5

Information Flow Controls

Access controls regulate the accessing of objects, but not what subjects might do with the information contained in them. Many difficulties with information "leakage" arise not from defective access control, but from the lack of any policy about information flow. Flow controls are concerned with the right of **dissemination of information,** irrespective of what object holds the information; they specify valid channels along which information may flow.

5.1 LATTICE MODEL OF INFORMATION FLOW

We shall describe flow controls using the lattice model introduced by Denning [Denn75,Denn76a]. The lattice model is an extension of the Bell and LaPadula [Bell73] model, which describes the security policies of military systems (see Section 5.6).

The lattice model was introduced to describe policies and channels of information flow, but not what it means for information to flow from one object to another. We shall extend the model to give a precise definition of information flow in terms of classical information theory.

An **information flow system** is modeled by a lattice-structured flow policy, states, and state transitions.

5.1.1 Information Flow Policy

An information flow policy is defined by a lattice $(SC, \preceq)$, where SC is a finite set of **security classes**, and $\preceq$ is a binary relation† partially ordering the classes of

† In [Denn76a], the notation "$\rightarrow$" is used to denote the relation "$\preceq$".

SC. The security classes correspond to disjoint classes of information; they are intended to encompass, but are not limited to, the familiar concepts of "security classifications" and "security categories" [Weis69,Gain72].

For security classes A and B, the relation $A \leq B$ means class A information is lower than or equal to class B information. Information is permitted to flow within a class or upward, but not downward or to unrelated classes; thus, class A information is permitted to flow into class B if and only if $A \leq B$. There is a lowest class, denoted *Low*, such that $Low \leq A$ for all classes A in SC. *Low* security information is permitted to flow anywhere. Similarly, there is a highest class, denoted *High*, such that $A \leq High$ for all A. *High* security information cannot leave the class *High*. The lattice properties of $(SC, \leq)$ are discussed in Sections 5.1.4 and 5.1.5.

Example:
The simplest flow policy specifies just two classes of information: confidential (*High*) and nonconfidential (*Low*); thus, all flows except those from confidential to nonconfidential objects are allowed. This policy specifies the requirements for a **selectively confined** service program that handles both confidential and nonconfidential data [Denn74,Fent74]. The service program is allowed to retain the customer's nonconfidential data, but it is not allowed to retain or leak any confidential data. An income tax computing service, for example, might be allowed to retain a customer's address and the bill for services rendered, but not the customer's income or deductions. This policy is enforced with flow controls that assign all outputs of the service program to class *Low*, except for the results returned to the customer. ■

Example:
The **multilevel security** policy for government and military systems represents each security class by a pair (A, C), where A denotes an **authority level** and C a **category**. There are four authority levels:

0—*Unclassified*
1—*Confidential*
2—*Secret*
3—*Top Secret* .

There are 2^m categories, comprising all possible combinations of m compartments for some m; examples of compartments might be *Atomic* and *Nuclear*. Given classes (A, C) and (A', C'), $(A, C) \leq (A', C')$ if and only if $A \leq A'$ and $C \subseteq C'$. Transmissions from (2, {*Atomic*}) to (2, {*Atomic, Nuclear*}) or to (3, {*Atomic*}) are permitted, for example, but those from (2, {*Atomic*}) to (1, {*Atomic*}) or to (3, {*Nuclear*}) are not. ■

5.1.2 Information State

The **information state** of a system is described by the value and security class of each object in the system. An object may be a logical structure such as a file,

record, field within a record, or program variable; or it may be a physical structure such as a memory location, register (including an address or instruction register), or a user. For an object x, we shall write "x" for both the name and value of x (the correct interpretation should be clear from context), and $\underline{x}$ (x with an underbar) for its security class. When we want to specify the value and class of x in some particular state s, we shall write x_s and $\underline{x}_s$, respectively. We shall write simply x and $\underline{x}$ when the state is clear from context or unimportant to the discussion.

The class of an object may be either constant or varying. With **fixed** or **constant** classes (also called "static binding"), the class of an object x is constant over the lifetime of x; that is, $\underline{x}_s = \underline{x}_{s'}$ for all states s and s' that include x. With **variable** classes (also called "dynamic binding"), the class of an object x varies with its contents; that is, $\underline{x}_s$ depends on x_s. A flow control mechanism could support both fixed and variable classes—for example, fixed classes for permanent or global objects, and variable ones for temporary or local ones. Users are assigned fixed classes called "security clearances". Unless explicitly stated otherwise, all objects have fixed security classes.

Given objects x and y, a flow from x to y is **authorized** (permitted) by a flow policy if and only if $\underline{x} \leq \underline{y}$; if y has a variable class, then $\underline{y}$ is its class after the flow.

5.1.3 State Transitions and Information Flow

State transitions are modeled by operations that create and delete objects, and operations that change the value or security class of an object. Information flows are always associated with operations that change the value of an object. For example, execution of the file copy operation "*copy*(*F1*, *F2*)" causes information to flow from file *F1* to file *F2*. Execution of the assignment statement "$y := x/1000$" causes some information to flow from x to y, but less than the assignment "$y := x$".

To determine which operations cause information flow, and the amount of information they transfer, we turn to information theory [see Section 1.4.1; in particular, Eq. (1.1) and (1.2)]. Let s' be the state that results from execution of a command sequence α in state s, written

$$s \vdash_\alpha s'$$

(read "s derives s' under α"). Given an object x in s and an object y in s', let $H_{y'}(x)$ be the equivocation (conditional entropy) of x_s given $y_{s'}$, and let $H_y(x)$ be the equivocation of x_s given the value y_s of y in state s; if y does not exist in state s, then $H_y(x) = H(x)$, where $H(x)$ is the entropy (uncertainty) of x. Execution of α in state s causes **information flow** from x to y, denoted

$$x_s \rightarrow_\alpha y_{s'} ,$$

if new information about x_s can be determined from $y_{s'}$; that is, if

$$H_{y'}(x) < H_y(x) .$$

We shall write $x \rightarrow_\alpha y$, or simply $x \rightarrow y$, if there exist states s and s' such that

execution of command sequence α causes a flow $x \rightarrow y$. Cohen's [Cohe77,Cohe78] definition of "strong dependency" and Millen's [Mill78] and Furtek's [Furt78] deductive formulations of information flow are similar to this definition, but they do not account for the probability distribution of the values of variables as provided by Shannon's information theory.

A flow $x_s \rightarrow_\alpha y_{s'}$ is authorized if and only if $\underline{x}_s \leq \underline{y}_{s'}$; that is, the final class of y is at least as great as the initial class of x.

The **amount of information** (in bits) transferred by a flow $x_s \rightarrow_\alpha y_{s'}$ is measured by the reduction in the uncertainty about x:

$$I(\alpha, x, y, s, s') = H_y(x) - H_{y'}(x) .$$

Letting $P_s(x)$ denote the probability distribution of x in state s, the **channel capacity** of $x \rightarrow_\alpha y$ is defined by the maximum amount of information transferred over all possible probability distributions of x:

$$C(\alpha, x, y) = \max_{P_s(x)} I(\alpha, x, y, s, s')$$

Note that if y can be represented with n bits, the channel capacity of any command sequence transferring information to y can be at most n bits. This means that if the value of y is derived from m inputs $x_1, \ldots, x_m$, on the average only m/n bits can be transferred from each input.

The following examples illustrate how the previous definitions are applied.

Example:
Consider the assignment

$$y := x .$$

Suppose x is an integer variable in the range [0, 15], with all values equally likely. Letting p_i denote the probability that $x = i$, we have

$$p_i = \begin{cases} \dfrac{1}{16} & 0 \leq i \leq 15 \\ 0 & \text{otherwise} . \end{cases}$$

If y is initially null, then

$$H_y(x) = H(x) = \sum_i p_i \log_2\left(\frac{1}{p_i}\right) = 16\left(\frac{1}{16}\right) \log_2 16 = 4 .$$

Because the exact value of x can be determined from y after the statement is executed, $H_{y'}(x) = 0$. Thus, 4 bits of information are transferred to y.

The capacity of an assignment "$y := x$" is usually much greater than 4 bits. If x and y are represented as 32-bit words, for example, the capacity is 32 bits. Or, if x and y are vectors of length n, where each vector element is a 32-bit word, the capacity is $32n$. In both cases, the capacity is achieved when all possible words are equally likely.

Note that the statement "$y := x$" does not cause a flow $x \rightarrow y$ if the value of x is known. This is because $H_{y'}(x) = H_y(x) = H(x) = 0$. ■

Example:
Consider the sequence of statements

$$z := x;$$
$$y := z .$$

Execution of this sequence can cause an "indirect flow" $x \rightarrow y$ through the intermediate variable z, as well as a direct flow $x \rightarrow z$. Note, however, that the sequence does not cause a flow $z \rightarrow y$, because the final value of y does not reveal any information about the initial value of z. ■

Example:
Consider the statement

$$z := x + y .$$

Let x and y be in the range [0, 15] with all values equally likely; thus $H(x) = H(y) = 4$ bits. Given z, the values of x and y are no longer equally likely (e.g., $z = 0$ implies both x and y must be 0, $z = 1$ implies either $x = 0$ and $y = 1$ or $x = 1$ and $y = 0$). Thus, both $H_{z'}(x)$ and $H_{z'}(y)$ are less than 4 bits, and execution of the statement reduces the uncertainty about both x and y.

Now, little can be deduced about the values of x and y from their sum when both values are unknown. Yet if one of the elements is known, the other element can be determined exactly. In general, the sum $z = x_1 + \ldots + x_n$ of n elements contains some information about the individual elements. Given additional information about some of the elements, it is often possible to deduce the unknown elements (e.g., given the sum $z1$ of $n - 1$ elements, the nth element can be determined exactly from the difference $z - z1$). The problem of hiding confidential information in sums (and other statistics) is studied in the next chapter. ■

Example:
It may seem surprising that the syntactically similar statement

$$z := x \oplus y ,$$

where x and y are as described in the previous example and "$\oplus$" denotes the exclusive-or operator, does not cause information to flow to z. This is because the value of z does not reduce the uncertainty about either x or y (all values of x and y are equally likely for any given z). This is a Vernam cipher, where x is the plaintext, y is the key, and z is the ciphertext. This example shows it is not enough for an object y to be functionally dependent on an object x for there to be a flow $x \rightarrow y$. It must be possible to learn something new about x from y.

We saw in Chapter 1 that most ciphers are theoretically breakable given enough ciphertext; thus, encryption generally causes information to flow from a plaintext message M to a ciphertext message $C = E_K(M)$. Determining M from C may be computationally infeasible, however, whence the information in C is not practically useful. Thus, in practice high security

information can be encrypted and transmitted over a low security channel without violating a flow policy. ■

Example:
Consider the **if** statement

if $x = 1$ **then** $y := 1$,

where y is initially 0. Suppose x is 0 or 1, with both values equally likely; thus $H(x) = 1$. After this statement is executed, y contains the exact value of x, giving $H_y(x) = 0$. Thus, 1 bit of information is transferred from x to y. Even if x is not restricted to the range [0, 1], the value of y reduces the uncertainty about x ($y = 1$ implies $x = 1$ and $y = 0$ implies $x \neq 1$).

The flow $x \rightarrow y$ caused by executing the **if** statement is called an **implicit flow** to distinguish it from an **explicit flow** caused by an assignment. The interesting aspect of an implicit flow $x \rightarrow y$ is that it can occur even in the absence of any explicit assignment to y. For the preceding **if** statement, the flow occurs even when $x \neq 1$ and the assignment to y is skipped. There must be a possibility of executing an assignment to y, however, and this assignment must be conditioned on the value of x; otherwise, the value of y cannot reduce the uncertainty about x.

The information in an implicit flow is encoded in the program counter (instruction register) of a process when it executes a conditional branch instruction. This information remains in the program counter as long as the execution path of the program depends on the outcome of the test, but is lost thereafter. ■

Example:
Consider the statement

if $(x = 1)$ and $(y = 1)$ **then** $z := 1$,

where z is initially 0. Suppose x and y are both 0 or 1, with both values equally likely; thus $H(x) = H(y) = 1$. Execution of this statement transfers information about both x and y to z ($z = 1$ implies $x = y = 1$, and $z = 0$ implies $x = 0$ with probability 2/3 and $x = 1$ with probability 1/3; similarly for y). The equivocations $H_{z'}(x)$ and $H_{z'}(y)$ are both approximately .7 (derivation of this is left as an exercise). Thus, the amount of information transferred about each of x and y is approximately .3 bit, and the total amount of information transferred is about .6 bit.

Rewriting the statement as a nested conditional structure

if $x = 1$
  then if $y = 1$ **then** $z := 1$

shows that implicit flows can be transferred through several layers of nesting. ■

Example:
Consider the **if** statement

$$\textbf{if } x \geq 8 \textbf{ then } y := 1 \ ,$$

where y is initially 0. Again suppose x is an integer variable in the range [0, 15], with all values equally likely, so $H(x) = 4$. To derive the equivocation $H_{y'}(x)$ from executing this statement, let q_j be the probability $y' = j$ ($j = 0$ or 1), and let $q_j(i)$ be the probability $x = i$ given $y' = j$. Then

$$q_0 = q_1 = \frac{1}{2}$$

$$q_0(i) = \begin{cases} \frac{1}{8} & 0 \leq i \leq 7 \\ 0 & \text{otherwise} \end{cases}$$

$$q_1(i) = \begin{cases} \frac{1}{8} & 8 \leq i \leq 15 \\ 0 & \text{otherwise} \end{cases}$$

and

$$H_{y'}(x) = \sum_{j=0}^{1} q_j \sum_{i=0}^{15} q_j(i) \log_2\left(\frac{1}{q_j(i)}\right)$$

$$= \left(\frac{1}{2}\right)\left[8\left(\frac{1}{8}\right)\log_2 8\right] + \left(\frac{1}{2}\right)\left[8\left(\frac{1}{8}\right)\log_2 8\right]$$

$$= \left(\frac{1}{2}\right) 3 + \left(\frac{1}{2}\right) 3 = 3 \ .$$

Execution of this statement, therefore, transfers 1 bit of information about x to y (namely, the high-order bit).

Suppose instead x has the following distribution:

$$p_i = \begin{cases} \frac{1}{16} & 0 \leq i \leq 7 \\ \frac{1}{2} & i = 8 \\ 0 & \text{otherwise} \ . \end{cases}$$

The uncertainty of x is:

$$H(x) = 8\left(\frac{1}{16}\right)\log_2 16 + \left(\frac{1}{2}\right)\log_2 2 = 2.0 + 0.5 = 2.5 \ .$$

The probability distributions of y and of x given y are:

$$q_0 = q_1 = \frac{1}{2}$$

$$q_0(i) = \begin{cases} \frac{1}{8} & 0 \le i \le 7 \\ 0 & \text{otherwise} \end{cases}$$

$$q_1(i) = \begin{cases} 1 & i = 8 \\ 0 & \text{otherwise} \end{cases} .$$

Therefore,

$$H_{y'}(x) = \left(\frac{1}{2}\right) 8 \left(\frac{1}{8}\right) \log_2 8 + \left(\frac{1}{2}\right) \log_2 1 = \left(\frac{1}{2}\right) 3 + \left(\frac{1}{2}\right) 0 = 1.5 .$$

Again, 1 bit of information is transferred. This is because y is assigned the value 1 with probability $1/2$. It may seem that more than 1 bit of information can be transferred by this statement, because when y is assigned the value 1, there is no uncertainty about x—it must be 8. On the other hand, when y is assigned the value 0, there is still uncertainty about the exact value of x—it could be anything between 0 and 7. The equivocation $H_{y'}(x)$ measures the expected uncertainty of x over all possible assignments to y. ■

In general, an **if** statement of the form

if $f(x)$ **then** $y := 1$

for some function f transfers 1 bit when the probability $f(x)$ is true is $1/2$. Let p_i be the probability $x = i$ for all possible values i of x. Assuming y is initially 0 and $H_y(x) = H(x)$, we have

$$q_0 = q_1 = \frac{1}{2}$$

$$q_0(i) = \begin{cases} 2p_i & \text{if } f(i) \text{ is false} \\ 0 & \text{otherwise} \end{cases}$$

$$q_1(i) = \begin{cases} 2p_i & \text{if } f(i) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

whence:

$$H_{y'}(x) = \left(\frac{1}{2}\right)\sum_i q_0(i) \log_2 \left(\frac{1}{q_0(i)}\right) + \left(\frac{1}{2}\right)\sum_i q_1(i) \log_2 \left(\frac{1}{q_1(i)}\right)$$

$$= \left(\frac{1}{2}\right)\sum_i 2p_i \log_2 \left(\frac{1}{2p_i}\right) = \sum_i p_i \log_2 \left(\frac{1}{2p_i}\right)$$

$$= \sum_i p_i \log_2 \left(\frac{1}{p_i}\right) - \sum_i p_i \log_2 2$$

$$= H(x) - 1 .$$

If the probability $f(x)$ is true is not $1/2$, less than 1 bit of information will be transferred (see exercises at end of chapter).

It should now come as no surprise that the channel capacity of the statement

if $f(x)$ **then** $y := 1$

is at most 1 bit. The reason is simple: because y can be represented by 1 bit (0 or 1), it cannot contain more than 1 bit of information about x.

5.1.4 Lattice Structure

A flow policy $(SC, \preceq)$ is a **lattice** if it is a partially ordered set (poset) and there exist least upper and greatest lower bound operators, denoted $\oplus$ and $\otimes$ respectively†, on SC (e.g., see [Birk67]). That $(SC, \preceq)$ is a **poset** implies the relation $\preceq$ is reflexive, transitive, and antisymmetric; that is, for all A, B, and C in SC:

1. Reflexive: $A \preceq A$
2. Transitive: $A \preceq B$ and $B \preceq C$ implies $A \preceq C$
3. Antisymmetric: $A \preceq B$ and $B \preceq A$ implies $A = B$.

That $\oplus$ is a **least upper bound** operator on SC implies for each pair of classes A and B in SC, there exists a unique class $C = A \oplus B$ in SC such that:

1. $A \preceq C$ and $B \preceq C$, and
2. $A \preceq D$ and $B \preceq D$ implies $C \preceq D$ for all D in SC.

By extension, corresponding to any nonempty subset of classes $S = \{A_1, \ldots, A_n\}$ of SC, there is a unique element $\oplus S = A_1 \oplus A_2 \oplus \ldots \oplus A_n$ which is the least upper bound for the subset. The highest security class, *High*, is thus $High = \oplus SC$.

