Chapter 15
Information Flow

BOTTOM: Masters, I am to discourse wonders: but ask me not what; for if I tell you, I am no true Athenian. I will tell you every thing, right as it fell out.
—*A Midsummer Night's Dream*, IV, ii, 30–33.

Although access controls can constrain the rights of a user, they cannot constrain the flow of information about a system. In particular, when a system has a security policy regulating information flow, the system must ensure that the information flows do not violate the constraints of the policy. Both compile-time mechanisms and runtime mechanisms support the checking of information flows. Several systems implementing these mechanisms demonstrate their effectiveness.

15.1 Basics and Background

Information flow policies define the way information moves throughout a system. Typically, these policies are designed to preserve confidentiality of data or integrity of data. In the former, the policy's goal is to prevent information from flowing to a user not authorized to receive it. In the latter, information may flow only to processes that are no more trustworthy than the data.

Any confidentiality and integrity policy embodies an information flow policy.

EXAMPLE: The Bell-LaPadula Model describes a lattice-based information flow policy. Given two compartments A and B, information can flow from an object in A to a subject in B if and only if B dominates A.

Let x be a variable in a program. The notation $\underline{x}$ refers to the information flow class of x.

EXAMPLE: Consider a system that uses the Bell-LaPadula Model. The variable *x*, which holds data in the compartment (TS, { NUC, EUR }), is set to 3. Then *x* = 3 and $\underline{x}$ = (TS, { NUC, EUR }).

Intuitively, information flows from an object *x* to an object *y* if the application of a sequence of commands *c* causes the information initially in *x* to affect the information in *y*.

> **Definition 15–1.** The command sequence *c* causes a *flow of information from x to y* if, after execution of *c*, some information about the value of *x* before *c* was executed can be deduced from the value of *y* after *c* was executed.

This definition views information flow in terms of the information that the value of *y* allows one to deduce about the value in *x*. For example, the statement

```
y := x;
```

reveals the value of *x* in the initial state, so information about the value of *x* in the initial state can be deduced from the value of *y* after the statement is executed. The statement

```
y := x / z;
```

reveals some information about *x*, but not as much as the first statement.

The final result of the sequence *c* must reveal information about the initial value of *x* for information to flow. The sequence

```
tmp := x;
y := tmp;
```

has information flowing from *x* to *y* because the (unknown) value of *x* at the beginning of the sequence is revealed when the value of *y* is determined at the end of the sequence. However, no information flow occurs from *tmp* to *x*, because the initial value of *tmp* cannot be determined at the end of the sequence.

EXAMPLE: Consider the statement

```
x := y + z;
```

Let *y* take any of the integer values from 0 to 7, inclusive, with equal probability, and let *z* take the value 1 with probability 0.5 and the values 2 and 3 with probability 0.25 each. Once the resulting value of *x* is known,the initial value of *y* can assume at most three values. Thus, information flows from *y* to *x*. Similar results hold for *z*.

EXAMPLE: Consider a program in which *x* and *y* are integers that may be either 0 or 1. The statement

```
if x = 1 then y := 0;
else y := 1;
```

does not explicitly assign the value of *x* to *y*.

Assume that *x* is equally likely to be 0 or 1. Then $H(x_s) = 1$. But $H(x_s \mid y_t) = 0$, because if *y* is 0, *x* is 1, and vice versa. Hence, $H(x_s \mid y_t) = 0 < H(x_s \mid y_s) = H(x_s) = 1$. Thus, information flows from *x* to *y*.

Definition 15–2. An *implicit flow of information* occurs when information flows from *x* to *y* without an explicit assignment of the form $y := f(x)$, where $f(x)$ is an arithmetic expression with the variable *x*.

The flow of information occurs, not because of an assignment of the value of *x*, but because of a flow of control based on the value of *x*. This demonstrates that analyzing programs for assignments to detect information flows is not enough. To detect all flows of information, implicit flows must be examined.

