texttt{ fi} \tag{9.21}$$

The TINI policy is not violated, because the same assignment to x_{L} is executed for any value of x_{H}.

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

$$x_{\mathsf{L}} := x_{\mathsf{H}};\ x_{\mathsf{L}} := 63; \tag{9.22}$$

TINI is satisfied, since the final value of x_{L} is not affected by the initial value of x_{H}.

This last program satisfies TINI if B does not mention x_{L} or x_{H}, even though a variable with label H affects a variable with label L in the body of the `while`.

$$\texttt{while } B \texttt{ do } x_{\mathsf{L}} := x_{\mathsf{H}} \texttt{ end} \tag{9.23}$$

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 impose 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.

9.4.2 TINI Enforcement

To be concrete in our discussions about how to enforce TINI and other noninterference policies, Figure 9.6 gives the grammer for IMP, a simple imperative

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

Figure 9.6: Syntax for IMP programs

programming language. Instead of including variable declarations, an IMP program will be accompanied by a function $\Gamma(\cdot)$ 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 §9.1.1. Finally, we write "ℓ_i: S" to indicate that a *statement label* ℓ_i names the control point associated with the start of statement S. Statement labels will also be used to refer to the statement at a control point. No statement label will appear more than once in a program, and statement labels are disjoint from the labels in Λ.

Assignment statements $\textit{var} := \textit{expr}$ 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` 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 ℓ in a program S, there is a set $\Theta_S(\ell)$ that contains the guard for those control-flow statements having a body that includes statement ℓ. So each guard in $\Theta_S(\ell)$ affects whether statement ℓ will be reached during some terminating execution of S. Figure 9.7 gives $\Theta_S(\cdot)$ for an example program. Notice, $\Theta_S(\ell)$ contains multiple guards when ℓ is nested within multiple control-flow statements. Guards in $\Theta_S(\ell)$, however, are not the only guards that can affect whether statement ℓ will be reached during executions of a program S. In Figure 9.7, for example, $G_4 \notin \Theta_S(\ell_7)$ holds even though G_4 could affect whether ℓ_7 will be reached—G_4 affects whether `while` statement ℓ_4 terminates, and if that `while` statement does not terminate then ℓ_7 will not be reached.

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

$$\Gamma_{\mathcal{E}}(\textit{expr}) \sqsubseteq \Gamma(w) \qquad (9.24)$$

```
$\ell_1$: $S_1$
$\ell_2$: if $G_2$ then $\ell_3$: $S_3$
                else $\ell_4$: while $G_4$ do
                            $\ell_5$: $S_5$
                            end;
                      $\ell_6$: $S_6$
    fi;
$\ell_7$: $S_7$
```

ℓ_i	$\Theta_S(\ell_i)$
ℓ_1	$\varnothing$
ℓ_2	$\varnothing$
ℓ_3	$\{G_2\}$
ℓ_4	$\{G_2\}$
ℓ_5	$\{G_2, G_4\}$
ℓ_6	$\{G_2\}$
ℓ_7	$\varnothing$

Figure 9.7: $\Theta_S(\ell_i)$ for a program S

With an illicit *implicit flow*, the illicit flow is caused by some guard G that affects whether ℓ: $w := expr$ is executed and that does not satisfy $\Gamma_{\mathcal{E}}(G) \sqsubseteq \Gamma(w)$. An illicit explicit flow cannot if none of those guards exists, because the following holds.

$$\left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \;\sqsubseteq\; \Gamma(w). \tag{9.25}$$

Therefore, the following condition ensures that executing an assignment statement ℓ: $w := expr$ does not cause an illicit explicit flow or an illicit implicit flow.