That $\otimes$ is a **greatest lower bound** operator on SC implies for each pair of classes A and B in SC, there exists a unique class $E = A \otimes B$ such that:

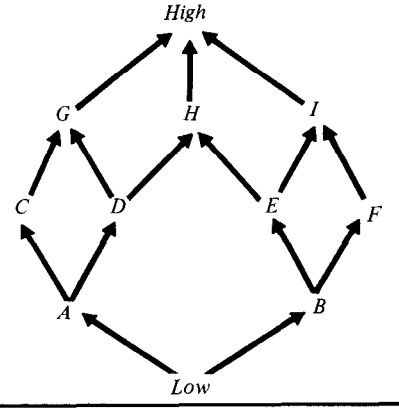
1. $E \preceq A$ and $E \preceq B$, and
2. $D \preceq A$ and $D \preceq B$ implies $D \preceq E$ for all D in SC.

By extension, corresponding to any subset $S = \{A_1, \ldots, A_n\}$ of SC, there is a unique element $\otimes S = A_1 \otimes A_2 \otimes \ldots \otimes A_n$ which is the greatest lower bound for the subset. The lowest security class, *Low*, is thus $Low = \otimes SC$. *Low* is an identity element on $\oplus$; that is, $A \oplus Low = A$ for all $A \in SC$. Similarly, *High* is an identity element on $\otimes$.

Example:
Figure 5.1 illustrates a lattice with 11 classes, where an arrow from a class X to a class Y means $X \preceq Y$. The graphical representation is a standard precedence graph showing only the nonreflexive, immediate relations. We have, for example,

† In the remainder of this chapter, "$\oplus$" denotes least upper bound rather than exclusive-or.

FIGURE 5.1 Lattice.

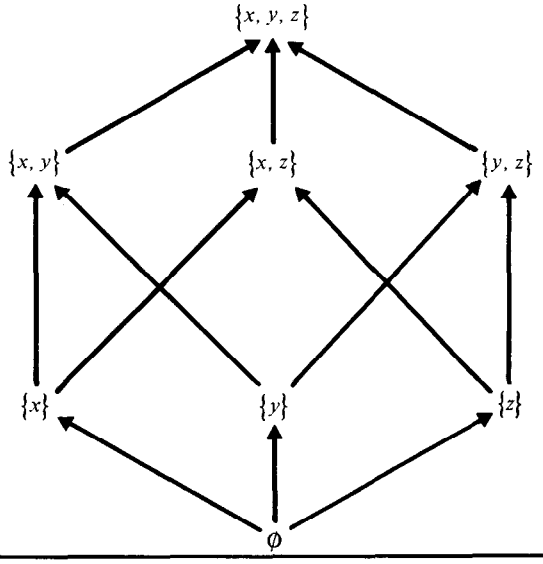


$A \oplus C = C, A \otimes C = A$
$C \oplus D = G, C \otimes D = A$
$C \oplus D \oplus E = High, C \otimes D \otimes E = Low.$ ■

A **linear lattice** is simply a linear ordering on a set of n classes $SC = \{0, 1, \ldots, n-1\}$ such that for all $i, j \in [0, n-1]$:

a. $i \oplus j = \max(i, j)$
b. $i \otimes j = \min(i, j)$
c. $Low = 0$; $High = n - 1$.

FIGURE 5.2 Subset lattice.



For $n = 2$, this lattice describes systems that distinguish confidential data in class 1 (*High*) from nonconfidential data in class 0 (*Low*). For $n = 4$, it describes the authority levels in military systems.

Given a set X, a **subset lattice** is derived from a nonlinear ordering on the set of all subsets of X. The ordering relation $\leq$ corresponds to set inclusion $\subseteq$, the least upper bound $\oplus$ to set union $\cup$, and the greatest lower bound $\otimes$ to set intersection $\cap$. The lowest class corresponds to the empty set and the highest class to X. Figure 5.2 illustrates for $X = \{x, y, z\}$.

A subset lattice is particularly useful for specifying arbitrary **input-output relations**. Suppose a program has m input parameters $x_1, \ldots, x_m$ and n output parameters $y_1, \ldots, y_n$ such that each output parameter is allowed to depend on only certain inputs [Jone75]. A subset lattice can be constructed from the subsets of $X = \{x_1, \ldots, x_m\}$. The class associated with each input x_i is the singleton set $\underline{x_i} = \{x_i\}$, and the class associated with each output y_j is the set $\underline{y_j} = \{x_i \mid x_i \rightarrow y_j \text{ is allowed}\}$.

Example:
Suppose a program has three input parameters x_1, x_2, and x_3, and two output parameters y_1 and y_2 subject to the constraint that y_1 may depend only on x_1 and x_2, and y_2 may depend only on x_1 and x_3. Then $\underline{y_1} = \{x_1, x_2\}$ and $\underline{y_2} = \{x_1, x_3\}$. ■

A subset lattice also describes policies for which X is a set of categories or properties, and classes are combinations of categories; information in an object a is allowed to flow into an object b if and only if b has at least the properties of a.

Example:
Consider a database containing medical, financial, and criminal records on individuals, and let $X = \{Medical, Financial, Criminal\}$. Medical information is permitted to flow into an object b if and only if $Medical \in \underline{b}$, and a combination of medical and financial information is permitted to flow into b if and only if both $Medical \in \underline{b}$ and $Financial \in \underline{b}$. ■

Another example is the set of security categories in military systems. Karger [Karg78] discusses the application of such lattices to the private sector and decentralized computer networks, where the security lattices may be large.

Still richer structures can be constructed from combinations of linear and subset lattices.

Example: Multilevel security.
The security classes of the military multilevel security policy (see Section 5.1.1) form a lattice determined by the (Cartesian) product of the linear lattice of authority levels and the subset lattice of categories. Let (A, C) and (A', C') be security classes, where A and A' are authority levels and C and C' are categories. Then

a. $(A, C) \leq (A', C')$ iff $A \leq A', C \subseteq C'$
b. $(A, C) \oplus (A', C') = (\max(A, A'), C \cup C')$
c. $(A, C) \otimes (A', C') = (\min(A, A'), C \cap C')$
d. $Low = (0, \{\}) = (Unclassified, \{\})$
e. $High = (3, X) = (Top\ Secret, X)$,

where X is the set of all compartments. ■

Because any arbitrary set of allowable input/output relations can be described by a subset lattice, we lose no generality or flexibility by restricting attention to lattice-structured flow policies.

It is also possible to take an arbitrary flow policy $P = (SC, \leq)$ and transform it into a lattice $P' = (SC', \leq')$; classes A and B in SC have corresponding classes A' and B' in SC' such that $A \leq B$ in P if and only if $A' \leq B'$ in P' [Denn76b]. This means that a flow is authorized under P if and only if it is authorized under P', where objects bound to class A in P are bound to class A' in P'. The transformation requires only that the relation $\leq$ be reflexive and transitive. To derive a relation $\leq'$ that is also antisymmetric, classes forming cycles (e.g., $A \leq B \leq C \leq A$) are compressed into single classes. To provide least upper and greatest lower bound operators, new classes are added. Figure 5.3 illustrates. The class AB is added to give A and B a least upper bound; the classes *Low* and *High* are added to give bounds on the complete structure. The classes D and E forming a cycle in the original structure are compressed into the single class DE in the lattice.

5.1.5 Flow Properties of Lattices

The lattice properties can be exploited in the construction of enforcement mechanisms. Transitivity of the relation $\leq$ implies any indirect flow $x \rightarrow y$ resulting from a sequence of flows

$$x = z_0 \rightarrow z_1 \rightarrow \ldots \rightarrow z_{n-1} \rightarrow z_n = y$$

is permitted if each flow $z_{i-1} \rightarrow z_i$ $(1 \leq i \leq n)$ is permitted, because

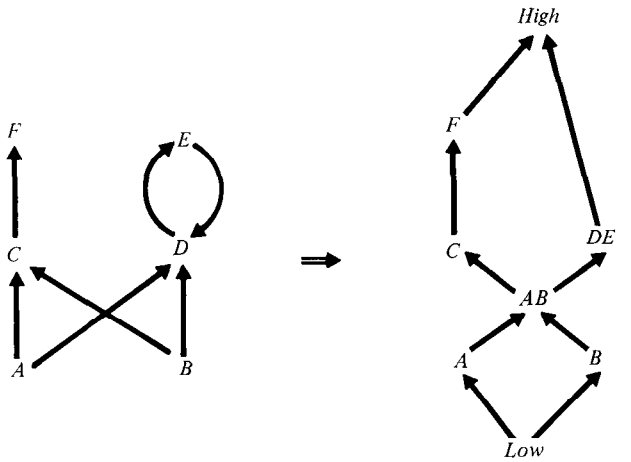
$$\underline{x} = \underline{z}_0 \leq \underline{z}_1 \leq \ldots \leq \underline{z}_{n-1} \leq \underline{z}_n = \underline{y}$$

implies $\underline{x} \leq \underline{y}$. Therefore, an enforcement mechanism need only verify direct flows.

Example:
The security of the indirect flow $x \rightarrow y$ caused by executing the sequence of statements

$z := x;$
$y := z$

FIGURE 5.3 Transformation of nonlattice policy into a lattice.



automatically follows from the security of the individual statements; that is, $\underline{x} \leq \underline{z}$ and $\underline{z} \leq \underline{y}$ implies $\underline{x} \leq \underline{y}$. ■

In general, transitivity of the relation $\leq$ implies that if executing each of the statements $S_1, \ldots, S_n$ is authorized, then executing the statements in sequence is authorized.

Transitivity greatly simplifies verifying implicit flows. To see why, suppose the value of a variable x is tested, and the program follows one of two execution paths depending on the outcome of the test. At some point, the execution paths join. Before the paths join, implicit flows from x (encoded in the program counter) must be verified. But after the paths join, information can only flow indirectly from x, and transitivity automatically ensures the security of these flows.

Example:
Consider the sequence

$z := 0;$
if $x = 1$ **then** $z := 1;$
$y := z,$

where x is initially 0 or 1. Execution of this sequence implicitly transfers information from x to z, and then explicitly from z to y. Because of transitivity, the security of the indirect flow $x \rightarrow y$ automatically follows from the security of the flows $x \rightarrow z$ and $z \rightarrow y$. ■

The existence of a least upper bound operator $\oplus$ implies that if $\underline{x}_1 \leq \underline{y}, \ldots, \underline{x}_n \leq \underline{y}$ for objects $x_1, \ldots, x_n$ and y, then there is a unique class $\underline{x} = \underline{x}_1 \oplus \ldots \oplus$

$\underline{x}_n$ such that $\underline{x} \preceq \underline{y}$. This means that a set of flows $x_1 \rightarrow y, \ldots, x_n \rightarrow y$ is authorized if the single relation $\underline{x} \preceq \underline{y}$ holds. This simplifies the design of verification mechanisms.

Example:
To verify the security of an assignment statement:

$$y := x_1 + x_2 * x_3 ,$$

a compiler can form the class $\underline{x} = \underline{x}_1 \oplus \underline{x}_2 \oplus \underline{x}_3$ as the expression on the right is parsed, and then verify the relation $\underline{x} \preceq \underline{y}$ when the complete statement is recognized. Similarly, a run-time enforcement mechanism can form the class $\underline{x}$ as the expression is evaluated, and verify the relation $\underline{x} \preceq \underline{y}$ when the assignment to y is performed. ■

The least upper bound operator $\oplus$ can be interpreted as a "class combining operator" specifying the class of a result $y = f(x_1, \ldots, x_n)$. With variable security classes, $\underline{x} = \underline{x}_1 \oplus \ldots \oplus \underline{x}_n$ is the minimal class that can be assigned to y such that the flows $x_i \rightarrow y$ are secure.

The existence of a greatest lower bound operator $\otimes$ implies that if $\underline{x} \preceq \underline{y}_1, \ldots, \underline{x} \preceq \underline{y}_n$ for objects x and $y_1, \ldots, y_n$, then there is a unique class $\underline{y} = \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$ such that $\underline{x} \preceq \underline{y}$. This means a set of flows $x \rightarrow y_1, \ldots, x \rightarrow y_n$ is authorized if the single relation $\underline{x} \preceq \underline{y}$ holds. This also simplifies the design of verification mechanisms.

Example:
To verify the security of an **if** statement

```
if x then
  begin
    y1 := 0;
    y2 := 0;
    y3 := 0
  end ,
```

a compiler can form the class $\underline{y} = \underline{y}_1 \otimes \underline{y}_2 \otimes \underline{y}_3$ as the statements are parsed, and then verify the implicit flows $x \rightarrow y_i$ $(i = 1, 2, 3)$ by checking that $\underline{x} \preceq \underline{y}$. ■

The least security class *Low* consists of all information that is unrestricted. This includes all data values that can be expressed in the language (e.g., integers and characters), and implies that execution of statements such as

```
x := 1
x := x + 1
x := 'On a clear disk you can seek forever'
```

is always authorized. Because *Low* is an identity on $\oplus$, the class of the expression "$x + 1$" is simply $\underline{x} \oplus Low = \underline{x}$.

5.2 FLOW CONTROL MECHANISMS

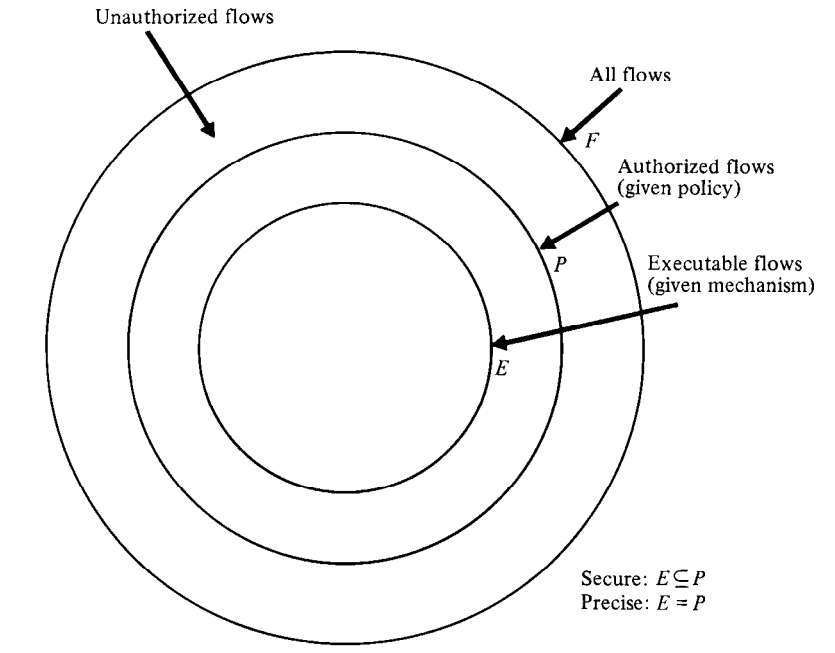
5.2.1 Security and Precision

Let F be the set of all possible flows in an information flow system, let P be the subset of F authorized by a given flow policy, and let E be the subset of F "executable" given the flow control mechanisms in operation. The system is **secure** if $E \subseteq P$; that is, all executable flows are authorized. A secure system is **precise** if $E = P$; that is, all authorized flows are executable. Figure 5.4 illustrates.

Note the similarity of Figure 5.4 with Figure 4.4, where a secure system is defined to be one that never enters an unauthorized state. We can also define a secure information flow system in terms of authorized states. Let s_0 be an initial state; s_0 is authorized, because flows are associated with state transitions. Let s be a state derived from s_0 by executing a command sequence α; then s is authorized if and only if all flows $x_{s_0} \rightarrow_{\alpha} y_s$ are authorized (i.e., $\underline{x}_{s_0} \leq \underline{y}_s$). By transitivity of the relation $\leq$, the state of the system remains authorized if the flows caused by each state transition are authorized. Defining secure information flow in terms of authorized states allows us to use a single definition of security for a system that supports both an access control policy and an information flow policy, as do most systems.

Although it is simple to construct a mechanism that provides security (by

FIGURE 5.4 Security and precision.



inhibiting all operations) it is considerably more difficult to construct a mechanism that provides precision as well.

Example:
Consider the assignment

$$y := k * x \ ,$$

and a policy in which $\underline{k} \preceq \underline{y}$ but $\underline{x} \not\preceq \underline{y}$; that is, information may flow from k to y but not from x to y. A mechanism that always prohibits execution of this statement will provide security. It may not be precise, however, because execution of the statement does not cause a flow $x \rightarrow y$ if either $k = 0$ or $H(x) = 0$ (i.e., there is no uncertainty about x). To design a mechanism that verifies the relation $\underline{x} \preceq \underline{y}$ only for actual flows $x \rightarrow y$ is considerably more difficult than designing one that verifies the relation $\underline{x} \preceq \underline{y}$ for any operation that can potentially cause a flow $x \rightarrow y$. ■

To further complicate the problem, it is generally undecidable whether a given system is secure or precise.

Example:
Consider the statement

if $f(n)$ *halts* **then** $y := x$ **else** $y := 0$,

where f is an arbitrary function and $\underline{x} \not\preceq \underline{y}$. Consider two systems: one that always allows execution of this statement, and another that always prohibits its execution. Clearly, it is undecidable whether the first system is secure or the second precise without solving the halting problem. To make matters worse, it is theoretically impossible to construct a mechanism that is both secure and precise [Jone75]. ■

In a secure system, the security class $\underline{y}$ of any object y will be at least as high as the class of the information stored in y. This does not imply, however, that a variable class $\underline{y}$ must monotonically increase over time. If information is removed from y, then $\underline{y}$ may decrease.

Example:
Consider the following sequence of statements:

$y := x;$
$z := y;$
$y := 0$.

After the first statement is executed, $\underline{y}$ must satisfy $\underline{x} \preceq \underline{y}$ to reflect the flow $x \rightarrow y$. After the last statement is executed, however, $\underline{y}$ can be lower than $\underline{x}$, because y no longer contains information about x. Thus, the security class of an object can be increased or decreased at any time as long it does not violate security. ■

In some systems, trusted processes are permitted to violate the flow requirements—for example, to lower the security class of an object. These processes, however, must meet the security policies of the system as a whole (see Section 5.6.3 for an example).

5.2.2 Channels of Flow

Lampson [Lamp73] observed that information flows along three types of channels:

- **Legitimate Channels,** which are intended for information transfer between processes or programs—e.g., the parameters of a procedure.
- **Storage Channels,** which are objects shared by more than one process or program—e.g., a shared file or global variable.
- **Covert Channels,** which are not intended for information transfer at all—e.g., a process's effect on the system load.

Legitimate channels are the simplest to secure. Securing storage channels is considerably more difficult, because every object—file, variable, and status bit—must be protected.