15.1.1 Information Flow Models and Mechanisms

An information flow policy is a security policy that describes the authorized paths along which that information can flow. Each model associates a label, representing a security class, with information and with entities containing that information. Each model has rules about the conditions under which information can move throughout the system.

In this chapter, we use the notation $\underline{x} \leq \underline{y}$ to mean that information can flow from an element of class *x* to an element of class *y*. Equivalently, this says that information with a label placing it in class $\underline{x}$ can flow into class $\underline{y}$.

Earlier chapters usually assumed that the models of information flow policies were lattices. We first consider nonlattice information flow policies and how their structures affect the analysis of information flow. We then turn to compiler-based information flow mechanisms and runtime mechanisms. We conclude with a look at flow controls in practice.

15.2 Compiler-Based Mechanisms

Compiler-based mechanisms check that information flows throughout a program are authorized. The mechanisms determine if the information flows in a program *could* violate a given information flow policy. This determination is not precise, in that secure paths of information flow may be marked as violating the policy; but it is secure, in that no unauthorized path along which information may flow will be undetected.

Definition 15–3. A set of statements is *certified* with respect to an information flow policy if the information flow within that set of statements does not violate the policy.

EXAMPLE: Consider the program statement

```
if x = 1 then y := a;
else y := b;
```

By the rules discussed earlier, information flows from x and a to y or from x and b to y, so if the policy says that $\underline{a} \le \underline{y}$, $\underline{b} \le \underline{y}$, and $\underline{x} \le \underline{y}$, then the information flow is secure. But if $\underline{a} \le \underline{y}$ only when some other variable $z = 1$, the compiler-based mechanism must determine whether $z = 1$ before certifying the statement. Typically, this is infeasible. Hence, the compiler-based mechanism would not certify the statement. The mechanisms described here follow those developed by Denning and Denning [247] and Denning [242].

15.2.1 Declarations

For our discussion, we assume that the allowed flows are supplied to the checking mechanisms through some external means, such as from a file. The specifications of allowed flows involve security classes of language constructs. The program involves variables, so some language construct must relate variables to security classes. One way is to assign each variable to exactly one security class. We opt for a more liberal approach, in which the language constructs specify the set of classes from which information may flow into the variable. For example,

```
x: integer class { A, B }
```

states that x is an integer variable and that data from security classes A and B may flow into x. Note that the classes are statically, not dynamically, assigned. Viewing the security classes as a lattice, this means that x's class must be at least the least upper bound of classes A and B—that is, $lub\{A, B\} \le \underline{x}$.

Two distinguished classes, *Low* and *High*, represent the greatest lower bound and least upper bound, respectively, of the lattice. All constants are of class *Low.*

Information can be passed into or out of a procedure through parameters. We classify parameters as *input parameters* (through which data is passed into the procedure), *output parameters* (through which data is passed out of the procedure), and *input/output parameters* (through which data is passed into and out of the procedure).

```
(* input parameters are named is; output parameters, os; *)
(* and input/output parameters, ios, with s a subscript *)
proc something(i1, ..., ik; var o1, ..., om, io1, ..., ion);
var l1, ..., lj;                (* local variables *)
begin
        S;                      (* body of procedure *)
end;
```

The class of an input parameter is simply the class of the actual argument:

i_s: *type* **class** { i_s }

Let $r_1, ..., r_p$ be the set of input and input/output variables from which information flows to the output variable o_s. The declaration for the type must capture this:

o_s: *type* **class** { $r_1, ..., r_p$ }

(We implicitly assume that any output-only parameter is initialized in the procedure.) The input/output parameters are like output parameters, except that the initial value (as input) affects the allowed security classes. Again, let $r_1, ..., r_p$ be defined as above. Then:

io_s: *type* **class** { $r_1, ..., r_p, io_1, ..., io_k$ }

EXAMPLE: Consider the following procedure for adding two numbers.

```
proc sum(x: int class { x };
             var out: int class { x, out });
begin
        out := out + x;
end;
```

Here, we require that $x \le out$ and $out \le out$ (the latter holding because $\le$ is reflexive).