Θ_S-Safe Assignment Statements. Ensure that

$$\left(\Gamma_{\mathcal{E}}(expr) \;\sqcup\; \left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \right) \;\sqsubseteq\; \Gamma(w) \tag{9.26}$$

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

Θ_S-Safe Assignment Statements is conservative—programs that comply will satisfy TINI, but programs that do not comply might also satisfy TINI. One reason for a program to be speciously rejected is that definition (9.9) for $\Gamma_{\mathcal{E}}(\cdot)$ ignores the semantics of expressions. For example, if variables v and w satisfy $\Gamma(v) \not\sqsubseteq \Gamma(w)$ then the program $w := v - v$ does not satisfy Θ_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 for programs to be speciously rejected is that Θ_S-Safe Assignment Statements ignores context. Program (9.21) is rejected due to `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_{L} for any initial value of x_{H}. A different effect of context is seen in program (9.22), where an assignment statement $x_{\mathsf{L}} := x_{\mathsf{H}}$ that does not satisfy Θ_S-Safe Assignment Statements is followed by an assignment statement $x_{\mathsf{L}} := 63$ that overwrites the illicit update.

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

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

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

Figure 9.8: Typing rules for TINI compliance

9.4.3 Enforcing TINI with Typing Rules

A type-safe programming language will have some *typing rules* that derive the set of *type-correct* programs. The typing rules 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*

$$\Gamma, \gamma \vdash_{\mathsf{ti}} S \tag{9.27}$$

where *typing context* Γ is a label assignment, and *control context* γ is a label from Λ.[9] 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 with variables $Vars(S)$ is *valid* if and only if

(i) $(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\to}_{\mathsf{ti}} V_{\sqsubseteq\lambda})$.

(ii) $\gamma \sqsubseteq \Gamma(w)$ holds for target w of every assignment statement in S. □

Requirement (i) supports our goal of having type-safe programs comply with TINI. Requirement (ii) makes γ an upper bound for the labels on guards of control-flow statements having bodies that could include S without violating Θ_S-Safe Assignment Statements. So requirement (ii) allows valid judgements for a compound statement to be derived from valid judgements for its component statements. The derivation given below for judgement (9.28) will illustrate.

[9]Consistent with the IMP syntax given in Figure 9.6, "statement" and "program" are used interchangeably in the following discussions.

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

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

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 9.8 is 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.

By design, the typing rules ensure that, in a type-correct program, no assignment statement violates Θ_S-Safe Assignment Statements condition (9.26). Analyzing an example is a good way to see how the rules check for such violations. Consider the following possible conclusion of rule IF, where S' denotes an IMP statement.

$$\Gamma, \bot_\Lambda \;\; \vdash_{\mathsf{ti}} \;\; S\colon \texttt{if}\, B\, \texttt{then}\;\; \ell\colon w := \mathit{expr}\;\; \texttt{else}\;\; S'\;\; \texttt{fi} \tag{9.28}$$

To derive this judgement requires having a derivation for each hypothesis of rule IF. The second hypothesis requires a derivation of the following.

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

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

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

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

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

So we have showed that the derivation of (9.28) requires:

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

This is exactly what Θ_S-Safe Assignment Statements condition (9.26) requires for assignment statement $\ell\colon\; w := \mathit{expr}$, since whether ℓ executes is affected by guard B of the `if` statement (and by no other guards).

1.	$\Gamma_{\mathcal{E}}(0) = \mathsf{L}$	... defn (9.9) 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 9.2.
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 (9.9) 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 9.2.
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 (9.9) 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.	$\Gamma, \mathsf{L} \vdash_{\mathsf{ti}}$ `if` $x \le y$ `then` $m := 0$ `else` $m := y$ `fi`	... IF with 8, 4, 7.

Figure 9.9: Example of Hilbert-style proof format

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

$$\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if } x \le y \texttt{ then } m := 0 \texttt{ else } m := y \texttt{ fi} \tag{9.29}$$

assuming $\Gamma(x) = \mathsf{L}$, $\Gamma(y) = \mathsf{H}$, and $\Gamma(m) = \mathsf{H}$ hold, Λ is $\{\mathsf{L}, \mathsf{H}\}$, and the rules for evaluating expressions involving $\sqsubseteq$ and $\sqcup$ are those given in Figure 9.2.

Hilbert-Style Proof Format. Figure 9.9 gives a type-correctness derivation as a list of sequentially numbered *steps*. Each step comprises a formula F (often, a judgement) and a justification J. When F is a judgement, J names a typing rule R and lists the numbers for earlier steps that discharge hypotheses needed to derive F by using rule R. 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 9.9.