> ***Example:***
> To illustrate the subtlety of this point, consider the following scheme by which a process p can transfer a value x to a process q through the lock bit of a shared file f: p arranges regular intervals of use and nonuse of the file according to the binary representation of x; q requests use of the file each interval, and determines the corresponding bit of x according to whether the request is granted. ■

Although it is possible in principle to enforce security for all flows along legitimate and storage channels, covert channels are another matter (see also [Lipn75]). The problem is that information can be encoded in some physical phenomenon detectable by an external observer. For example, a process may cause its running time to be proportional to the value of some confidential value x which it reads. By measuring the running time on a clock that operates independently of the system, a user can determine the value of x. This type of covert channel is called a "timing channel"; other resource usage patterns may be exploited, such as the electric power consumed while running a program, or system throughput.

The only known technical solution to the problem of covert channels requires that jobs specify in advance their resource requirements. Requested resources are dedicated to a job, and the results, even if incomplete, are returned at precisely the time specified. With this strategy, nothing can be deduced from running time or resource usage that was not known beforehand; but even then, users can deduce something from whether their programs successfully complete. This scheme can be prohibitively expensive. Cost effective methods of closing all covert channels probably do not exist.

5.3 EXECUTION-BASED MECHANISMS

Security can be enforced either at execution time by validating flows as they are about to occur (prohibiting unauthorized ones), or at compile time by verifying the flows caused by a program before the program executes. This section studies the first approach; Sections 5.4–5.6 study the second approach. Before describing specific execution-based mechanisms, we discuss the general problem of dynamically enforcing security for implicit flows.

5.3.1 Dynamically Enforcing Security for Implicit Flow

Initially we assume objects have fixed security classes. We then consider the problems introduced by variable classes.

Dynamically enforcing security for explicit flows is straightforward, because an explicit flow always occurs as the result of executing an assignment of the form

$$y := f(x_1, \ldots, x_n) \ .$$

A mechanism can enforce the security of the explicit flows $x_i \rightarrow y$ $(1 \leq i \leq n)$ by verifying the relation $\underline{x}_1 \oplus \ldots \oplus \underline{x}_n \leq \underline{y}$ at the time of the assignment to y. If the relation is not true, it can generate an error message, and the assignment can be skipped or the program aborted.

Dynamically enforcing security for implicit flows would appear to be more difficult, because an implicit flow can occur in the absence of any explicit assignment. This was illustrated earlier by the statement

if $x = 1$ **then** $y := 1$.

This seems to suggest that verifying implicit flows to an object only at the time of an explicit assignment to the object is insecure. For example, verifying the relation $\underline{x} \leq \underline{y}$ only when the assignment "$y := 1$" is performed in the preceding statement would be insecure.

The interesting result, proved by Fenton [Fent74], is that security can be enforced by verifying flows to an object only at the time of explicit assignments to the object. But there is one catch: attempted security violations cannot generally be reported. This means if an unauthorized implicit flow is detected at the time of an assignment, not only must that assignment be skipped, but the error must not be reported to the user, and the program must keep running as though nothing has happened. The program cannot abort or even generate an error message to the user unless the user's clearance is at least that of the information causing the flow violation. It would otherwise be possible to use the error message or abnormal termination to leak high security data to a user with a low security clearance. Moreover, the program must terminate in a low security state; that is, any information encoded in the program counter from tests must belong to the class *Low*. Details are given in Sections 5.3.3 and 5.3.4.

Example:
Secure execution of the **if** statement

if $x = 1$ **then** $y := 1$

is described by

if $x = 1$
  then if $\underline{x} \leq \underline{y}$ **then** $y := 1$ **else skip**
  else skip .

Suppose x is 0 or 1, y is initially 0, $\underline{x} = High$, and $\underline{y} = Low$; thus, the flow $x \rightarrow y$ is not secure. Because the assignment to y is skipped both when $x = 1$ (because the security check fails) and when $x = 0$ (because the test "$x = 1$" fails), y is always 0 when the statement terminates, thereby giving no information about x. Note that if an error flag E is set to 1 when the security check fails, then the value of x is encoded in the flag ($E = 1$ implies $x = 1$, $E = 0$ implies $x = 0$). ■

In general, suppose an assignment

$y := f(x_1, \ldots, x_m)$

is directly conditioned on variables $x_{m+1}, \ldots, x_n$. Then the explicit and implicit flow to y can be validated by checking that the relation

$$\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \oplus \underline{x}_{m+1} \oplus \ldots \oplus \underline{x}_n \leq \underline{y}$$

holds, skipping the assignment if it does not. This is secure, because the value of y is not changed when the security check fails; it is as though the program never even made the test that would have led to the assignment and, therefore, the implicit flow.

Fenton's result is significant for two reasons. First, it is much simpler to construct a run-time enforcement mechanism if all implicit and explicit flows can be validated only at the time of actual assignments. Second, such a mechanism is likely to be more precise than one that checks implicit flows that occur in the absence of explicit assignments.

Example:
Consider the statement

if $x = 1$ **then** $y := 1$ **else** $z := 1$

where $\underline{x} = High$. Suppose that when $x = 1$, $\underline{y} = High$ and $\underline{z} = Low$, but when $x \neq 1$, $\underline{y} = Low$ and $\underline{z} = High$. If both relations $\underline{x} \leq \underline{y}$ and $\underline{x} \leq \underline{z}$ are tested on both branches, the program will be rejected, even though it can be securely executed using Fenton's approach. (Verification of this is left to the reader.) ■

Now, an error flag can be securely logged in a record having a security class

at least that of the information. Although it is insecure to report it to a user in a lower class, the capacity of the leakage channel is at most 1 bit (because at most 1 bit can be encoded in a 1-bit flag). The error message can, however, potentially disclose the exact values of all variables in the system. To see why, suppose all information in the system is encoded in variables $x_1, \ldots, x_n$, and there exists a variable y known to be 0 such that $\underline{x_i} \preceq \underline{y}$ $(i = 1, \ldots, n)$. Then execution of the statement

if $(x_1 = val_1)$ and $(x_2 = val_2)$ and . . . and $(x_n = val_n)$
then $y := 1$

generates an error message when $x_1 = val_1, \ldots, x_n = val_n$, disclosing the exact values of $x_1, \ldots, x_n$. But if a single $x_i \neq val_i$, an error message is not generated, and little can be deduced from the value of y. Thus, an intruder could not expect to learn much from the statement. Similarly, execution of the statement

if $\sim ((x_1 = val_1)$ and $(x_2 = val_2)$ and . . . and $(x_n = val_n))$
then $y := 1$

terminates successfully (without causing a security error) when $x_1 = val_1, \ldots, x_n = val_n$, leaking the exact values of $x_1, \ldots, x_n$. Note, however, that the values of $x_1, \ldots, x_n$ are not encoded in y, because y will be 0 even when the test succeeds. The values are encoded in the termination status of the program, which is why only 1 bit of information can be leaked.

It is clearly unsatisfactory not to report errors, but errors can be logged, and offending programs removed from the system. This solution may be satisfactory in most cases. It is not satisfactory, however, if the 1 bit of information is sufficiently valuable (e.g., a signal to attack). There is a solution—we can verify the security of all flows caused by a program before the program executes. This approach is studied in Sections 5.4 and 5.5.

Suppose now that objects have variable security classes. If an object y is the target of an assignment "$y := f(x_1, \ldots, x_m)$" conditioned on objects $x_{m+1}, \ldots, x_n$, changing y's class to

$$\underline{y} := \underline{x_1} \oplus \ldots \oplus \underline{x_m} \oplus \underline{x_{m+1}} \oplus \ldots \oplus \underline{x_n}$$

at the time of the assignment might seem sufficient for security. Although it is secure for the explicit flows, it is not secure for the implicit flows, because the execution path of the program will be different for different values of $x_{m+1}, \ldots, x_n$. This is illustrated by the following example.

Example:
Consider the execution of the procedure *copy1* shown in Figure 5.5. Suppose the local variable z has a variable class (initially *Low*), $\underline{z}$ is changed whenever z is assigned a value, and flows into y are verified whenever y is assigned a value. Now, if the procedure is executed with $x = 0$, the test "$x = 0$" succeeds and $(z, \underline{z})$ becomes $(1, \underline{x})$; hence the test "$z = 0$" fails, and y remains 0. If it is executed with $x = 1$, the test "$x = 0$" fails, so $(z, \underline{z})$ remains $(0, Low)$; hence the test "$z = 0$" succeeds, y is assigned the value 1,

FIGURE 5.5 Procedure *copy1*.

```
procedure copy1(x: integer; var y: integer);
  "copy x to y"
  var z: integer;
  begin
    y := 0;
    z := 0;
    if x = 0 then z := 1;
    if z = 0 then y := 1
  end
end copy1
```

and the relation $Low \leq \underline{y}$ is verified. In either case, execution terminates with $y = x$, but without verifying the relation $\underline{x} \leq \underline{y}$. The system, therefore, is insecure when $\underline{x} \not\leq \underline{y}$. ■

To construct a secure mechanism for variable classes, several approaches are possible. Denning [Denn75] developed an approach that accounts for all implicit flows, including those occurring in the absence of explicit ones. In the *copy1* program, for example, $\underline{z}$ would be increased to $\underline{x}$ even when the test "$x = 0$" fails, and the relation $\underline{z} \leq \underline{y}$ would be verified regardless of whether the test "$z = 0$" succeeds. The disadvantage of this approach is that a compiler must analyze the flows of a program to determine what objects could receive an implicit flow, and insert additional instructions into the compiled code to increase a class if necessary.

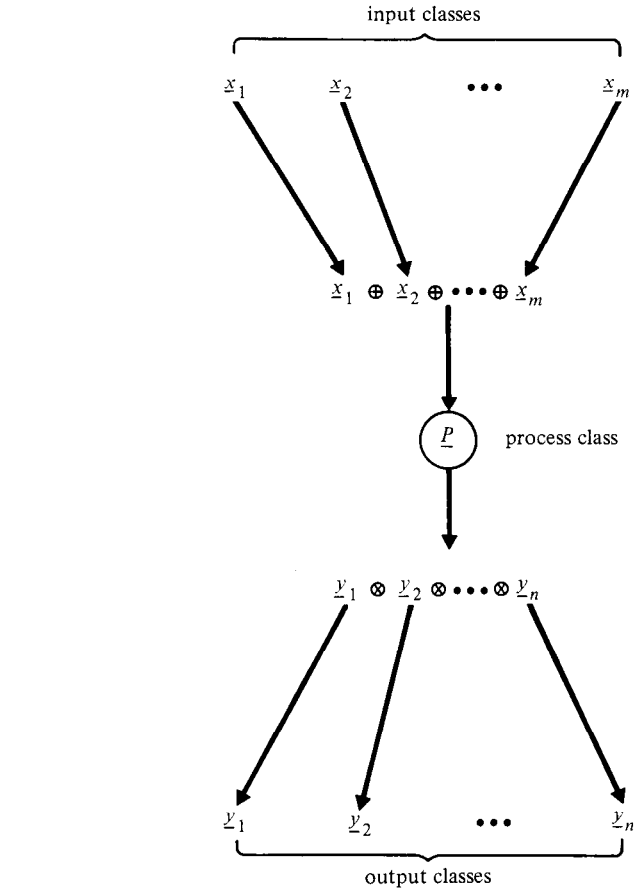
Fenton [Fent73] and Gat and Saal [Gat75] proposed a different solution. Their solution involves restoring the class and value of any object whose class was increased during execution of a conditional structure to the class and value it had just before entering the structure. Hence, the objects whose class and value are restored behave as "local objects" within the conditional structure. This ensures the security of all implicit flows by nullifying those that caused a class increase. The disadvantage of this approach is the added complexity to the run-time enforcement mechanism.

Lampson suggested another approach that is much simpler to implement than either of the preceding. The class of an object would be changed only to reflect explicit flows into the object; implicit flows would be verified at the time of explicit ones, as for fixed classes. For example, execution of the statement "**if** $x = 1$ **then** $y := z$" would set $\underline{y}$ to $\underline{z}$, and then verify the relation $\underline{x} \leq \underline{y}$.

5.3.2 Flow-Secure Access Controls

Simple flow controls can be integrated into the access control mechanisms of operating systems. Each process p is assigned a security clearance $\underline{p}$ specifying the highest class p may read from and the lowest class p may write into. Security is

FIGURE 5.6 Access control mechanism.



enforced by access controls that permit p to acquire read access to an object x only if $\underline{x} \preceq \underline{p}$, and write access to an object y only if $\underline{p} \preceq \underline{y}$. Hence, p can read from x_1, . . . , x_m and write into y_1, . . . , y_n only if

$$\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \preceq \underline{p} \preceq \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$$

(see Figure 5.6). This automatically guarantees the security of all flows, explicit or implicit, internal to the process.

In military systems, access controls enforce both a **nondiscretionary** policy of information flow based on the military classification scheme (the multilevel security policy), and a **discretionary** policy of access control based on "need-to-know" (that is, on the principle of least privilege). A process running with a *Secret* clearance, for example, is permitted to read only from *Unclassified, Confidential,* and *Secret* objects; and to write only to *Secret* and *Top Secret* objects (although

integrity constraints may prevent it from writing into *Top Secret* objects). All the kernel-based systems discussed in Section 4.6.1 use access controls to enforce multilevel security for user and untrusted system processes.

One of the first systems to use access controls to enforce multilevel security was the ADEPT-50 time-sharing system developed at SDC in the late 1960s [Weis69]. In ADEPT, the security clearance $\underline{p}$ of a process p, called its "high water mark", is dynamically determined by the least upper bound of the classes of all files opened for read or write operations; thus $\underline{p}$ is monotonically nondecreasing. When the process closes a newly created file f, the class $\underline{f}$ is set to $\underline{p}$. Rotenberg's [Rote74] Privacy Restriction Processor is similar, except that $\underline{p}$ is determined by the $\oplus$ of the classes opened for read, and whenever the process writes into a file f, the file's class is changed to $\underline{f} := \underline{f} \oplus \underline{p}$.

This approach of dynamically assigning processes and objects to variable security classes can lead to leaks, as illustrated by the following example.

Example:
Suppose the procedure *copy1* (see Figure 5.5) is split between processes *p1* and *p2*, where *p1* and *p2* communicate through a global variable z dynamically bound to its security class:

p1: **if** $x = 0$ **then** $z := 1$
p2: **if** $z = 0$ **then** $y := 1$.

Now suppose $\underline{p1}$ and $\underline{p2}$ are set to the $\oplus$ of the classes of all objects opened for read or write operations, y and z are initially 0 and in class *Low*, $\underline{z}$ is changed only when z is opened for writing, and flows to y are verified only when y is opened for writing. When $x = 0$, *p1* terminates with $z = 1$ and $\underline{z} = \underline{p1} = \underline{x}$; thus, $\underline{p2}$ is set to $\underline{x}$. But the test "$z = 0$" in *p2* fails, so y is never opened for writing, and the relation $\underline{p2} \leq \underline{y}$ is never verified. When $x = 1$, *p1* terminates with $\underline{p1} = \underline{x}$; however, because z is never opened for writing, $(z, \underline{z})$ remains (0, *Low*); thus, $\underline{p2} = Low$, y becomes 1, and the relation $Low \leq \underline{y}$ is verified. In both cases, *p2* terminates with $y = x$, even though the relation $\underline{x} \leq \underline{y}$ is never verified. Thus, a leak occurs if $\underline{x} \not\leq \underline{y}$.

This problem does not arise when objects and processes have fixed security classes. To see why, suppose *p1* runs in the minimal class needed to read x; i.e., $\underline{p1} = \underline{x}$. Then *p1* will never be allowed to write into z unless $\underline{x} \leq \underline{z}$. Similarly, *p2* will not be allowed to read z unless $\underline{z} \leq \underline{p2}$, and it will never be allowed to write into y unless $\underline{p2} \leq \underline{y}$. Hence, no information can flow from x to y unless $\underline{x} \leq \underline{z} \leq \underline{y}$. ■

Because of the problems caused by variable classes, most access-control-based mechanisms bind objects and processes to fixed security classes. The class of a process p is determined when p is initiated.

Flow-secure access controls provide a simple and efficient mechanism for enforcing information flow within user processes. But they are limited, because they do not distinguish different classes of information within a process. For example, if a process reads both confidential (*High*) and nonconfidential (*Low*) data,

then $\underline{p}$ must be *High*, and any objects written by p must be in class *High*. The process cannot be given write access to objects in class *Low*, because there would be no way of knowing whether the information transferred to these objects was confidential or nonconfidential. The process cannot, therefore, transfer information derived only from the nonconfidential inputs to objects in class *Low*.

In general, it is not possible with access controls alone to enforce security in processes that handle different classes of information simultaneously. This rules out using access controls to enforce the security of certain operating system processes that must access information in different classes and communicate with processes at different levels. Yet the flows within system processes must be secure, lest other processes exploit this to establish leakage channels through system state variables. For example, a Trojan Horse in a file editor could use such a channel to leak *Top Secret* information in a file being edited to a user with a lower clearance.

To enforce security within processes that handle different classes of information, the information flow internal to a process must be examined. The remainder of this chapter describes hardware and software mechanisms to do this.

5.3.3 Data Mark Machine

Fenton [Fent74,Fent73] studied a run-time validation mechanism in the context of an abstract machine called a **Data Mark Machine** (DMM). The Data Mark Machine is a Minsky machine [Mins67] extended to include tags (**data marks**) for marking the security class of each register (memory location).

A Minsky machine has three instructions:

$x := x + 1$ "increment"
if $x = 0$ **then goto** n **else** $x := x - 1$ "branch on zero or decrement"
halt,

where x is a register and n is a statement label. Despite its simplicity, the machine can compute all computable functions as long as there are at least two (infinite) registers and a register containing zero.

Fenton's important observation was that a **program counter class**, $\underline{pc}$, could be associated with a process to validate all implicit flows caused by the process. This class is determined as follows: whenever a process executes a conditional branch

if $x = 0$ **then goto** n,

the current value and class of pc is pushed onto a stack, and $(pc, \underline{pc})$ is replaced with $(n, \underline{pc} \oplus \underline{x})$. The class $\underline{pc}$ is increased by $\underline{x}$ because information about x is encoded in the execution path of the program. The only way $\underline{pc}$ can be decreased is by executing a **return** instruction, which restores $(pc, \underline{pc})$ to its earlier value. This forces the program to return to the instruction following the conditional branch, whence the execution path is no longer directly conditioned on the value of x.

Initially $\underline{pc} = \mathit{Low}$, so that immediately before executing an instruction on a path directly conditioned on the values of $x_1, \ldots, x_m$, $\underline{pc} = \underline{x}_1 \oplus \ldots \oplus \underline{x}_m$. If the

TABLE 5.1 Data Mark Machine (DMM).