The declarations presented so far deal only with basic types, such as integers, characters, floating point numbers, and so forth. Nonscalar types, such as arrays, records (structures), and variant records (unions) also contain information. The rules for information flow classes for these data types are built on the scalar types.

Consider the array

```
a: array 1 .. 100 of int;
```

First, look at information flows out of an element $a[i]$ of the array. In this case, information flows from $a[i]$ and from i, the latter by virtue of the index indicating which element of the array to use. Information flows into $a[i]$ affect only the value in $a[i]$, and so do not affect the information in i. Thus, for information flows from $a[i]$, the class involved is $lub\{ \underline{a[i]}, \underline{i} \}$; for information flows into $a[i]$, the class involved is $\underline{a[i]}$.

15.2.2 Program Statements

A program consists of several types of statements. Typically, they are

1. Assignment statements
2. Compound statements
3. Conditional statements
4. Iterative statements
5. Goto statements
6. Procedure calls
7. Function calls
8. Input/output statements.

We consider each of these types of statements separately, with two exceptions. Function calls can be modeled as procedure calls by treating the return value of the function as an output parameter of the procedure. Input/output statements can be modeled as assignment statements in which the value is assigned to (or assigned from) a file. Hence, we do not consider function calls and input/output statements separately.

15.2.2.1 Assignment Statements

An assignment statement has the form

```
y := f(x1, ..., xn)
```

where y and $x_1, ..., x_n$ are variables and f is some function of those variables. Information flows from each of the x_i's to y. Hence, the requirement for the information flow to be secure is

- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq \underline{y}$

EXAMPLE: Consider the statement

```
x := y + z;
```

Then the requirement for the information flow to be secure is $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$.

15.2.2.2 Compound Statements

A compound statement has the form

```
begin
     S_1;
     ...
     S_n;
end;
```

where each of the S_i's is a statement. If the information flow in each of the statements is secure, then the information flow in the compound statement is secure. Hence, the requirements for the information flow to be secure are

- S_1 secure
- ...
- S_n secure

EXAMPLE: Consider the statements

```
begin
     x := y + z;
     a := b * c - x;
end;
```

Then the requirements for the information flow to be secure are $lub\{\ \underline{y}, \underline{z}\ \} \leq \underline{x}$ for S_1 and $lub\{\ \underline{b}, \underline{c}, \underline{x}\ \} \leq \underline{a}$ for S_2. So, the requirements for secure information flow are $lub\{\ \underline{y}, \underline{z}\ \} \leq \underline{x}$ and $lub\{\ \underline{b}, \underline{c}, \underline{x}\ \} \leq \underline{a}$.

15.2.2.3 Conditional Statements

A conditional statement has the form

```
if f(x_1, ..., x_n) then
     S_1;
else
     S_2;
end;
```

where $x_1, \ldots, x_n$ are variables and f is some (boolean) function of those variables. Either S_1 or S_2 may be executed, depending on the value of f, so both must be secure. As discussed earlier, the selection of either S_1 or S_2 imparts information about the values of the variables $x_1, \ldots, x_n$, so information must be able to flow from those variables to any targets of assignments in S_1 and S_2. This is possible if and only if the *lowest* class of the targets dominates the highest class of the variables $x_1, ..., x_n$. Thus, the requirements for the information flow to be secure are

- S_1 secure
- S_2 secure
- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq glb\{ \underline{y} \mid y$ is the target of an assignment in S_1 and S_2 }

As a degenerate case, if statement S_2 is empty, it is trivially secure and has no assignments.