Derivation-Tree Proof Format. Figure 9.10 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 9.10. Tags (DT1 and DT2) on the first two derivation trees allow their conclusions to be used for 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 9.11. Here, justifications for the hypotheses needed to

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

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

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

Figure 9.10: Example of derivation tree proof format

1. $\Gamma, \mathsf{L} \vdash_{\mathsf{ti}}$ `if` $x \le y$ `then` $m := x$ `else` $m := y$ `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 (9.9) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\perp_{\{\mathsf{L},\mathsf{H}\}} = \mathsf{L}$.
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 9.2.
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 9.2.

Figure 9.11: Example of hierarchically presented proof format

infer the judgement of step n are listed after that step, 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, but without the distraction of how each of those steps is being justified. The format also enables the reader to focus on the reasoning used to support the justification for any given step.

9.4.4 Dynamic Enforcement of TINI

A *reference monitor*[10] is a component that 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,by definition, be

[10]Chapter 11 gives a detailed treatment of reference monitors.

made by the reference monitor without having 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. Given the Batch and Asynchronous assumptions (page 253), blocked executions are then 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 these 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 Θ_S-Safe Assignment Statements condition (9.26). This suggests that a reference monitor for enforcing TINI should check this condition whenever an assignment statement is about to execute. To perform this check for an assignment statement $\ell\colon w := \mathit{expr}$ in some monitored program S, the reference monitor needs labels $\Gamma(w)$, $\Gamma_{\mathcal{E}}(\mathit{expr})$, 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 $\Gamma_{\mathcal{E}}(\mathit{expr})$ can be determined by the reference monitor if (i) reaching an assignment statement $\ell\colon w := \mathit{expr}$ 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) the code for each of these events (`if` G, `fi`, `while` G, or `end`) is available to the reference monitor when that event occurs.

Figure 9.12 gives the actions for such a reference monitor $\mathcal{R}_{TI}$. A `require(`B`)` statement is used there. Execution of `require(`B`)` evaluates B. If B evaluates to *false* then the reference monitor deletes the state and blocks further progress of the monitored program that was being executed when the reference monitor was invoked; if B evaluates to *true* then the monitored program is allowed to proceed.

$\mathcal{R}_{TI}$ is invoked and checks Θ_S-Safe Assignment Statements condition (9.26) whenever an assignment statement is reached in monitored program S. To facilitate this checking, $\mathcal{R}_{TI}$ also is invoked so that it can update a stack[11] *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

[11]We use operations `push(`*sti*, v`)` to insert value v onto stack *sti*; `pop(`*sti*`)` to remove the most recently added value from stack *sti*; and a function `top(`*sti*`)` that returns the value currently at the top of stack*sti*.

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

Figure 9.12: Reference monitor $\mathcal{R}_{TI}$ for TINI

program S.) The updates to this stack ensure that

$$\mathtt{top}(\mathit{sti}) = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \tag{9.30}$$

holds whenever an assignment statement (say) $\ell\colon w := \mathit{expr}$ is about to execute in the monitored program. Therefore, the value of $\mathtt{top}(\mathit{sti})$ along with $\Gamma(w)$ and $\Gamma_{\mathcal{E}}(\mathit{expr})$, can be used by the reference monitor to check Θ_S-Safe Assignment Statements condition (9.26) 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 $\mathcal{R}_{TI} \triangleright S$. By construction, terminating executions of $\mathcal{R}_{TI} \triangleright S$ are terminating executions of S where Θ_S-Safe Assignment Statements condition (9.26) holds for every assignment statement that was executed.

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

$$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{\mathcal{R}_{TI} \triangleright S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}) \tag{9.31}$$

If (9.31) did not hold then, according to definition (9.16) of $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$, 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 $\mathcal{R}_{TI} \triangleright S$ 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 Θ_S-Safe Assignment Statements condition (9.26), so execution would be blocked before performing that assignment statement, which contradicts the assumption that we started with terminating executions.

Note, however, that if S does not satisfy TINI then there must be terminating executions of S that become blocked executions of $\mathcal{R}_{TI} \triangleright S$. The initial states of this smaller set of terminating executions for $\mathcal{R}_{TI} \triangleright S$ exhibits additional correlations over the initial states of the terminating executions for S without $\mathcal{R}_{TI}$ present. 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.