Instruction	Execution
$x := x + 1$	if $\underline{pc} \leq \underline{x}$ then $\underline{x} := x + 1$ else skip
if $x = 0$	if $x = 0$
then goto n	then $\{push(pc, \underline{pc}); \underline{pc} := \underline{pc} \oplus \underline{x}; pc := n\}$
else $x := x - 1$	else {if $\underline{pc} \leq \underline{x}$ then $x := x - 1$ else skip}
if' $x = 0$	if $x = 0$
then goto n	then {if $\underline{x} \leq \underline{pc}$ then $pc := n$ else skip}
else $x := x - 1$	else {if $\underline{pc} \leq \underline{x}$ then $x := x - 1$ else skip}
return	$pop(pc, \underline{pc})$
halt	if empty stack then Halt

instruction is an assignment "$y := y + 1$" or "$y := y - 1$" then the hardware validates the relation $\underline{pc} \leq \underline{y}$, inhibiting the assignment if the condition is not satisfied; this ensures the security of the implicit flows $x_i \rightarrow y$ ($i = 1, \ldots, m$). The **halt** instruction requires the stack be empty, so the program cannot terminate without returning from each branch. This ensures the final state of the program contains only *Low* security information. If security violations are not reported, the mechanism is completely secure as discussed earlier. If violations are reported and insecure programs aborted, an insecure program can leak at most 1 bit.

The complete semantics of the Data Mark Machine are summarized in Table 5.1. The second **if** statement, denoted **if'**, allows a program to branch without stacking the program counter (stacking is unnecessary for security because $\underline{x} \leq \underline{pc}$ implies $\underline{pc} = \underline{pc} \oplus \underline{x}$) .

Example:
Figure 5.7 shows how the *copy1* program of Figure 5.5 can be translated into instructions for the DMM. For simplicity, we assume y and z are initially 0. Note that the translated program modifies the value of x; this does not affect its flow from x to y.

The following execution trace shows the effect of executing each instruction when $x = 0$ and $\underline{x} \leq \underline{z}$:

Instruction	x	y	z	$\underline{pc}$	Security Check
initial	0	0	0	*Low*	
1				$\underline{x}$	
4			1		$\underline{x} \leq \underline{z}$
5				*Low*	
2			0		$Low \leq \underline{z}$
3					

The reader should verify that the program causes information to flow from x to y only when $\underline{x} \leq \underline{z} \leq \underline{y}$. ■

FIGURE 5.7 Translation of *copy1* for DMM.

```
1  if x = 0 then goto 4 else x := x − 1
2  if z = 0 then goto 6 else z := z − 1
3  halt
4  z := z + 1
5  return
6  y := y + 1
7  return
```

5.3.4 Single Accumulator Machine

The protection features of the Data Mark Machine can be implemented in actual systems. We shall outline how this could be done on a single accumulator machine (SAM) with a tagged memory. Our approach is similar to the one proposed in [Denn75].

The security class of each data object is stored in the tag field of the corresponding memory location. A variable tag $\underline{acc}$ represents the class of the information in the accumulator. As in the Data Mark Machine, there is a program counter stack, and a class $\underline{pc}$ associated with the current program counter. The semantics of typical instructions are shown in Table 5.2. (The semantics for operations to subtract, multiply, etc. would be similar to those for ADD.)

Example:
Execution of the statement "$y := x1 * x2 + x3$" is shown next:

Operation	Execution Trace
LOAD $x1$	$acc := x1$; $\underline{acc} := \underline{x1} \oplus \underline{pc}$
MULT $x2$	$acc := acc * x2$; $\underline{acc} := \underline{acc} \oplus \underline{x2} \oplus \underline{pc}$
ADD $x3$	$acc := acc + x3$; $\underline{acc} := \underline{acc} \oplus \underline{x3} \oplus \underline{pc}$
STORE y	if $\underline{acc} \oplus \underline{pc} \leq \underline{y}$ then $y := acc$

TABLE 5.2 Single accumulator machine (SAM).

Operation	Execution
LOAD x	$acc := x$; $\underline{acc} := \underline{x} \oplus \underline{pc}$
STORE y	if $\underline{acc} \oplus \underline{pc} \leq \underline{y}$ then $y := acc$ else skip
ADD x	$acc := acc + x$; $\underline{acc} := \underline{acc} \oplus \underline{x} \oplus \underline{pc}$
B n	$pc := n$
BZ n	if $(acc = 0)$ then $\{push(pc, \underline{pc})$; $\underline{pc} := \underline{pc} \oplus \underline{acc}$; $pc := n\}$
BZ′ n	if $(acc = 0)$ and $(\underline{acc} \leq \underline{pc})$ then $pc := n$ else skip
RETURN	$pop(pc, \underline{pc})$
STOP	if empty stack then stop

FIGURE 5.8 Translation of procedure *copy1* on SAM.

$y := 0;$	1	LOAD 0	
	2	STORE y	
$z := 0;$	3	LOAD 0	
	4	STORE z	
if $x = 0$	5	LOAD x	
	6	BZ 8	"*push* (7, $\underline{pc}$)"
	7	B 11	
then $z := 1;$	8	LOAD 1	
	9	STORE z	
	10	RETURN	"*pop* − goto 7"
if $z = 0$	11	LOAD z	
	12	BZ 14	"*push* (13, $\underline{pc}$)"
	13	B 17	
then $y := 1$	14	LOAD 1	
	15	STORE y	
	16	RETURN	"*pop* − goto 13"
	17	STOP	

TABLE 5.3 Semantics for STORE on variable class machine.

Operation	Execution
STORE y	if $\underline{pc} \leq \underline{acc}$ then $\{y := acc; \underline{y} := \underline{acc}\}$

Note the check associated with the STORE verifies $\underline{x1} \oplus \underline{x2} \oplus \underline{x3} \oplus \underline{pc} \leq \underline{y}$. ■

Figure 5.8 shows how the procedure *copy1* in Figure 5.5 can be translated into instructions on the single accumulator machine. Execution of the program is left as an exercise.

If the memory tags are variable, security cannot be guaranteed by changing the semantics of the STORE operation to:

$$y := acc; \underline{y} := \underline{acc} \oplus \underline{pc} \ .$$

(See the discussion of the procedure *copy1*.) Security can be guaranteed, however, by changing an object's class to reflect explicit flows to the object, as long as implicit flows are verified as for fixed classes. This approach is taken in Table 5.3.

5.4 COMPILER-BASED MECHANISM

We first consider a simple program certification mechanism that is easily integrated into any compiler. The mechanism guarantees the secure execution of each statement in a program—even those that are not executed, or, if executed, do not cause an information flow violation. The mechanism is not precise, however, as it

will reject secure programs. For example, consider the following program segment

if $x = 1$ **then** $y := a$ **else** $y := b$.

Execution of this segment causes either a flow $a \rightarrow y$ or a flow $b \rightarrow y$, but not both. The simple certification mechanism, however, requires that both $\underline{a} \leq \underline{y}$ and $\underline{b} \leq \underline{y}$. This is not a problem if both relations hold; but if $\underline{a} \leq \underline{y}$ holds only when $x = 1$, and $\underline{b} \leq \underline{y}$ holds only when $x \neq 1$, we would like a more precise mechanism. To achieve this, in Section 5.5 we combine flow proofs with correctness proofs, so that the actual execution path of a program can be taken into account.

The mechanism described here is based on [Denn75,Denn77]. It was developed for verifying the internal flows of applications programs running in an otherwise secure system, and not for security kernel verification.

5.4.1 Flow Specifications

We first consider the certification of procedures having the following structure:

procedure *pname*($x_1, \ldots, x_m$; **var** $y_1, \ldots, y_n$);
  var $z_1, \ldots, z_p$; "local variables"
  S "statement body"
end *pname* ,

where $x_1, \ldots, x_m$ are input parameters, and $y_1, \ldots, y_n$ are output parameters or input/output parameters. Let u denote an input parameter x or input/output parameter y, and let v denote either a parameter or local variable. The declaration of v has the form

v: type **class** $\{u \mid u \rightarrow v$ is allowed$\}$,

where the **class** declaration specifies the set of all parameters permitted to flow into v. Note that the class of an input parameter x will be specified as the singleton set $\underline{x} = \{x\}$. The class of an input/output parameter y will be of the form $\{y, u_1, \ldots, u_k\}$, where $u_1, \ldots, u_k$ are other inputs to y. If y is an output only, the class of y will be of the form $\{u_1, \ldots, u_k\}$ (i.e., $y \notin \underline{y}$); hence, its value must be cleared on entry to the procedure to ensure its old value cannot flow into the procedure. References to global variables are not permitted; thus, each nonlocal object referenced by a procedure must be explicitly passed as a parameter.

The class declarations are used to form a subset lattice of allowable input-output relations as described in Section 5.1.4. Specifying the security classes of the parameters and local variables as a subset lattice simplifies verification. Because each object has a fixed security class during program verification, the problems caused by variable classes are avoided. At the same time, the procedure is not restricted to parameters having specific security classes; the classes of the actual parameters need only satisfy the relations defined for the formal parameters. We could instead declare specific security classes for the objects of a program; the mechanism described here works for both cases.

Example:
The following gives the flow specifications of a procedure that computes the maximum of its inputs:

```
procedure max(x: integer class {x};
              y: integer class {y};
              var m: integer class {x, y});
  begin
    if x > y then m := x else m := y
  end
end max .
```

The security class specified for the output m implies that $\underline{x} \leq \underline{m}$ and $\underline{y} \leq \underline{m}$. ■

Example:
The following gives the input-output specifications of a procedure that swaps two variables x and y, and increments a counter i (recording the number of swaps):

```
procedure swap(var x, y: integer class {x, y};
               var i: integer class {i});
  var t: integer class {x, y};
  begin
    t := x;
    x := y;
    y := t;
    i := i + 1
  end
end swap .
```

Because both $\underline{x} \leq \underline{y}$ and $\underline{y} \leq \underline{x}$ are required for security, the specifications state that $\underline{x} = \underline{y}$; this class is also assigned to the local variable t. Note that i is in a class by itself because it does not receive information from either x or y. ■

5.4.2 Security Requirements

A procedure is **secure** if it satisfies its specifications; that is, for each input u and output y, execution of the procedure can cause a flow $u \rightarrow y$ only if the classes specified for u and y satisfy the relation $\underline{u} \leq \underline{y}$. The procedures in the preceding examples are secure.

We shall define the security requirements for the statement (body) S of a procedure recursively, giving sufficient conditions for security for each statement type. These conditions are expressed as constraints on the classes of the parameters and local variables. A procedure is then secure if its flow specifications (i.e.,

class declarations) imply these constraints are satisfied. The conditions are not necessary for security, however, and some secure programs will be rejected. Initially, we assume S is one of the following:

1. Assignment: $b := e$
2. Compound: **begin** $S_1; \ldots; S_n$ **end**
3. Alternation: **if** e **then** S_1 [**else** S_2]
4. Iteration: **while** e **do** S_1
5. Call: $q(a_1, \ldots, a_m, b_1, \ldots, b_n)$

where the S_i are statements, and e is an expression with operands $a_1, \ldots, a_n$, which we write as

$$e = f(a_1, \ldots, a_n) \ ,$$

where the function f has no side effects. The class of e is given by

$$\underline{e} = \underline{a}_1 \oplus \ldots \oplus \underline{a}_n \ .$$

We assume all data objects are simple scalar types or files, and all statements terminate and execute as specified; there are no abnormal terminations due to exceptional conditions or program traps. Structured types and abnormal program terminations are considered later.

The security conditions for the explicit flow caused by an assignment are as follows:

Security conditions for assignment:
Execution of an assignment

$$b := e$$

is secure if $\underline{e} \preceq \underline{b}$. ■

Because input and output operations can be modeled as assignments in which the source or target object is a file, they are not considered separately here.

Because the relation $\preceq$ is transitive, the security conditions for a compound statement are as follows:

Security conditions for compound:
Execution of the statement

begin $S_1; \ldots; S_n$ **end**

is secure if execution of each of $S_1, \ldots, S_n$ is secure. ■

Consider next the alternation statement

if e **then** S_1 [**else** S_2] .

If objects $b_1, \ldots, b_m$ are targets of assignments in S_1 [or S_2], then execution of the

if statement can cause implicit flows from e to each b_j. We therefore have the following:

Security conditions for alternation:
Execution of the statement

if e **then** S_1 [**else** S_2]

is secure if
(i) Execution of S_1 [and S_2] is secure, and
(ii) $\underline{e} \leq \underline{S}$, where $\underline{S} = \underline{S}_1$ [$\otimes \underline{S}_2$] and
$\underline{S}_1 = \otimes\{\underline{b} \mid b$ is a target of an assignment in $S_1\}$,
$\underline{S}_2 = \otimes\{\underline{b} \mid b$ is a target of an assignment in $S_2\}$ ■

Condition (ii) implies $\underline{e} \leq \underline{b}_1 \otimes \ldots \otimes \underline{b}_m$, and, therefore, $\underline{e} \leq \underline{b}_j$ $(1 \leq j \leq m)$.

Example:
For the following statement

if $x > y$ **then**
begin
$z := w$;
$i := k + 1$
end ,

condition (ii) is given by $\underline{x} \oplus \underline{y} \leq \underline{z} \otimes \underline{i}$. ■

Consider an iteration statement

while e **do** S_1 .

If $b_1, \ldots, b_m$ are the targets of flows in S_1, then execution of the statement can cause implicit flows from e to each b_j. Security is guaranteed by the same condition as for the **if** statement:

Security conditions for iteration:
Execution of the statement

while e **do** S_1

is secure if
(i) S terminates,
(ii) Execution of S_1 is secure, and
(iii) $\underline{e} \leq \underline{S}$, where $\underline{S} = \underline{S}_1$ and
$\underline{S}_1 = \otimes \{\underline{b} \mid b$ is a target of a possible flow in $S_1\}$. ■

Because other iterative structures (e.g., **for** and **repeat-until**) can be described in terms of the **while**, we shall not consider them separately here.

Nonterminating loops can cause additional implicit flows, because execution

of the remaining statements is conditioned on the loop terminating—this is discussed later. Even terminating loops can cause covert flows, because the execution time of a procedure depends on the number of iterations performed. There appears to be no good solution to this problem.

Finally, consider a call

$$q(a_1, \ldots, a_m, b_1, \ldots, b_n) \ ,$$

where $a_1, \ldots, a_m$ are the actual input arguments and $b_1, \ldots, b_n$ the actual input/output arguments corresponding to formal parameters $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$, respectively. Assuming q is secure, execution of q can cause a flow $a_i \rightarrow b_j$ only if $\underline{x}_i \leq \underline{y}_j$; similarly, it can cause a flow $b_i \rightarrow b_j$ only if $\underline{y}_i \leq \underline{y}_j$. We therefore have the following:

Security conditions for procedure call:
Execution of the call

$$q(a_1, \ldots, a_m, b_1, \ldots, b_n)$$

is secure if

(i) q is secure, and
(ii) $\underline{a}_i \leq \underline{b}_j$ if $\underline{x}_i \leq \underline{y}_j$ $(1 \leq i \leq m, 1 \leq j \leq n)$ and
$\underline{b}_i \leq \underline{b}_j$ if $\underline{y}_i \leq \underline{y}_j$ $(1 \leq i \leq n, 1 \leq j \leq n)$ ■

If q is a main program, the arguments correspond to actual system objects. The system must ensure the classes of these objects satisfy the flow requirements before executing the program. This is easily done if the certification mechanism stores the flow requirements of the parameters with the object code of the program.

Example:
Consider the procedure $max(x, y, m)$ of the preceding section, which assigns the maximum of x and y to m. Because the procedure specifies that $\underline{x} \leq \underline{m}$ and $\underline{y} \leq \underline{m}$ for output m, execution of a call "$max(a, b, c)$" is secure if $\underline{a} \leq \underline{c}$ and $\underline{b} \leq \underline{c}$. ■

Example:
The security requirements of different statements are illustrated by the procedure shown in Figure 5.9, which copies x into y using a subtle combination of implicit and explicit flows (the security conditions are shown to the right of each statement). Initially x is 0 or 1. When $x = 0$, the first test "$z = 1$" succeeds, and the first iteration sets $y = 0$ and $z = x = 0$; the second test "$z = 1$" then fails, and the program terminates. When $x = 1$, the first test "$z = 1$" succeeds, and the first iteration sets $y = 0$ and $z = x = 1$; the second test "$z = 1$" also succeeds, and the second iteration sets $y = 1$ and $z = 0$; the third test "$z = 1$" then fails, and the program terminates. In both cases, the program terminates with $y = x$. The flow $x \rightarrow y$ is indirect: an explicit flow x

FIGURE 5.9 Procedure *copy2*.

```
procedure copy2(x: integer class {x};
                var y: integer class {x});
  "copy x to y"
  var z: integer class {x};
  begin
    z := 1;                 Low ≤ z
    y := −1;                Low ≤ y
    while z = 1 do          z ≤ y ⊗ z
      begin
        y := y + 1;         y ≤ y
        if y = 0            y ≤ z
          then z := x       x ≤ z
          else z := 0       Low ≤ z
      end
  end
end copy2
```

$\rightarrow z$ occurs during the first iteration; this is followed by an implicit flow $z \rightarrow y$ during the second iteration due to the iteration being conditioned on z.

The security requirements for the body of the procedure imply the relations $\underline{x} \leq \underline{z} \leq \underline{y} \leq \underline{z}$ must hold. Because the flow specifications state $\underline{y} = \underline{z} = \underline{x} = \{x\}$, the security requirements are met, and the procedure is secure. If the specifications did not state the dependency of either y or z on x, the security requirements would not be satisfied.

Now, consider a call "*copy2*(a, b)", for actual arguments a and b. Because y is the only formal output parameter in *copy2* and x satisfies the relation $\underline{x} \leq \underline{y}$, the call is secure provided $\underline{a} \leq \underline{b}$. Although the stronger relation $\underline{x} = \underline{y}$ holds in *copy2*, it is unnecessary that $\underline{a} = \underline{b}$ because the argument a is not an output of *copy2* (hence the relation $\underline{b} \leq \underline{a}$ need not hold). Thus, a call "*copy2*(a, b)" is secure if b is in a higher class than a, but not vice versa. ■

5.4.3 Certification Semantics

The certification mechanism is sufficiently simple that it can be easily integrated into the analysis phase of a compiler. Semantic actions are associated with the syntactic types of the language as shown in Table 5.4 (see [Denn75,Denn77] for details). As an expression $e = f(a_1, \ldots, a_n)$ is parsed, the class $\underline{e} = \underline{a}_1 \oplus \ldots \oplus \underline{a}_n$ is computed and associated with the expression. This facilitates verification of explicit and implicit flows from $a_1, \ldots, a_n$.

Example:
As the expression e = "$a + b*c$" is parsed, the classes of the variables are associated with the nodes of the syntax tree and propagated up the tree, giving $\underline{e} = \underline{a} \oplus \underline{b} \oplus \underline{c}$. ■

TABLE 5.4 Certification semantics.

Expression e	Semantic Actions
$f(a_1, \ldots, a_n)$ Statement S	$\underline{e} := \underline{a}_1 \oplus \ldots \oplus \underline{a}_n$
$b := e$	$\underline{S} := \underline{b}$; verify $\underline{e} \leq \underline{S}$
begin $S_1; \ldots; S_n$ **end**	$\underline{S} := \underline{S}_1 \otimes \ldots \otimes \underline{S}_n$
if e **then** S_1 [**else** S_2]	$\underline{S} := \underline{S}_1$ [$\otimes \underline{S}_2$] ; verify $\underline{e} \leq \underline{S}$
while e **do** S_1	$\underline{S} := \underline{S}_1$; verify $\underline{e} \leq \underline{S}$
$q(a_1, \ldots, a_m; b_1, \ldots, b_n)$	verify $\underline{a}_i \leq \underline{b}_j$ if $\underline{x}_i \leq \underline{y}_j$ verify $\underline{b}_i \leq \underline{b}_j$ if $\underline{y}_i \leq \underline{y}_j$ $\underline{S} := \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$

Similarly, as each statement S is recognized, a class $\underline{S} = \otimes\{\underline{b} \mid b$ is the target of an assignment in $S\}$ is computed and associated with the statement. This facilitates verification of implicit flows into S.

Example:
Figure 5.10 illustrates the certification of the statement

```
if a = b then
  begin
    c := 0;
    d := d + 1
  end
else d := c * e .
```

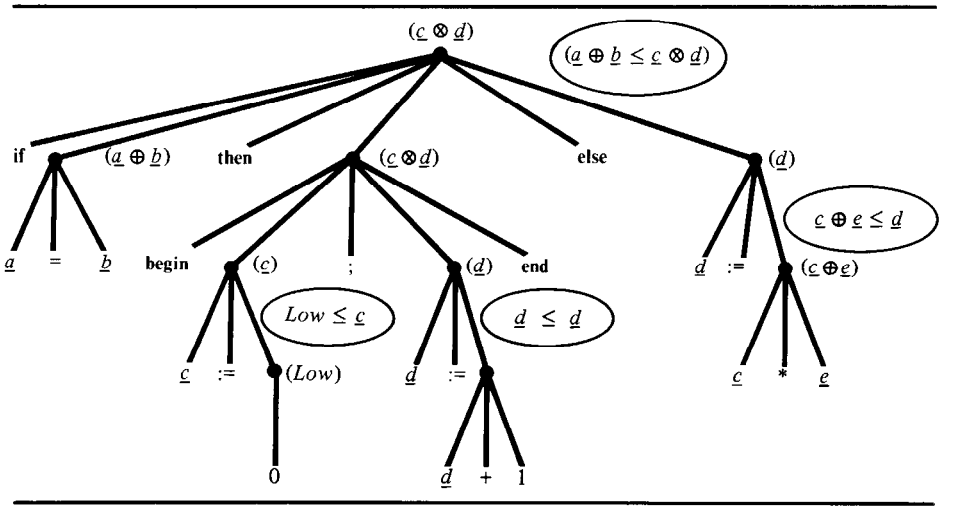
The overall parse is represented as a syntax tree for the statement. The security classes (in parentheses) and verification checks (circled) are shown opposite each subtree. The semantic actions propagate the classes of expressions and statements up the tree. ■

We shall now consider extensions to the flow semantics and certification mechanisms to handle structured data types, arbitrary control structures for sequential programs, concurrent programs, and abnormal terminations.

5.4.4 General Data and Control Structures

The mechanism can be extended to handle complex data structures. We shall outline the security requirements for 1-dimensional arrays here. Consider a vector

FIGURE 5.10 Certification of a statement.



$a[1{:}n]$. We first observe that if an array reference "$a[e]$" occurs within an expression, where e is a subscript expression, then the array reference can contain information about $a[e]$ or e.

Example:
Consider the statement

$$b := a[e] .$$

If e is known but $a[e]$ is not, then execution of this statement causes a flow $a[e] \rightarrow b$. If $a[i]$ is known for $i = 1, \ldots, n$ but e is not, it can cause a flow $e \rightarrow b$ (e.g., if $a[i] = i$ for $i = 1, \ldots, n$, then $b = e$). ■

We next observe that an assignment of the form "$a[e] := b$" can cause information about e to flow into $a[e]$.

Example:
If an assignment "$a[e] := 1$" is made on an all-zero array, the value of e can be obtained from the index of the only nonzero element in a. ■

If all elements $a[i]$ belong to the same class $\underline{a}$, the certification mechanism is easily extended to verify flows to and from arrays. For an array reference "$a[e]$", the class $\underline{a} \oplus \underline{e}$ can be associated with the reference to verify flows from a and e. For an array assignment "$a[e] := b$", the relation $\underline{e} \leq \underline{a}$ can be verified along with the relation $\underline{b} \leq \underline{a}$.

If the elements belong to different classes, it is necessary to check only the classes $\underline{a[i]}$ for those i in the range of e. This is because there can be no flow to or from $a[j]$ if e never evaluates to j (there must be a possibility of accessing an object for information to flow).

Example:
Given $a[1{:}4]$ and $b[1{:}4]$, the statement

if $x \leq 2$ **then** $b[x] := a[x]$

requires only that

$\underline{x} \oplus \underline{a[i]} \leq \underline{b[i]}$, $i = 1, 2$. ■

Performing such a flow analysis is beyond the scope of the present mechanism.

As a general rule, a mechanism is needed to ensure addresses refer to the objects assumed during certification. Otherwise, a statement like "$a[e] := b$" might cause an invalid flow $b \rightarrow c$, where c is an object addressed by $a[e]$ when e is out of range. There are several possible approaches to constructing such a mechanism. One method is for the compiler to generate code to check the bounds of array subscripts and pointer variables. The disadvantage of this approach is that it can substantially increase the size of a program as well as its running time. A more efficient method is possible if each array object in memory has a descriptor giving its bounds; the hardware can then check the validity of addresses in parallel with instruction execution. A third method is to prove that all subscripts and pointer variables are within their bounds; program proving is discussed in the next section.

The certification mechanism can be extended to control structures arising from arbitrary **goto** statements. Certifying a program with unrestricted **goto**s, however, requires a control flow analysis of the program to determine the objects receiving implicit flows. (This analysis is unnecessary if **goto**s are restricted—for example, to loop exits—so that the scope of conditional expressions can be determined during syntax analysis.) Following is an outline of the analysis required to do the certification. All **basic blocks** (single-entry, single-exit substructures) are

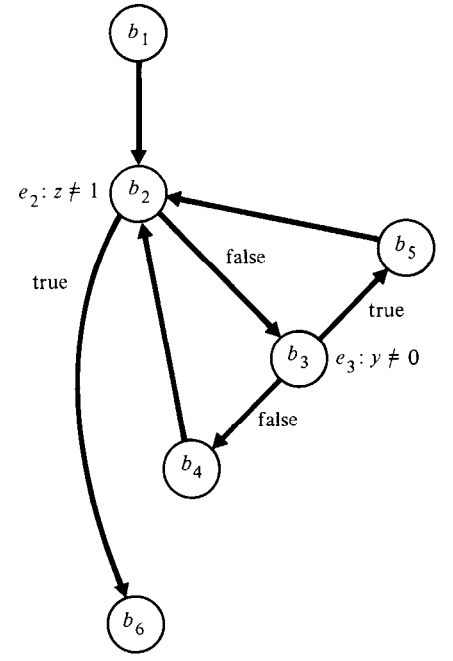
FIGURE 5.11 Procedure *copy2* with goto.

```
procedure copy2(x: integer class {x};
                var y: integer class {x});
   "copy x to y"
 var z: integer class {x};
 begin
1:    z := 1;                          b1
      y := -1;
2:    if z ≠ 1 then goto 6             b2
3:    y := y + 1;                      b3
      if y ≠ 0 then goto 5;
4:    z := x;                          b4
      goto 2;
5:    z := 0;                          b5
      goto 2;
6:    end                              b6
end copy2
```

identified. A **control flow graph** is constructed, showing transitions among basic blocks; associated with block b_i is an expression e_i that selects the successor of b_i in the graph (e.g., see [Alle70]). The security class of block b_i is the $\otimes$ of the classes of all objects that are the targets of flows in b_i (if there are no such objects, this class is *High*). The **immediate forward dominator** $IFD(b_i)$ is computed for each block b_i. It is the closest block to b_i among the set of blocks that lie on all paths from b_i to the program exit and, therefore, is the point where the divergent execution paths conditioned on e_i converge. Define B_i as the set of blocks on some path from b_i to $IFD(b_i)$ excluding b_i and $IFD(b_i)$. The security class of B_i is $\underline{B_i} = \otimes \{\underline{b_j} \mid b_j \in B_i\}$. Because the only blocks directly conditioned on the selector expression e_i of b_i are those in B_i, the program is secure if each block b_i is independently secure and $\underline{e_i} \leq \underline{B_i}$ for all i.

Example:
Figure 5.11 shows how the *copy2* procedure of Figure 5.9 can be written with **gotos** and partitioned into blocks. The control flow graph of the program is shown in Figure 5.12. Proof that the program is secure is left as an exercise. ■

FIGURE 5.12 Control flow graph of *copy2*.



$IFD(b_1) = b_2$
$IFD(b_2) = b_6$
$IFD(b_3) = IFD(b_4) = IFD(b_5) = b_2$

5.4.5 Concurrency and Synchronization

Reitman [Reit79] and Andrews and Reitman [Andr80] show that information can flow among concurrent programs over synchronization channels.

Example:
Suppose procedures p and q synchronize through a shared semaphore s as follows:

```
procedure p(x: integer class {x};
            var s: semaphore class {s, x});
  begin
    if x = 1 then signal(s)
  end
end p
procedure q(var y: integer class {s, y};
            var s: semaphore class {s, y});
  begin
    y := 0;
    wait(s);
    y := 1
  end
end q
```

where **wait**(s) delays q until p issues a **signal**(s) operation. Concurrent execution of these procedures causes an implicit flow of information from parameter x of p to parameter y of q over the synchronization channel associated with the semaphore s: if $x = 0$, y is set to 0, and q is delayed indefinitely on s; if $x = 1$, q is signaled, and y is set to 1. Thus, if p and q are invoked concurrently as follows:

```
cobegin
   p(a, s)
||
   q(b, s)
coend
```

the value of argument a flows into argument b. ■

The flow $x \rightarrow y$ in the preceding example is caused by **wait**(s) and **signal**(s), which read and modify the value of s as follows (the complete semantics of these operations are not shown):

	Read	Write
wait(s)	wait for $s > 0$	$s := s - 1$
signal(s)		$s := s + 1$

FIGURE 5.13 Synchronization flows.

```
procedure copy3(x: integer class {x};
                var y: integer class {x});
  "copy x to y"
  var s0: semaphore class {x};
      s1: semaphore class {x});
  cobegin
      "Process 1"
      if x = 0 then signal(s0) else signal(s1)
   ||
      "Process 2"
      wait(s0); y := 1; signal(s1);
   ||
      "Process 3"
      wait(s1); y := 0; signal(s0);
  coend
end copy3
```

Therefore, execution of the **if** statement in p causes an implicit flow from x to s, causing the value of x to flow into q.

When $x = 0$, q is left waiting on semaphore s. Because this is similar to a nonterminating **while** loop, we might wonder if all synchronization channels are associated with abnormal terminations from timeouts. If so, they would have a channel capacity of at most 1 bit, and could be handled as described in the next section. Reitman and Andrews show, however, that information can flow along synchronization channels even when the procedures terminate normally.

Example:
Consider the program *copy3* shown in Figure 5.13. When $x = 0$, process 2 executes before process 3, so the final value of y is 0; when $x \neq 0$, process 3 executes before process 2, so the final value of y is 1. Hence, if x is initially 0 or 1, execution of *copy3* sets y to the value of x. ■

Because each statement logically following a **wait**(s) operation is conditioned on a **signal**(s) operation, there is an implicit flow from s to every variable that is the target of an assignment in a statement logically following the **wait**. To ensure the security of these flows, Reitman and Andrews require the class of every such variable y satisfy the relation $\underline{s} \leq \underline{y}$.

With parallel programs, information can also flow over global channels associated with loops.

Example:
Consider the program *copy4* shown in Figure 5.14. Execution of *copy4* trans-

FIGURE 5.14 Global flows in concurrent programs.

```
procedure copy4(x: integer class {x};
                var y: integer class {x});
  "copy x to y"
  var e0, e1: boolean class {x};
  begin
    e0 := e1 := true ;
    cobegin
       if x = 0 then e0 := false else e1 := false
    ||
       begin
         while e0 do ;
         y := 1;
         e1 := false
       end
    ||
       begin
         while e1 do ;
         y := 0;
         e0 := false
       end
    coend
  end
end copy4
```

fers information from x to y, with variables $e0$ and $e1$ playing the role of semaphores $s0$ and $s1$. ■

Our present certification mechanism would not verify the relations $\underline{e0} \leq \underline{y}$ and $\underline{e1} \leq \underline{y}$ in the preceding example, because the assignments to y are outside the loop bodies. To verify these "global flows", Reitman and Andrews consider the expression e of a statement "**while** e **do** S_1" to have a global scope that includes any statement logically following the **while**; the relation $\underline{e} \leq \underline{y}$ is verified for every object y that is the target of an assignment in the global scope. A "**while** e", therefore, is treated like a "**wait**(e)". Extending the certification semantics of Table 5.4 to verify synchronization and global flows in parallel programs is left as an exercise.

Reed and Kanodia [Reed79] observed that synchronization flows can also emanate from a waiting process; because **wait**(s) modifies the value of s, another waiting process can be blocked or delayed as a result. They show that by using **eventcounts** for process synchronization, information flows are restricted to signalers.

A signaling process can covertly leak a value by making the length of delay

proportional to the value. The problem is the same as with loops, and there does not appear to be a satisfactory solution to either.

5.4.6 Abnormal Terminations

Information can flow along covert channels associated with abnormal program terminations.

Example:
Consider the following copy procedure:

```
procedure copy5(x: integer class {x};
                var y: integer class {});
  "insecure procedure that leaks x to y"
  begin
    y := 0;
    while x = 0 do ;
    y := 1
  end
end copy5 .
```

If $x = 0$, then y becomes 0, and the procedure hangs in the loop; if $x = 1$, then y becomes 1, and the procedure terminates. ■

The flow $x \rightarrow y$ in the *copy5* procedure occurs because the statement "$y := 1$" is conditioned on the value of x; thus there is an implicit flow $x \rightarrow y$, even though y is not the target of a flow in the statements local to the loop.

Such covert flows are not confined to nonterminating loops.

Example:
If the **while** statement in *copy5* is replaced by the statement:

if $x = 0$ **then** $x := 1/x$;

the value of x can still be deduced from y; if $x = 0$, the procedure abnormally terminates with a divide-by-zero exception and $y = 0$; if $x = 1$, the procedure terminates normally with $y = 1$. ■

Indeed, the nonterminating **while** statement could be replaced by any action that causes abnormal program termination: end-of-file, subscript-out-of-range, etc. Furthermore, the leak occurs even without the assignments to y, because the value of x can be determined by whether the procedure terminates normally.

Example:
Consider this copy procedure:

```
procedure copy6(x: integer class {x};
                var y: integer class {}) ;
  "insecure procedure that leaks x to y"
  var sum: integer class {x};
      z: integer class {};
  begin
    z := 0;
    sum := 0;
    y := 0;
    while z = 0 do
      begin
        sum := sum + x;
        y := y + 1;
      end
  end
end copy6 .
```

This procedure loops until the variable *sum* overflows; the procedure then terminates, and x can be approximated by MAX/y, where MAX is the largest possible integer. The program trap causes an implicit flow $x \rightarrow y$ because execution of the assignment to y is conditioned on the value of *sum*, and thus x, but we do not require that $\underline{x} \leq \underline{y}$. ■

The problem of abnormal termination can be handled by inhibiting all traps except those for which actions have been explicitly defined in the program [Denn77]. Such definitions could be made with a statement similar to the **on** statement of PL/I:

on condition **do** statement ,

where "condition" names a trap condition (overflow, underflow, end-of-file, divide-by-zero, etc.) for some variable, and "statement" specifies the action to be taken. When the trap occurs, the statement is executed, and control then returns to the point of the trap. The security requirements for programs can then be extended to ensure the secure execution of all programmer defined traps.

Example:
If the statement

on overflow *sum* **do** $z := 1$

were added to the *copy6* procedure, the security check $\underline{sum} \leq \underline{z}$ would be made, and the procedure would be declared insecure. ■

This still leaves the problem of undefined traps and abnormal terminations from timeouts and similar events (overflowing a page limit or punched card limit, or running out of memory or file space). Many such problems can be eliminated by proving properties about the security and correctness of procedures; this is

discussed in the next section. A mechanism that detects and logs abnormal termination of certified procedures is also essential.

5.5 PROGRAM VERIFICATION

Andrews and Reitman [Andr80] developed a deductive system for information flow based on the lattice model and on Hoare's [Hoar69] deductive system for functional correctness. The flow requirements of a procedure q are expressed as **assertions** about the values and classes of objects before and after q executes. Given an assertion P (precondition) about the initial state of q, and an assertion Q (postcondition) about the final state of q, q's security is proved (or disproved) by showing that Q satisfies the security requirements, and that if P is true on entry to q, Q will be true on exit. The proof is constructed by inserting intermediate assertions into the program, and applying **axioms** and **proof rules** defined over the statements of the language.

The approach has two advantages over the simple certification mechanism. First, it provides a single system for proving both security and correctness. Second, it gives a more precise enforcement mechanism for security.

Example:
Consider this procedure:

procedure *copy7*(x, a, b: **integer**; **var** y: **integer**);
  if $x = 1$ **then** $y := a$ **else** $y := b$
end *copy7* .