Here is an example. Assume that $\Gamma(x_{\mathsf{L}}) = \mathsf{L}$ and $\Gamma(x_{\mathsf{H}}) = \mathsf{H}$ hold.

$$\texttt{if } \textit{even}(x_{\mathsf{H}}) \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else skip fi} \qquad (9.32)$$

$\mathcal{R}_{TI}$ only allows terminating executions of (9.32) that start in states where $\textit{even}(x_{\mathsf{H}})$ is *false*; other initial states result in blocked executions. So an L-observer of a terminating execution of (9.32) when $\mathcal{R}_{TI}$ is present learns something about the initial value of x_{H}—that initially x_{H} was odd. TINI is being enforced by $\mathcal{R}_{TI}$, though. This is because (i) differences in the initial values for x_{H} in terminating executions are not visible to an L-observer reading x_{L} when execution terminates, and (ii) an L-observer cannot detect that an execution is blocked and, therefore, cannot determine that the initial value of x_{H} is even.

9.4.5 Comparison of TINI Enforcement Mechanisms

Type-correctness checking 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 a context switch to execute each reference monitor action. 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).

The key difference, however, between type-correctness and TINI enforcement using a reference monitor is permissiveness. Type-checking rejects any program S in which there is 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 (9.32) illustrates. Program (9.32) 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 $\textit{even}(x_{\mathsf{H}})$ 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 a set of typing rules are together defining an algorithm that a type-checker can execute, we conclude that inferences about statement reachability cannot be incorporated into typing rules.

9.5 Trusted Code and Weaker Policies

Public outputs from many real systems are affected by secret inputs that, nevertheless, are not revealed. Examples of such public outputs include encryption of a secret for transmission or storage, redaction[12] of a document for widespread disclosure, and transmission of an acknowledgement message to confirm receipt of a request involving secret values. Real 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. We conclude that noninterference policies may need to be relaxed in parts of a system.

To avoid these problems, many systems incorporate statements or routines that, by fiat, are allowed to violate noninterference. This is variously known 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 any other variable. 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, however, is to enforce a security policy that is not as stringent as noninterference. We would build on (i) the weaker properties that the trusted code satisfies, and (ii) leverage a noninterference policy being enforced for all of the other code.

***Example: Uncertainty-Based Confidentiality.** An example of such a weaker security policy is *uncertainty-based confidentiality*. It asserts that an observer remains uncertain about the values of some specified variables because so many values remain plausible—despite the observer's certainty about the values of other variables:

> **Uncertainty-Based Confidentiality.** The values of variables with labels λ' satisfying $\lambda' \sqsubseteq \lambda$ leaves sufficient ambiguity about the initial values of variables with labels λ'' satisfying $\lambda \not\sqsubseteq \lambda''$. □

Uncertainty-Based Confidentiality does not prohibit the values of variables with labels λ' satisfying $\lambda' \sqsubseteq \lambda$ from being affected by the values of variables with labels λ'' satisfying $\lambda \not\sqsubseteq \lambda''$. So it is weaker than noninterference. But Uncertainty-Based Confidentiality does protect against disclosure of the values of variables with labels λ'' satisfying $\lambda \not\sqsubseteq \lambda''$. Moreover, the confidentiality examples at the beginning of this subsection—encryption, redaction, and transmission of

[12] *Redaction* deletes or obscures parts of a document, producing a version that complies with a given confidentiality policy.

acknowledgments—do not violate Uncertainty-Based Confidentiality, whereas they do violate noninterference.