Because of the implicit flow from x to y and the explicit flows from a and b to y, the certification mechanism requires the following relations hold for any execution of *copy7*:

$$\underline{x} \leq \underline{y}, \underline{a} \leq \underline{y}, \text{ and } \underline{b} \leq \underline{y} .$$

This is stronger than necessary, because any particular execution will transfer information from either a into y or from b into y but not both. ■

To achieve a more precise enforcement mechanism, the flow logic assigns a variable v to a variable class $\underline{\underline{v}}$ representing the **information state** of v at any given point in the program. An assertion about the information state of v is of the form

$$\underline{\underline{v}} \leq C,$$

where C is a security class. The assertion states the information in v is no more sensitive than C. The bound C may be expressed as a specific class, as in $\underline{\underline{v}} \leq Low$, which states v contains information belonging to the lowest security class only. This assertion would hold, for example, after the unconditional execution of the assignment "$v := 0$". The bound may be expressed as the fixed class of v, as in $\underline{\underline{v}} \leq \underline{v}$, which states that v contains information no more sensitive than its class $\underline{v}$

permits. The bound may also be expressed as a join of the information states of other variables, as in $\underline{v} \leq \underline{u}_1 \oplus \ldots \oplus \underline{u}_n$, which states v contains information no more sensitive than that in $u_1, \ldots, u_n$.

A special class $\underline{pc}$ represents the information state of the program counter (Andrews and Reitman call this class *local*); $\underline{pc}$ simulates the program counter class $\underline{pc}$ in Fenton's Data Mark Machine (see Section 5.3.3), and is used to prove properties about implicit flows.

Assertions about the classes of variables can be combined with assertions about their values. An assertion about the final state of the *copy7* procedure is given by:

$$\{\underline{pc} \leq PC_{c7},$$
$$(x = 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{a} \oplus PC_{c7}) \, , \, (x \neq 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{b} \oplus PC_{c7})\} \, ,$$

where PC_{c7} represents a bound on $\underline{pc}$ upon entering *copy7*. This says when $x = 1$, the information in y is no more sensitive than that in x, a, and the program counter pc; when $x \neq 1$, the information in y is no more sensitive than that in x, b, and pc.

In the flow logic, execution of an assignment statement

$$v := f(u_1, \ldots, u_n)$$

is described by the assertion

$$\underline{v} \leq \underline{u}_1 \oplus \ldots \oplus \underline{u}_n \, ,$$

relating the information state of v to that of $u_1, \ldots, u_n$. Note that this differs from the certification mechanism of the previous section, where the assignment statement imposes the security condition

$$\underline{u}_1 \oplus \ldots \oplus \underline{u}_n \leq \underline{v} \, ,$$

relating the fixed security class of v to that of $u_1, \ldots, u_n$. Section 5.5.6 shows how security conditions relate to assertions in the flow logic.

An assertion P will be expressed as a predicate in first-order calculus using "," for conjunction, "$\supset$" for implication, "$\sim$" for negation, and "$\forall$" for universal quantification (we shall not use disjunction or existential quantification in our examples).

Let P and Q be assertions about the information state of a program. For a given statement S,

$$\{P\} \; S \; \{Q\}$$

means if the **precondition** P is true before execution of S, then the **postcondition** Q is true after execution of S, assuming S terminates. The notation

$$P[x \leftarrow y]$$

means the assertion P with every free occurrence of x replaced with y.

The simplest proof rule is the rule of consequence:

Rule of consequence:

Given: $P \supset P', \{P'\}\ S\ \{Q'\}, Q' \supset Q$
Conclusion: $\{P\}\ S\ \{Q\}$ ■

The axioms and proof rules for each statement type are now given in Sections 5.5.1–5.5.5.

5.5.1 Assignment

Let S be the statement

$$b := e\,,$$

and let P be an assertion about the information state after S is executed. The axiom for correctness is simply

$$\{P[b \leftarrow e]\}\ b := e\ \{P\}\,,$$

because any assertion about b that is true after the assignment must be true of the value of e before the assignment. Note, however, the change in b is reflected in the postcondition P, and not the precondition.

Example:
Consider the statement

$$y := x + 1\,.$$

The axiom can be used to prove that the precondition $\{0 \leq x \leq 4\}$ implies the postcondition $\{1 \leq y \leq 5\}$:

$\{0 \leq x \leq 4\}$
$\{1 \leq (x + 1) \leq 5\}$
$y := x + 1$
$\{1 \leq y \leq 5\}\,.$

The second assertion follows from the first by simple implication. The third follows from the second (and the effect of executing the assigment statement) by the axiom for assignment. ■

The precondition $\{P[b \leftarrow e]\}$ of the assignment axiom is called a **weakest precondition** because it is the weakest condition needed to establish the postcondition. Of course, the rule of consequence always allows us to substitute stronger conditions.

Example:
The following is also true:

$$\{x = 3, z = 0\}$$

$y := x + 1$
$\{1 \leq y \leq 5\}$. ■

In general, we would like to establish the minimal precondition needed to prove security and correctness.

Extending the assignment axiom to include information flow, we must account for both the explicit flow from e and the implicit flow from pc. We get

Assignment axiom:

$\{P[b \leftarrow e; \underline{b} \leftarrow \underline{e} \oplus \underline{pc}]\}\ b := e\ \{P\}$ ■

This states the sensitivity of b after the assignment will be no greater than that of e and pc before the assignment.

Example:
Suppose execution of the statement "$y := x + 1$" is conditioned on the value of variable z. Given the precondition $\{0 \leq x \leq 4, \underline{pc} \leq \underline{z}\}$, we can prove:

$\{0 \leq x \leq 4, \underline{pc} \leq \underline{z}\}$
$\{0 \leq x \leq 4, \underline{x} \oplus \underline{pc} \leq \underline{x} \oplus \underline{z}, \underline{pc} \leq \underline{z}\}$
$y := x + 1$
$\{1 \leq y \leq 5, \underline{y} \leq \underline{x} \oplus \underline{z}, \underline{pc} \leq \underline{z}\}$

The postcondition $\{\underline{y} \leq \underline{x} \oplus \underline{z}\}$ states that the information in y after the statement is executed is no more sensitive than that in x and z. ■

5.5.2 Compound

Let S be the statement

begin $S_1; \ldots; S_n$ **end** .

The proof rule for correctness is

Compound rule:

Given: $\{P_i\}\ S_i\ \{P_{i+1}\}$ for $i = 1, \ldots, n$
Conclusion: $\{P_1\}$ **begin** $S_1; \ldots; S_n$ **end** $\{P_{n+1}\}$ ■

No extensions are needed to handle information flow security.

Example:
Consider the statement

begin "compute $y = x$ mod n for $x \geq 0$"
  $i := x$ div n;
  $y := x - i * n$
end

The following proves $y = x \bmod n$ when $x \geq 0$ and $n > 0$, and $\underline{y} \leq \underline{x} \oplus \underline{n}$ when $\underline{\underline{pc}} \leq Low$:

$\{x \geq 0,\ n > 0,\ \underline{\underline{pc}} \leq Low\}$
begin
$\{0 \leq x - (x \text{ div } n)*n < n,\ \underline{x} \oplus \underline{n} \leq \underline{x} \oplus \underline{n},\ \underline{\underline{pc}} \leq Low\}$
$i := x \text{ div } n;$
$\{0 \leq x - i*n < n,\ \underline{i} \leq \underline{x} \oplus \underline{n},\ \underline{\underline{pc}} \leq Low\}$
$\{0 \leq x - i*n < n,\ (x - i*n - x) \bmod n = 0,$
$\underline{x} \oplus \underline{i} \oplus \underline{n} \leq \underline{x} \oplus \underline{n},\ \underline{\underline{pc}} \leq Low\}$
$y := x - i*n$
end
$\{0 \leq y < n,\ (y - x) \bmod n = 0,\ \underline{y} \leq \underline{x} \oplus \underline{n},\ \underline{\underline{pc}} \leq Low\}$.

Because the initial value of i does not flow into y, its class does not appear in the postcondition. The proof consists of a sequence of statements and assertions. Each assertion follows from its preceding assertion either by simple implication (e.g., the second and fourth assertions), or by the axioms and proof rules (e.g., the third and fifth assertions). ■

5.5.3 Alternation

Let S be the statement

if e **then** S_1 **else** S_2 ,

where S_2 is the null statement when there is no **else** part. The proof rule for correctness follows:

Given: $\{P, e\}\ S_1\ \{Q\}$
$\{P, \sim e\}\ S_2\ \{Q\}$
Conclusion: $\{P\}$ **if** e **then** S_1 **else** S_2 $\{Q\}$ ■

Example:
Consider the statement

if $x \geq 0$ **then** $y := x$ **else** $y := -x$.

We can prove that execution of this statement sets $y = |x|$ for any initial value of x:

$\{\}$
if $x \geq 0$
$\{x \geq 0\}$
$\{x \geq 0,\ x = |x|\}$
then $y := x$
$\{y = |x|\}$
– – – – – – – – – –
$\{x < 0\}$

$\{x < 0, -x = |x|\}$
else $y := -x$
$\{y = |x|\}$
$\{y = |x|\}$. ■

To extend the proof rule to secure information flow, we must account for the implicit flow from e into S_1 and S_2 via the program counter. To do this, the precondition P is written as a conjunction $P = \{V, L\}$, where L is an assertion about $\underline{\underline{pc}}$ and V is an assertion about the remaining information state. Similarly, the postcondition Q is written as a conjunction $Q = \{V', L\}$. On entry to (and exit from) S_1 and S_2, L is replaced with an assertion L' such that $\{V, L\}$ implies the assertion $L'[\underline{\underline{pc}} \leftarrow \underline{\underline{pc}} \oplus \underline{e}]$. This allows the class $\underline{\underline{pc}}$ to be temporarily increased by $\underline{e}$ on entry to S_1 or S_2, and then restored on exit. Because $\underline{\underline{pc}}$ is restored after execution of the **if** statement, the assertion L is invariant (i.e., it is the same in P and Q); however, the assertion V may be changed by S_1 or S_2. We thus have the following rule:

Alternation rule:

Given: $P = \{V, L\}, Q = \{V', L\}$
$\{V, e, L'\}\ S_1\ \{V', L'\}$
$\{V, \sim e, L'\}\ S_2\ \{V', L'\}$
$P \supset L'\ [\underline{\underline{pc}} \leftarrow \underline{\underline{pc}} \oplus \underline{e}]$
Conclusion: $\{P\}$ **if** e **then** S_1 **else** $S_2\ \{Q\}$ ■

Example:
Consider the statement

if $x = 1$ **then** $y := a$ **else** $y := b$

of the procedure *copy7*. We can prove for all x, a, and b,

$\{\underline{\underline{pc}} \leq Low\}$
if $x = 1$
$\{x = 1, \underline{\underline{pc}} \leq \underline{x}\}$
$\{x = 1, \underline{\underline{pc}} \oplus \underline{a} \leq \underline{x} \oplus \underline{a}, \underline{\underline{pc}} \leq \underline{x}\}$
then $y := a$
$\{x = 1, \underline{y} \leq \underline{x} \oplus \underline{a}, \underline{\underline{pc}} \leq \underline{x}\}$
- - - - - - - - - - - - - - - - - -
$\{x \neq 1, \underline{\underline{pc}} \leq \underline{x}\}$
$\{x \neq 1, \underline{\underline{pc}} \oplus \underline{b} \leq \underline{x} \oplus \underline{b}, \underline{\underline{pc}} \leq \underline{x}\}$
else $y := b$
$\{x \neq 1, \underline{y} \leq \underline{x} \oplus \underline{b}, \underline{\underline{pc}} \leq \underline{x}\}$
$\{\underline{\underline{pc}} \leq Low,$
$(x = 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{a}), (x \neq 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{b})\}$. ■

5.5.4 Iteration

Let S be the statement

while e **do** S_1 .

The proof rule for correctness is

Given: $\{P, e\}\ S_1\ \{P\}$
Conclusion: $\{P\}$ **while** e **do** $S_1\ \{P, \sim e\}$ ■

where P is the loop invariant. This is extended to security as for alternation:

Iteration rule:
Given: $P = \{V, L\}$
$\{V, L', e\}\ S_1\ \{V, L'\}$
$P \supset L'\ [\underline{pc} \leftarrow \underline{pc} \oplus \underline{e}]$
Conclusion: $\{P\}$ **while** e **do** $S_1\ \{P, \sim e\}$ ■

To handle parallel programs, a slightly more complicated rule is needed. To certify "global flows" from the loop condition e to statements outside S_1 that logically follow the **while** statement, the assertions P and Q are expressed as the conjunctions

$P = \{V, L, G\}$
$Q = \{V, \sim e, L, G'\}$,

where G is an assertion about a special class $\underline{\underline{global}}$, and

$P \supset G'[\underline{\underline{global}} \leftarrow \underline{\underline{global}} \oplus \underline{pc} \oplus \underline{e}]$.

Unlike $\underline{pc}$, $\underline{\underline{global}}$ is never restored; instead it continually increases, because all subsequent statements of the program are conditioned on loop terminations.

The class $\underline{\underline{global}}$ is also used to certify synchronization flows (see exercises at end of chapter).

5.5.5 Procedure Call

Let q be a procedure of the form:

procedure $q(x;$ **var** $y)$;
local variable declarations;
$\{P, \underline{pc} \leq PC_q\}$
S
$\{Q\}$
end q ,

where x is a vector of input parameters, and y is a vector of output parameters (or input/output parameters). The body S may reference global variables z and global constants. We assume all variables in x, y, and z are disjoint (see [Grie80] for proof rules for overlapping variables).

We assume the security proof of q is of the form

$$\{P, \underline{\underline{pc}} \leq PC_q\}\ S\ \{Q\}\ ,$$

where P is a precondition that must hold at the time of the call, PC_q is a placeholder for a bound on $\underline{\underline{pc}}$ at the time of the call, and Q is a postcondition that holds when q terminates. We assume Q does not reference the local variables of q. Q may reference the value parameters x; we assume all such references are to the initial values and classes passed to q (assignments to x in q do not affect the actual parameters anyway). Q may reference the variable parameters y and global variables z; the initial values and classes of these variables at the time of call are referenced by y', $\underline{y}'$ and z', $\underline{z}'$.

Consider a call of the form

$$q(a; b)\ ,$$

where a is a vector of arguments corresponding to the parameters x, and b is a vector of arguments corresponding to the parameters y.

Reitman and Andrews give a proof rule for verifying the security of procedure calls, but not their correctness. We shall instead use the following proof rule, which is an extension of correctness proof rules (e.g., see [Grie80,Lond78]), allowing for the substitution of classes as well as values:

Procedure call rule:

Given: $\{P, \underline{\underline{pc}} \leq PC_q\}\ S\ \{Q\}$

Conclusion:
$$\{P[(x, \underline{x}) \leftarrow (a, \underline{a}); (y, \underline{y}) \leftarrow (b, \underline{b}); PC_q \leftarrow PC],$$
$$\forall u, v:$$
$$Q[(x, \underline{x}) \leftarrow (a, \underline{a});$$
$$(y, \underline{y}) \leftarrow (u, \underline{u}); (y', \underline{y}') \leftarrow (b, \underline{b});$$
$$(z, \underline{z}) \leftarrow (v, \underline{v}); (z', \underline{z}') \leftarrow (z, \underline{z}); PC_q \leftarrow PC] \supset$$
$$R[(b, \underline{b}) \leftarrow (u, \underline{u}); (z, \underline{z}) \leftarrow (v, \underline{v})]\}$$
$$q(a, b)$$
$$\{R\}\ \blacksquare$$

where PC is a bound on $\underline{\underline{pc}}$ at the time of the call; that is, $\underline{\underline{pc}} \leq PC$ is implied by the precondition of the call. The assertion $\{\forall u, v{:}Q[] \supset R[]\}$ says R can be true after the call if and only if for all possible values u assigned to the variable parameters y, and all possible values v assigned to the global variables z, the postcondition Q implies R before the call when a is substituted for x, b for y', z for z', and the current bound PC for PC_q.

Example:

Consider the following procedure, with precondition and postcondition as shown:

```
procedure f(var y1, y2: integer);
 {pc ≤ PC_f}
  begin
    y2 := y1;
    y1 := 0
  end
```

$\{\underline{y1} \leq PC_f, \underline{y2} \leq \underline{y1}' \oplus PC_f, \underline{pc} \leq PC_f\}$
end *f* .

Note the precondition of *f* consists only of an assertion about $\underline{pc}$ (i.e., the condition *P* in the proof rule is "true"). The postcondition states the final value of *y1* is no more sensitive than the program counter, and that of *y2* is no more sensitive than the initial value of *y1* and the program counter. Consider a call "*f*(*b1*, *b2*)". Given the precondition $\{\underline{pc} \leq Low, \underline{b1} \leq High, \underline{b2} \leq Low\}$, we can prove execution of *f* lowers $\underline{b1}$ but raises $\underline{b2}$ as follows:

$\{\underline{b1} \leq High, \underline{b2} \leq Low, \underline{pc} \leq Low\}$
$\{\underline{pc} \leq Low,$
$\quad \forall u1, u2: (\underline{u1} \leq Low, \underline{u2} \leq High, \underline{pc} \leq Low) \supset,$
$\quad\quad (\underline{u1} \leq Low, \underline{u2} \leq High, \underline{pc} \leq Low)\}$
$f(b1, b2)$
$\{\underline{b1} \leq Low, \underline{b2} \leq High, \underline{pc} \leq Low\}$. ■

Example:
Consider the following procedure, which deciphers a string *x* (we shall assume the key is part of the decipher function):

procedure *decipher*(*x*: **string** "ciphertext";
var *y*: **string** "plaintext");
"decipher *x* into *y*"
$\{\underline{pc} \leq PC_d\}$
"decipher transformation"
$\{\underline{y} \leq \underline{x} \oplus PC_d, \underline{pc} \leq PC_d\}$
end *decipher* .

The postcondition states the final state of *y* is no more sensitive than the state of *x* and the program counter.

Now, consider the procedure shown in Figure 5.15, which calls *deci-*

FIGURE 5.15 Procedure *getmsg*.

```
procedure getmsg(c: string; var p: string);
 "get message: decipher c and return the
  corresponding plaintext p if the authority
  of the calling program is at least level 2
  (Secret); the authority is represented by
  the global constant a"
 {a ≤ Low, pc ≤ PC_g}
 if a ≥ 2
   then decipher (c, p)
   else p := null
 {a ≤ Low, pc ≤ PC_g,
  (a ≥ 2) ⊃ (p ≤ c ⊕ PC_g),   (a < 2) ⊃ (p ≤ PC_g)}
end getmsg
```

pher. The postcondition states the final class of p is bounded by $\underline{c} \oplus PC_g$ when $a \geq 2$, and by PC_g when $a < 2$, where PC_g is a placeholder for a bound on $\underline{pc}$ at the time of the call. Because the initial value of p is lost, its initial sensitivity does not affect its final sensitivity. The authority level a is in the lowest class. We shall now prove that the precondition of *getmsg* implies the postcondition.

$\{\underline{a} \leq Low, \underline{pc} \leq PC_g\}$
if $a \geq 2$
$\{a \geq 2, \underline{a} \leq Low, \underline{pc} \leq PC_g\}$
$\{\underline{pc} \leq PC_g,$
$\forall u:(\underline{u} \leq \underline{c} \oplus PC_g, \underline{pc} \leq PC_g) \supset$
$(a \geq 2, \underline{a} \leq Low, \underline{u} \leq \underline{c} \oplus PC_g, \underline{pc} \leq PC_g)\}$
then *decipher*(c, p)
$\{a \geq 2, \underline{a} \leq Low, \underline{p} \leq \underline{c} \oplus PC_g, \underline{pc} \leq PC_g\}$
- -
$\{a < 2, \underline{a} \leq Low, \underline{pc} \leq PC_g\}$
else $p :=$ null
$\{a < 2, \underline{a} \leq Low, \underline{p} \leq PC_g, \underline{pc} \leq PC_g\}$
$\{\underline{a} \leq Low, \underline{pc} \leq PC_g,$
$(a \geq 2) \supset (\underline{p} \leq \underline{c} \oplus PC_g), (a < 2) \supset (\underline{p} \leq PC_g)\}$. ■

5.5.6 Security

Proving a program satisfies its assertions does not automatically imply the program is secure. The assertions must also satisfy the security requirements imposed by the flow policy.

Given a statement S and a proof $\{P\}\ S\ \{Q\}$, execution of S is secure if and only if:

1. P is initially true,
2. For every object v assigned to a fixed security class $\underline{v}$, $Q \supset \{\underline{v} \leq \underline{v}\}$

The second requirement states that the information represented in the final value of v (represented by $\underline{v}$) must be no more sensitive than that permitted by the class $\underline{v}$.

Example:
Earlier we proved:

$\{\underline{pc} \leq Low\}$
if $x = 1$ **then** $y := a$ **else** $y := b$
$\{\underline{pc} \leq Low,$
$(x = 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{a}), \quad (x \neq 1) \supset (\underline{y} \leq \underline{x} \oplus \underline{b})\}$.

Let $x = 1$, $\underline{x} = Low$, $\underline{a} = Low$, $\underline{b} = High$, $\underline{y} = Low$, and $\underline{pc} = Low$. Since

the postcondition does not contain any assertions about the sensitivity of information in x, a, or b, it does not imply that $\underline{x} \leq Low$, $\underline{a} \leq Low$, $\underline{b} \leq High$, and $\underline{y} \leq Low$. To show execution of this statement is secure, we instead prove:

$\{x = 1, \underline{x} \leq Low, \underline{a} \leq Low, \underline{b} \leq High, \underline{pc} \leq Low\}$
if $x = 1$ **then** y := a **else** $y := b$
$\{x = 1, \underline{x} \leq Low, \underline{a} \leq Low, \underline{b} \leq High, \underline{y} \leq \underline{x} \oplus \underline{a}, \underline{pc} \leq Low\}$
$\{x = 1, \underline{x} \leq Low, \underline{a} \leq Low, \underline{b} \leq High, \underline{y} \leq Low, \underline{pc} \leq Low\}$. ■

Example:
Consider the following statement, which passes objects c and p to the procedure *getmsg* of Figure 5.15 (we have chosen the same names for our actual arguments as for the formal parameters of g)

$getmsg(c, p)$.

Let $a = 3$ (*Top Secret*), $\underline{a} = 0$ (*Unclassified* or *Low*), $\underline{c} = 2$ (*Secret*), $\underline{p} = 3$, and $PC = 3$. Because $a \geq 2$, the ciphertext c will be deciphered, and the result assigned to p. We can prove:

$\{a = 3, \underline{a} \leq Low, \underline{c} \leq 2, \underline{pc} \leq 3\}$
$\{\underline{a} \leq Low, \underline{pc} \leq 3,$
$\forall u:(\underline{a} \leq Low, \underline{u} \leq 3, \underline{pc} \leq 3) \supset$
$(a = 3, \underline{a} \leq Low, \underline{c} \leq 2, \underline{u} \leq 3, \underline{pc} \leq 3)\}$
$getmsg(c, p)$
$\{a = 3, \underline{a} \leq Low, \underline{c} \leq 2, \underline{p} \leq 3, \underline{pc} \leq 3\}$.

Because the postcondition implies the contents of each object is no more sensitive than its fixed class, the preceding call is secure.

Next let $a = 1$ (*Confidential*), $\underline{a} = 0$, $\underline{c} = 2$, $\underline{p} = 1$, and $PC = 1$. Here $a < 2$, so c will not be deciphered, and p will be assigned the null value. We can prove:

$\{a = 1, \underline{a} \leq Low, \underline{c} \leq 2, \underline{pc} \leq 1\}$
$\{\underline{a} \leq Low, \underline{pc} \leq 1,$
$\forall u:(\underline{a} \leq Low, \underline{u} \leq 1, \underline{pc} \leq 1) \supset$
$(a = 1, \underline{a} \leq Low, \underline{c} \leq 2, \underline{u} \leq 1, \underline{pc} \leq 1)\}$
$getmsg(c, p)$
$\{a = 1, \underline{a} \leq Low, \underline{c} \leq 2, \underline{p} \leq 1, \underline{pc} \leq 1\}$.

Again the postcondition implies execution of the call is secure.

If $a \geq 2$ and $\underline{p} < \underline{c}$, we cannot prove execution of the call is secure (because it is not secure!—see exercises at the end of Chapter). Assuming only verified programs are allowed to execute, this means a top secret process, for example, cannot cause secret ciphertext to be deciphered into a lower classified variable. ■

Once a procedure has been formally verified, we would like assurances it has

not been tampered with or replaced by a Trojan Horse. This is a good application for digital signatures (see Section 1.3.1). The verification mechanism would sign the code for a procedure after its security had been proved, and the operating system would not load and execute any code without a valid signature.

5.6 FLOW CONTROLS IN PRACTICE

5.6.1 System Verification

Efforts to build verifiably secure systems have been motivated largely by the security requirements of the Department of Defense. Prior to the 1970s, no commercially available system had withstood penetration, and no existing system could adequately enforce multilevel security. To deal with this problem, classified information was processed one level at a time, and the system was shut down and cleared before processing information at another level. This was a costly and cumbersome mode of operation.

In the early 1970s, the Air Force Electronic Systems Division sponsored several studies aimed at developing techniques to support the design and verification of multilevel security systems. The methodology that emerged from these studies was founded on the concept of security kernel (see Section 4.6.1), and was reported in papers by Anderson [Ande72], Schell, Downey, and Popek [Sche72], Bell and Burke [Bell74], and Bell and LaPadula [Bell73]. By the late 1970s, the methodology had been applied (to various degrees) to the design and verification of several security kernels including those listed in Section 4.6.1.

A security kernel is specified in terms of states and state transitions, and verified following the general approach described in Section 4.6.3. The multilevel security requirements of the kernel are based on a model introduced by Bell and LaPadula at MITRE [Bell73]. The model assumes objects have fixed security classes; this assumption is formalized by an axiom called the **tranquility principle**. Multilevel security is given in terms of two axioms called the simple security condition and the *-property (pronounced "star-property"). The **simple security condition** states that a process p may not have read access to an object x unless $\underline{x} \leq \underline{p}$. The ***-property** states that a process may not have read access to an object x and write access to an object y unless $\underline{x} \leq \underline{y}$. This is stronger than necessary, because p may not cause any information flow from x to y; applications of the model to security kernel verification have relaxed this to require $\underline{x} \leq \underline{y}$ only when y is functionally dependent on x.

Researchers at MITRE [Mill76,Mill81] and at SRI [Feie77,Feie80] have developed techniques for verifying multilevel security in formal program specifications expressed as V- and O-functions (see Section 4.6.3). Flows from state variables are specified by references to primitive V-functions, and flows into state variables by changes to primitive V-functions (specified by the effects of O-functions). Security is verified by proving the security classes of the state variables and of the process executing the specified function satisfy the conditions for simple security and the *-property. The principal difference between security proofs for

specifications and those for programs (such as described in the preceding sections) is that specifications are nondeterministic; that is, the statements in the effects of an *O*-function are unordered.

Feiertag [Feie80] developed a model and tool for verifying multilevel security for *V*- and *O*-function specifications written in SPECIAL. The tool is part of SRI's Hierarchical Design Methodology HDM (see Section 4.6.3). A formula generator constructs formulas from the specifications that either prove or disprove a module is multilevel secure. Nontrivial formulas are passed to the Boyer-Moore theorem prover, where they are proved (or disproved) to be theorems. The user specifies the security classes of the state variables and the partial ordering relation $\leq$, so the tool is not limited to the military classification scheme. The Feiertag model is the basis for the PSOS secure object manager [Feie77,Neum80]. The tool has been used to verify the specifications of both the KSOS-11 and KSOS-6 (SCOMP) kernels. Verification of the KSOS kernel revealed several insecure channels in the design, some of which will remain as known potential low-bandwidth signaling paths [Neum80].

MITRE [Mill81] has also developed an automated tool for analyzing flows in modules written in a specification language. The flow analyzer associates semantic attributes with the syntactic types of the specification language in much the same way as in the compiler-based mechanism described in Section 5.4.3. This information is then used to produce tables of formulas which, if true, give sufficient conditions for security (as in Feiertag's tool). The analyzer is generated using the YACC parser generator on UNIX, so it can be easily adapted to different specification languages; analyzers for subsets of SPECIAL and Ina Jo (the specification language developed at SDC) have been implemented.

The MITRE flow analyzer is based in part on the deductive formulation of information flow developed jointly by Millen [Mill78] and Furtek [Furt78]. The deductive formulation describes information flow in a module by transition **constraints** on the values of the state variables on entry to and exit from the module. Such a constraint is of the form

$$x1_{u1} \ldots xk_{uk} \times y_v ,$$

where $x1, \ldots, xk$, and y are state variables; and $u1, \ldots, uk$, and v represent their respective values. The constraint states that if xi has the value ui ($i = 1, \ldots, k$) on entry to the module, then y cannot have the value v on exit from the module. Thus, the condition on the right of the "×" represents an impossible exit condition given the entry condition on the left. The constraint thus limits the possible values of the variables, making it possible to deduce something about the value of one variable from the values of other variables.

Example:

Consider the statement (formal specification)

$$'y = x ,$$

relating the new (primed) value of y to the old (unprimed) value of x. This statement is described by the set of constraints

$$\{x_u \times y_v \mid v \neq u\} ,$$

which states that if x initially has the value u, then the new value of y cannot be v for any $v \neq u$. Thus the initial value of x can be deduced from the new value of y, and the security requirements of the statement are given by the condition

$$\underline{x} \preceq \underline{y} . \quad \blacksquare$$

Example:
The specification statement

if $x = 0$ **then** $'y = 0$ **else** $'y = 1$

is described by the set of constraints

$$\{x_0 \times y_v \mid v \neq 0\} \cup \{x_u \times y_v \mid u \neq 0, v \neq 1\} ,$$

which shows that access to the new value of y can be used to deduce something about the initial value of x. The security requirements of this statement are given by the same condition as the previous statement. ■

The MITRE flow analyzer can handle arrays of elements belonging to different classes.

Example:
Consider the specification statement

if $x < 2$ **then** $'y = a[x]$.

This statement is described by the set of constraints

$$\{x_i\, a[i]_u \times y_v \mid v \neq u, i < 2\} .$$

The security requirements of the statement are thus given by

$$\underline{x} \preceq \underline{y}$$
$$\underline{a[i]} \preceq \underline{y} \text{ for } i < 2 . \quad \blacksquare$$

Rushby's [Rush81] approach to the design and verification of secure systems (see Section 4.6.3) seems to obviate the need to prove multilevel flow security for a security kernel. Since the kernel serves to isolate processes sharing the same machine, it suffices to prove such an isolation is achieved. Proofs of multilevel security are needed only for the explicit communication channels connecting processes.

5.6.2 Extensions

Practical systems often have information flow requirements that extend or conflict with the information flow models we have described thus far; for example,

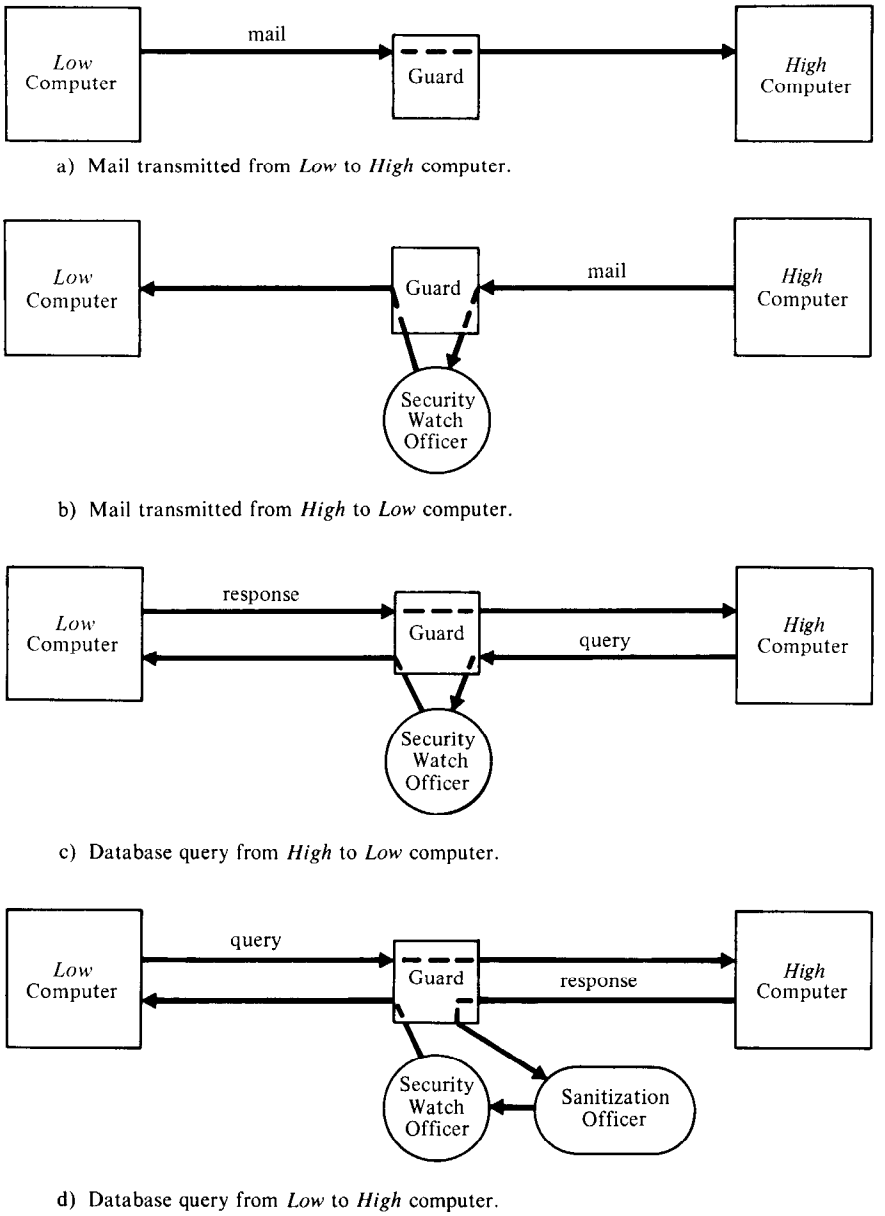
1. **Integrity.** Information flow models describe the dissemination of information, but not its alteration. Thus an *Unclassified* process, for example, can write nonsense into a *Top Secret* file without violating the multilevel security policy. Although this problem is remedied by access controls, efforts to develop a multilevel integrity model have been pursued. Biba [Biba77] proposed a model where each object and process is assigned an integrity level, and a process cannot write into an object unless its integrity level is at least that of the object (in contrast, its security level must be no greater than that of the object).
2. **Sanitization and Downgrading.** Written documents routinely have their security classifications lowered ("downgraded"), and sensitive information is edited ("sanitized") for release at a lower level. Because these operations violate the multilevel security policy, they cannot be handled by the basic flow control mechanisms in a security kernel. The current practice is to place them in trusted processes that are permitted to violate the policies of the kernel. The trusted processes are verified (or at least audited) to show they meet the security requirements of the system as a whole. An example of such a trusted process is described in the next section.
3. **Aggregation.** An aggregate of data might require a higher classification than the individual items. This is because the aggregate might give an overall picture that cannot be deduced (or easily deduced) from an individual item. For example, the aggregate of all combat unit locations might reveal a plan of attack. This is the opposite of the inference problem discussed in the next chapter, where the problem is to provide a means of releasing statistics that give an overall picture, while protecting the individual values used to compute the statistics.

5.6.3 A Guard Application

The ACCAT Guard [Wood79] is an interesting blend of flow controls and cryptography. Developed by Logicon at the Advanced Command and Control Architectural Testbed (ACCAT), the Guard is a minicomputer interface between two computers (or networks of computers) of different classifications (called *Low* and *High*).

The Guard supports two types of communication: network mail and database transactions. The Guard allows information to flow without human intervention from *Low* to *High* computers, but approval by a Security Watch Officer is required for flows from *High* to *Low*. Figure 5.16 shows the role of the Security Watch Officer for both types of communication. For mail, information flows in one direction only, so human intervention is required only for mail transmitted from *High* to *Low* computers (Figure 5.16b). Database transactions always require human intervention, however, because information flows in one direction with the query and the reverse direction with the response. Queries from a *High* computer to a *Low* computer must pass through the Security Watch Officer,

FIGURE 5.16 Information flow through the Guard and Security Watch Officer.



a) Mail transmitted from *Low* to *High* computer.