We illustrate with an implementation of a secret ballot election. Each voter i stores into a ballot b_i the name of some candidate from a set C, and the winner m of the election is the candidate named in a majority[13] of the ballots:

$$S\colon\quad m := maj(b_1, \ldots, b_n) \tag{9.33}$$

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 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 an uncertainty-based confidentiality policy where, for an initial system state s and each candidate $c \in C$, there will be an initial system state s' satisfying $b_j = c$ and indistinguishable to voter i from s.

$$\begin{aligned}&(\forall i,\, j,\, i \neq j\colon\ (\forall c \in C\colon\\ &\qquad (\forall s\colon\ (\exists s'\colon\ s =_{\{b_i\}} s' \ \wedge\ s'.b_j = c \ \wedge\ [\![S]\!](s) =_{\{b_i, m\}} [\![S]\!](s')\ ))))\end{aligned} \tag{9.34}$$

To establish compliance with (9.34), it suffices to give an expression $\mathcal{SK}(i,j,c,s)$ for producing a state that substituted for s' satisfies[14]

$$s =_{\{b_i\}} s' \quad \wedge \quad s'.b_j = c \quad \wedge \quad [\![S]\!](s) =_{\{b_i, m\}} [\![S]\!](s'). \tag{9.35}$$

Expression $\mathcal{SK}(i,j,c,s)$ generates states that serve as witnesses 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 $b_j = c$ holds, for any candidate c. Therefore, no possible value of b_j is ruled out by what voter i can read.

Here is a proposed definition for $\mathcal{SK}(i,j,c,s)$.

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

If there are 2 candidates and at least 5 voters, it is straightforward to establish that (9.35) is satisfied when s' is replaced by this definition for $\mathcal{SK}(i,j,c,s)$.[15]

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

[14]We are proving an existentially-quantified formula $(\exists x\colon\ P(x))$ 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: $\frac{P(E)}{(\exists x\colon\ P(x))}$. Expression E is called a *Skolem function.*

[15]We show that each conjunct of (9.35) 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 $n \ge 5$ 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.

But (9.35) 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 (9.35) for such elections—with so few voters, knowing the values of b_i and m can reduce a voter i's uncertainty about b_j. Here is a problematic scenerio. 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

$$s\colon\ [m \mapsto ?,\ b_1 \mapsto c_1,\ b_2 \mapsto c_2,\ b_3 \mapsto c_2],$$

so c_2 is the majority. $\mathcal{SK}(i, j, c, s)$ would have to produce a state where $b_3 = c_1$ holds and the majority remains c_2. However, no values for the b_k, where $1 \le k \le 3$, $k \ne 1$, $k \ne 3$ hold (*viz.* 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 (9.35). The requirement for at least 5 voters when there are 2 candidates is often surprising to people who have used informal reasoning and ignored edge cases. There is a lesson about the use of informal assurance arguments for trusted code.

Notes and Reading for Chapter 9

Dorothy Denning was the first to suggest that security policies ought to specify restrictions on information flow rather than specifying restrictions on access to information containers. The approach was summarized in two papers [7, 10], which are based on her Ph.D. dissertation [6]; an interview [9] with Denning explores what motivated and influenced this work. Denning's dissertation introduces the terms "explicit flow" and "implicit flow" for distinguishing information flows caused by control structures.[16] 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 [8, chptr 5] does give a formal definition. That definition is formulated in terms of entropy as defined by Shannon [28] 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.[17] Often, Gougen and Messequer [12] will be cited, because that paper introduced and formalized *noninterference assertions*, which assert that actions performed by one group of users do not affect outputs seen by

[16]Denning was not the first and not the only researcher to have investigated information flows arising from control structures. Fenton [11] had earlier discussed how to prevent such information flows in connection with implementing memoryless subsystems. Also presented at the SOSP conference where Denning gave a preliminary version of her paper [7], Jones and Lipton [13] uses the term "negative inference" to describe leaks caused by control structures.

[17]Alternatives that have been suggested, include constraints [20], non-deducibility [29], generalized non-interference [17], restrictiveness [18], selective interleavings [19], trace closure properties [32], and the modular assembly kit [15, 16]. None has attracted a large following.

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 [12] was not the first paper to suggest such a counterfactual definition, though. Cohen [5] had previously introduced *strong dependency*, which defined information flow from x to y as variation in x that results in variation in y. However, the theory given in Cohen [5] uses inscrutable formalisms, 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 [6]. Volpano, Smith, and Irvine combine these pieces in a paper [30] that, for programs having a fixed label assignment, gives typing rules to enforce what Sabelfeld and Sands [27] later call termination-insensitive noninterference (TINI). The soundness proof in Volpano, Smith, and Irvine [30] 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 [26] explores the differences in permissiveness and shows that a reference monitor like $\mathcal{R}_{TI}$ not only enforces TINI but is more permissive than a type system. However, reference monitors do not always lead to increased permissiveness, as seen in Chapter 10.