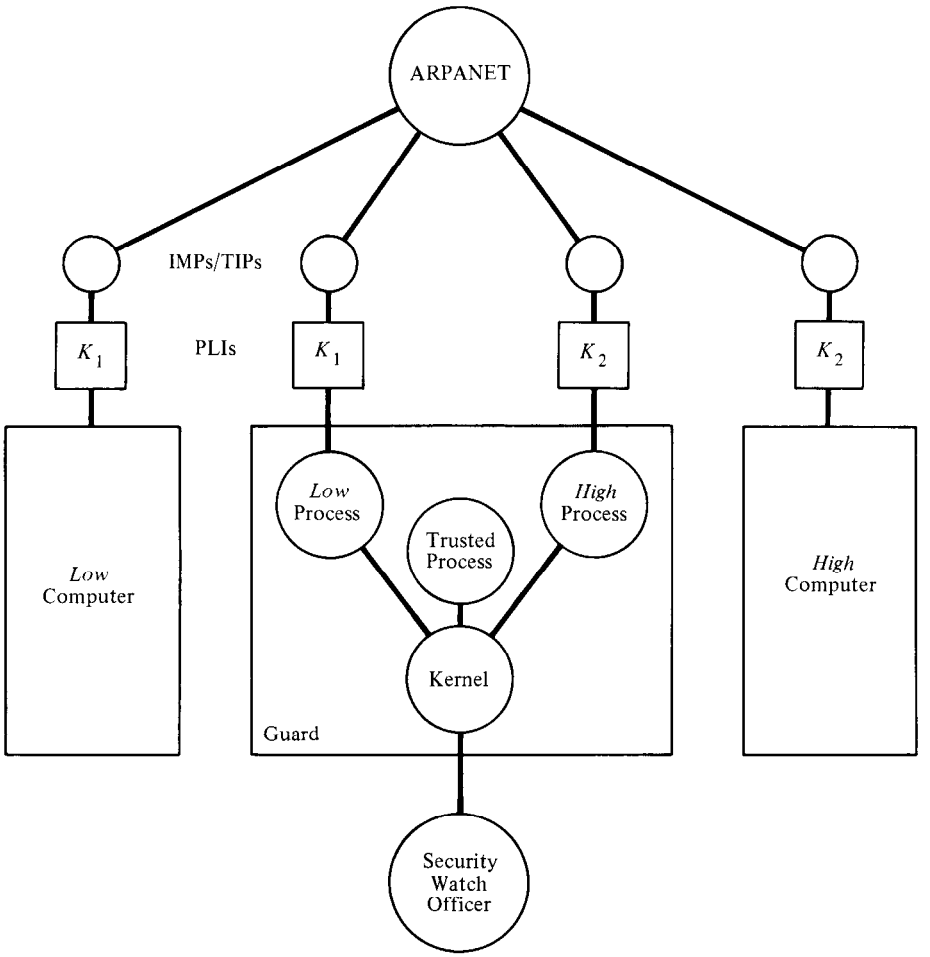
b) Mail transmitted from *High* to *Low* computer.

c) Database query from *High* to *Low* computer.

d) Database query from *Low* to *High* computer.

though the response can be returned without human intervention (Figure 5.16c). Queries from a *Low* to *High* computer can pass through the Guard without human intervention, but here the response must be reviewed. The Guard passes the response first through a Sanitization Officer, who edits the response as needed to

FIGURE 5.17 The ACCAT Guard.



remove *High* security information, and then through the Security Watch Officer (Figure 5.16d).

Figure 5.17 shows the structure of the Guard system. The computers are connected to encryption devices called Private Line Interfaces (PLIs). The PLIs in turn are connected to the ARPANET (Advance Research Projects Agency NETwork) through Interface Message Processors (IMPs) or Terminal Interface Processors (TIPs). Separate encryption keys logically connect the *Low* computers to a *Low* process in the Guard and the *High* computers to a *High* process in the Guard. Because the keys are different, the *Low* and *High* computers cannot communicate directly.

The *Low* and *High* processes run in a secure UNIX environment (such as provided by KSOS). Flows from *Low* to *High* processes in the Guard are handled

by the security mechanisms provided by the kernel. Because flows from *High* to *Low* processes violate the multilevel security policy, they are controlled by a trusted process that interfaces with the Security Watch Officer and downgrades information. Verification of the downgrade trusted process requires showing that all flows from *High* to *Low* are approved by the Security Watch Officer (see [Ames80]). The *Low* and *High* processes communicate with the trusted processes through inter-process communication messages provided by the kernel. The Security Watch Officer's terminal is directly connected to the kernel to prevent spoofing.

The multilevel security problem can be solved in part by running each classification level on a separate machine, and using encryption to enforce the flow requirements between machines [Davi80]. But this is only a partial solution. As illustrated by the Guard, some applications must process multiple classification levels and even violate the general flow policy. These applications must be supported by a secure system so they cannot be misused. The design and verification of specialized systems (such as the Guard), however, should be simpler than for general-purpose systems.

EXERCISES

5.1 Consider the following statement

if $x > k$ **then** $y := 1$,

where x is an integer variable in the range $[1, 2m]$, with all values equally likely, y is initially 0, and k is an integer constant. Give a formula for $H_{y'}(x)$, the equivocation of x given the value of y after the statement is executed. Show that the amount of information transferred by this statement is maximal when $k = m$ by showing this problem is equivalent to showing that the entropy of a variable with two possible values is maximal when both values are equally likely (see Exercise 1.4).

5.2 Consider the statement

if $x > k$ **then** $y := 1$,

where x has the probability distribution

$$p_i = \begin{cases} \frac{1}{2} & x = 0 \\ \frac{1}{4} & x = 1 \\ \frac{1}{4} & x = 2 , \end{cases}$$

and y is initially 0. Compute the entropy $H(x)$. Compute the equivocation $H_{y'}(x)$ both for $k = 0$ and $k = 1$.

5.3 Consider the statement

if $(x = 1)$ and $(y = 1)$ **then** $z := 1$,

where x and y can each be 0 or 1, with both values equally likely, and z is initially 0. Compute the equivocations $H_{z'}(x)$ and $H_{z'}(y)$.

5.4 Consider the statement

$$z := x + y ,$$

where x and y can each be 0 or 1, with both values equally likely. Compute the equivocations $H_{z'}(x)$ and $H_{z'}(y)$.

5.5 Let x be an integer variable in the range $[0, 2^{32} - 1]$, with all values equally likely. Write a program that transfers x to y using implicit flows. Compare the running time of your program with the trivial program "$y := x$".

5.6 Consider the lattice in Figure 5.1. What class corresponds to each of the following?

a. $A \oplus B$, $A \otimes B$
b. $B \oplus I$, $B \otimes I$
c. $B \oplus C$, $B \otimes C$
d. $A \oplus C \oplus D$, $A \otimes C \otimes D$
e. $A \oplus B \oplus D$, $A \otimes B \otimes D$.

5.7 Trace the execution of the procedure *copy1* on the Data Mark Machine (see Figure 5.7 and Table 5.1) for $x = 1$ when $\underline{z} \preceq \underline{y}$. Show that execution of *copy1* is secure when $\underline{x} \prec \underline{y}$ both for $x = 0$ and $x = 1$. Note that if $x \not\preceq y$, then either $x \not\preceq z$ or $z \not\preceq y$.

5.8 Trace the execution of the procedure *copy1* on the single accumulator machine (see Figure 5.8 and Table 5.2) for both $x = 0$ and $x = 1$ when $\underline{x} = High$, $\underline{y} = Low$, $\underline{z} = High$, and $\underline{pc}$ is initially *Low*, showing that execution of this procedure is secure.

5.9 Draw a syntax tree showing how the certification mechanism of Section 5.4.3 verifies the flows in the following statement:

```
while a > 0 do
  begin
    a := a - x;
    b := a * y
  end .
```

5.10 Following the approach in Section 5.4.2, give security conditions for a **case** statement:

```
case a of
  v1: S1;
  v2: S2;
       .
       .
       .
  vn: Sn;
end ,
```

where a is a variable and $v_1, \ldots, v_n$ are values.

5.11 For each Boolean expression in the *copy2* procedure shown in Figures 5.11 and 5.12, identify the set of blocks directly conditioned on the expression.

Assume the value of x can be any integer (i.e., is not restricted to the values 0 and 1). Use your results to identify all implicit flows in the procedure, showing that the procedure is secure.

5.12 Extend the certification semantics of Table 5.6 to include the following statements for concurrent programs:

signal(s) ;
wait(s) ;
cobegin S_1; . . . ; S_n **end** .

Your semantics should provide an efficient method of verifying the relations $\underline{s} \leq \underline{y}$ for every object y that is the target of an assignment in a statement logically following an operation **wait**(s). Note that you will have to extend the semantics of other statements to do this. In particular, you will have to extend the semantics of the **while** statement to certify global flows from the loop condition. Show how the extended mechanism certifies the procedures *copy3* (Figure 5.13) and *copy4* (Figure 5.14).

5.13 Given the statement

```
if x = 0
  then begin
         t := a;
         y := t
       end ,
```

prove the precondition $\{\underline{x} \leq Low, \underline{pc} \leq Low\}$ implies the postcondition $\{\underline{x} \leq Low, (x = 0) \supset (\underline{y} \leq \underline{a}), \underline{pc} \leq Low\}$.

5.14 Prove the precondition implies the postcondition in the following procedure:

```
procedure mod(x, n: integer; var y: integer); var i: integer;
 {n > 0, pc ≤ PC_m}
  begin "compute y = x mod n"
    i := x div n;
    y := x − i*n;
    if y < 0 then y := y + n
  end .
 {0 ≤ y < n, (y − x) mod n = 0, y ≤ x ⊕ n ⊕ PC_m, pc ≤ PC_m}
end mod .
```

5.15 Using the procedure *mod* in the preceding exercise, prove the following:

$\{a = 12, n = 5, \underline{pc} \leq Low, \underline{n} \leq Low\}$
mod(a, n, b)
$\{b = 2, \underline{b} \leq \underline{a}, \underline{pc} \leq Low, \underline{n} \leq Low\}$.

5.16 Give a proof rule for an array assignment of the form

$a[i_1, \ldots, i_n] := b$.

Prove the following:

```
{n ≤ Low, pc ≤ Low}
begin
  i := 1;
  while i ≤ n do
    begin
      a[i] := 0;
      i := i + 1
    end
end
{∀j: (1 ≤ j ≤ n) ⊃ a[j] ≤ Low, n ≤ Low, pc ≤ Low} .
```

5.17 Consider a call *getmsg*(c, p) to the *getmsg* procedure of Figure 5.15, where $a = 2$, $\underline{a} = Low$, $PC = 2$, $\underline{c} = 2$, and $\underline{p} = 1$. Show it is not possible to prove execution of the call is secure.

5.18 Develop axioms and proof rules for the following statements for concurrent programs:

signals(s);
wait(s);
cobegin $S_1; \ldots; S_n$ **end** .

You will have to introduce a class *global* to verify global flows and flows caused by synchronization. The class *global* should increase by both $\underline{s}$ and $\underline{pc}$ after execution of **wait**(s) [note that to do this, you must make your syntactic substitution in the precondition of **wait**(s)]. You should assume the flow proofs

$$\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\}\ S_n\ \{Q_n\}$$

are "interference free" in the sense that executing some S_i does not invalidate the proof of S_j $(1 \leq j \leq n)$. Unlike $\underline{pc}$, the class *global* is never restored. Show how *global* can be included in the proof rules for procedure calls. Consider the procedure *copy3* of Figure 5.13. Prove that the postcondition $\{\underline{y} \leq X, \underline{pc} \leq PC\}$ follows from the precondition $\{\underline{x} \leq X, \underline{pc} \leq PC\}$ and the body of the procedure. Do the same for the procedure *copy4* of Figure 5.14.

5.19 Following Millen's and Furtek's deductive formulation of information flow as described in Section 5.6.1, give constraints for the following specification statements, where x, y, and z are variables:

a. $'y = x + 1$.
b. $'z = x + y$.
c. **if** $x = 1$ **then** $'y = a$ **else** $'y = b$, for variables a and b.
d. **if** $x < 2$ **then** $'b[x] = a[x]$, for arrays $a[1{:}n]$ and $b[1{:}n]$.

REFERENCES

Alle70. Allen, F. E., "Control Flow Analysis," *Proc. Symp. Compiler Optimization, ACM SIGPLAN Notices* Vol. 5(7) pp. 1–19 (July 1970).

Ames80. Ames, S. R. and Keeton-Williams, J. G., "Demonstrating Security for Trusted Applications on a Security Kernel Base," pp. 145–156 in *Proc. 1980 Symp. on Security and Privacy*, IEEE Computer Society (April 1980).

Ande72. Anderson, J. P., "Computer Security Technology Planning Study," ESD-TR-73-51, Vols. I and II, USAF Electronic Systems Div., Bedford, Mass. (Oct. 1972).

Andr80. Andrews, G. R. and Reitman, R. P., "An Axiomatic Approach to Information Flow in Parallel Programs," *ACM Trans. on Prog. Languages and Systems* Vol. 2(1) pp. 56–76 (Jan. 1980).

Bell73. Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations and Model," M74-244, The MITRE Corp., Bedford, Mass. (May 1973).

Bell74. Bell, D. E. and Burke, E. L., "A Software Validation Technique for Certification: The Methodology," ESD-TR-75-54, Vol. I, The MITRE Corp., Bedford, Mass. (Nov. 1974).

Biba77. Biba, K. J., "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Mass. (Apr. 1977).

Birk67. Birkhoff, G., *Lattice Theory*, Amer. Math. Soc. Col. Pub., XXV, 3rd ed. (1967).

Cohe77. Cohen, E., "Information Transmission in Computational Systems," *Proc. 6th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 11(5) pp. 133–139 (Nov. 1977).

Cohe78. Cohen, E., "Information Transmission in Sequential Programs," pp. 297–335 in *Foundations of Secure Computation*, ed. R. A. DeMillo et al., Academic Press, New York (1978).

Davi80. Davida, G. I., DeMillo, R. A., and Lipton, R. J., "A System Architecture to Support a Verifiably Secure Multilevel Security System," pp. 137–145 in *Proc. 1980 Symp. on Security and Privacy*, IEEE Computer Society (Apr. 1980).

Denn74. Denning, D. E., Denning, P. J., and Graham, G. S., "Selectively Confined Subsystems," in *Proc. Int. Workshop on Protection in Operating Systems*, IRIA, Rocquencourt, LeChesnay, France, pp. 55–61 (Aug. 1974).

Denn75. Denning, D. E., "Secure Information Flow in Computer Systems," Ph.D. Thesis, Purdue Univ., W. Lafayette, Ind. (May 1975).

Denn76a. Denning, D. E., "A Lattice Model of Secure Information Flow," *Comm. ACM* Vol. 19(5) pp. 236–243 (May 1976).

Denn76b. Denning, D. E., "On the Derivation of Lattice Structured Information Flow Policies," CSD TR 180, Computer Sciences Dept., Purdue Univ., W. Lafayette, Ind. (Mar. 1976).

Denn77. Denning, D. E. and Denning, P. J., "Certification of Programs for Secure Information Flow," *Comm. ACM* Vol. 20(7) pp. 504–513 (July 1977).

Feie77. Feiertag, R. J., Levitt, K. N., and Robinson, L., "Proving Multilevel Security of a System Design," *Proc. 6th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 11(5) pp. 57–66 (Nov. 1977).

Feie80. Feiertag, R. J., "A Technique for Proving Specifications are Multilevel Secure," Computer Science Lab. Report CSL-109, SRI International, Menlo Park, Calif. (Jan. 1980).

Fent73. Fenton, J. S., "Information Protection Systems," Ph.D. Dissertation, Univ. of Cambridge, Cambridge, England (1973).

Fent74. Fenton, J. S., "Memoryless Subsystems," *Comput. J.* Vol. 17(2) pp. 143–147 (May 1974).

Furt78. Furtek, F., "Constraints and Compromise," pp. 189–204 in *Foundations of Secure Computation*, ed. R. A. DeMillo et al., Academic Press, New York (1978).

Gain72. Gaines, R. S., "An Operating System Based on the Concept of a Supervisory Computer," *Comm. ACM* Vol. 15(3) pp. 150–156 (Mar. 1972).

Gat75. Gat, I. and Saal, H. J., "Memoryless Execution: A Programmer's Viewpoint," IBM Tech. Rep. 025, IBM Israeli Scientific Center, Haifa, Israel (Mar. 1975).

Grie80. Gries, D. and Levin, G., "Assignment and Procedure Call Proof Rules," *ACM Trans. on Programming Languages and Systems* Vol. 2(4) pp. 564–579 (Oct. 1980).

Hoar69. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Comm. ACM* Vol. 12(10) pp. 576–581 (Oct. 1969).

Jone75. Jones, A. K. and Lipton, R. J., "The Enforcement of Security Policies for Computation," *Proc. 5th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 9(5), pp. 197–206 (Nov. 1975).

Karg78. Karger, P. A., "The Lattice Model in a Public Computing Network," *Proc. ACM Annual Conf.* Vol. 1 pp. 453–459 (Dec. 1978).

Lamp73. Lampson, B. W., "A Note on the Confinement Problem," *Comm. ACM* Vol. 16(10) pp. 613–615 (Oct. 1973).

Lipn75. Lipner, S. B., "A Comment on the Confinement Problem," *Proc. 5th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol 9(5) pp. 192–196 (Nov. 1975).

Lond78. London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., and Popek, G. J., "Proof Rules for the Programming Language Euclid," *Acta Informatica* Vol. 10 pp. 1–26 (1978).

Mill76. Millen, J. K., "Security Kernel Validation in Practice," *Comm. ACM* Vol. 19(5) pp. 243–250 (May 1976).

Mill78. Millen, J. K., "Constraints and Multilevel Security," pp. 205–222 in *Foundations of Secure Computation*, ed. R. A. DeMillo et al., Academic Press, New York (1978).

Mill81. Millen, J. K., "Information Flow Analysis of Formal Specifications," in *Proc. 1981 Symp. on Security and Privacy*, IEEE Computer Society, pp. 3–8 (Apr. 1981).

Mins67. Minsky, M., *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J. (1967).

Neum80. Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L., "A Provably Secure Operating System: The System, Its Applications, and Proofs," Computer Science Lab. Report CSL-116, SRI International, Menlo Park, Calif. (May 1980).

Reed79. Reed, D. P. and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," *Comm. ACM* Vol. 22(2) pp. 115–123 (Feb. 1979).

Reit79. Reitman, R. P., "A Mechanism for Information Control in Parallel Programs," *Proc. 7th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.*, pp. 55–62 (Dec. 1979).

Rote74. Rotenberg, L. J., "Making Computers Keep Secrets," Ph.D. Thesis, TR-115, MIT (Feb. 1974).

Rush81. Rushby, J. M., "Design and Verification of Secure Systems," *Proc. 8th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.*, Vol. 15(5), pp. 12–21 (Dec. 1981).

Sche72. Schell, R., Downey, P., and Popek, G., "Preliminary Notes on the Design of a Secure Military Computer System," MCI-73-1, USAF Electronic Systems Div., Bedford, Mass. (Oct. 1972).

Weis69. Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," pp. 119–133 in *Proc. Fall Jt. Computer Conf.*, Vol. 35, AFIPS Press, Montvale, N.J. (1969).

Wood79. Woodward, J. P. L., "Applications for Multilevel Secure Operating Systems," pp. 319–328 in *Proc. NCC*, Vol. 48. AFIPS Press, Montvale, N.J. (1979).