The development and implementation of secure ways to store classified documents in computer systems, however, predated and proceeded independently of research into specifying and enforcing TINI and other information flow policies. In anticipation of a day when documents would be stored in computers shared by users having different clearances, DoD interest centered on operating systems and supporting appropriate access control to files. Multilevel security labels were developed to replicate the (paper) document-classification system that was already in use by DoD.[18]

[18] The current U.S. document-classification scheme is described in Executive Order 13526 [21] signed by President Barack Obama in December 2009. It is the most recent in a series of Executive Orders concerned with U.S. document classification, starting with Executive Order 8381 [25] signed by President Roosevelt in March 1940 [24, chapter 3]. Quist [24, chapter 2], drawing heavily from an unpublished manuscript by Patterson [23], chronicles the precursors and development of U.S. document-classification schemes, which were derived from the British circa 1917. In fact, by late in the 19th century, Britain had all of the elements for a modern document-classification scheme in place. Prior to the Crimean War (1853–1856), the British War Office had been marking documents that should be kept confidential, and by 1894 British Army regulations were distinguishing between markings "Secret" and "Confidential" each of which imposed specific rules for handling and disclosure. An early version of "need-to-know" appears in an 1868 publication of British Army regulations:

> Access to official records is only permitted to those who are entrusted with the duties of the office or department to which they belong...

Peacetime classification of information in Britain commenced with an 1866 report on mines

A scheme published in 1973 by Bell and La Padula [2, 1] working at MITRE became the basis for virtually all DoD computer security work for the next decade.[19] Biba [4] later shows how multilevel security labels used by Bell and La Padula for confidentiality could be reinterpreted for specifying and enforcing integrity policies. An attempt to implement a secure version of the Multics [22] operating system established a need to further extend the theory. Bell and La Padula [3] adds trusted subjects; Walter et al. [31] is credited with developing the special access rules needed for tree-structured directories. (Directories in Multics have a specific semantics and thus warranted special treatment.) Landwehr [14] positions the Bell and La Padula work relative to other (early) formal models for computer security.

Bibliography

[1] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: A mathematical model. Technical Report ESD-TR-73-278, Volume II, Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, November 1973.

[2] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Volume I, Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, November 1973.

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

[4] K. J. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, June 1975.

[5] 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.

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

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

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

and torpedoes—new technologies that, if known by an adversary, might disrupt the Royal Navy's superiority.

[19]A group of researchers at Case Western University independently developed essentially the same restrictions, but couched in terms of repositories and agents rather than files and processes. Their paper [31], however, is rarely cited.

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

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

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

[12] 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.

[13] 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.

[14] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Barack Obama. Classfied national security information. Executive Order EO 13526, The White House, December 2009. https://www.fas.org/irp/offdocs/eo/eo-13526.htm.

[22] Elliott I. Organick. *The Multics System: An Examination of its Structure.* MIT Press, 1972.

[23] Andrew Patterson, Jr. "CONFIDENTIAL" – The beginning of defense-information marking. Unpublished manuscript. Sterling Chemistry Laboratory, Yale University, April 1980.

[24] Arvin S. Quist. Security Classification of Information, Volume 1. Introduction, History, and Adverse Impacts. Technical Report ORCA–12, Oak Ridge Classification Associates, LLC, September 2002. http://www.fas.org/sgp/library/quist/index.html.

[25] Franklin D. Roosevelt. Defining certain vital military and naval installations and equipment. Executive Order EO 8381, The White House, March 1940. https://www.fas.org/irp/offdocs/eo/eo-8381.htm.

[26] 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.

[27] 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 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.

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

[29] 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.

[30] 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.

[31] K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, and D. G. Shumway. Primitive models for computer security. Interim Technical Report ESD-TR-4-117, Case Western Reserve University, 1974. NTIS AD-778 467.

[32] 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.