EXAMPLE: Consider the statements

```
if x + y < z then
     a := b;
else
     d := b * c - x;
end;
```

Then the requirements for the information flow to be secure are $\underline{b} \leq \underline{a}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{d}$ for S_2. But the statement that is executed depends on the values of x, y, and z. Hence, information also flows from x, y, and z to d and a. So, the requirements are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$, $\underline{b} \leq a$, and $lub\{ \underline{x}, \underline{y}, \underline{z} \} \leq glb\{ \underline{a}, \underline{d} \}$.

15.2.2.4 Iterative Statements

An iterative statement has the form

```
while f($x_1$, ..., $x_n$) do
     S;
```

where $x_1, ..., x_n$ are variables and f is some (boolean) function of those variables. Aside from the repetition, this is a conditional statement, so the requirements for information flow to be secure for a conditional statement apply here.

To handle the repetition, first note that the number of repetitions causes information to flow only through assignments to variables in S. The number of repetitions is controlled by the values in the variables $x_1, ..., x_n$, so information flows from those variables to the targets of assignments in S—but this is detected by the requirements for information flow of conditional statements.

However, if the program never leaves the iterative statement, statements after the loop will never be executed. In this case, information has flowed from the variables $x_1, ..., x_n$ by the *absence* of execution. Hence, secure information flow also requires that the loop terminate.

Thus, the requirements for the information flow to be secure are

- Iterative statement terminates
- *S* secure
- $lub\{\underline{x_1}, ..., \underline{x_n}\} \le glb\{\ \underline{y} \mid y \text{ is the target of an assignment in } S\ \}$

EXAMPLE: Consider the statements

```
while i < n do
begin
     a[i] := b[i];
     i := i + 1;
end;
```

This loop terminates. If $n \le i$ initially, the loop is never entered. If $i < n$, i is incremented by a positive integer, 1, and so increases, at each iteration. Hence, after $n - i$ iterations, $n = i$, and the loop terminates.

Now consider the compound statement that makes up the body of the loop. The first statement is secure if $\underline{i} \le \underline{a[i]}$ and $\underline{b[i]} \le \underline{a[i]}$; the second statement is secure because $\underline{i} \le \underline{i}$. Hence, the compound statement is secure if $lub\{\ \underline{i}, \underline{b[i]}\ \} \le \underline{a[i]}$.

Finally, $a[i]$ and i are targets of assignments in the body of the loop. Hence, information flows into them from the variables in the expression in the *while* statement. So, $lub\{\ \underline{i}, \underline{n}\ \} \le glb\{\ \underline{a[i]}, \underline{i}\ \}$. Putting these together, the requirement for the information flow to be secure is $lub\{\ \underline{b[i]}, \underline{i}, \underline{n}\ \} \le glb\{\ \underline{a[i]}, \underline{i}\ \}$ (see Exercise 2).

15.2.2.5 Goto Statements

A goto statement contains no assignments, so no explicit flows of information occur. Implicit flows may occur; analysis detects these flows.

Definition 15–4. A *basic block* is a sequence of statements in a program that has one entry point and one exit point.

EXAMPLE: Consider the following code fragment.

```
proc transmatrix(x: array [1..10][1..10] of int class { x };
                var y: array [1..10][1..10] of int class { y } );
     var i, j: int class { tmp };
     begin
        i := 1;                        (* b1 *)
     l2: if i > 10 goto l7;            (* b2 *)
        j := 1;                        (* b3 *)
     l4: if j > 10 then goto l6;       (* b4 *)
```

```
        y[j][i] := x[i][j];         (* b5 *)
        j := j + 1;
        goto l4;
-------------------------------------------------
    l6: i := i + 1;                 (* b6 *)
        goto l2;
-------------------------------------------------
    l7:                             (* b7 *)
end;
```

There are seven basic blocks, labeled b_1 through b_7 and separated by lines. The second and fourth blocks have two ways to arrive at the entry—either from a jump to the label or from the previous line. They also have two ways to exit—either by the branch or by falling through to the next line. The fifth block has three lines and always ends with a branch. The sixth block has two lines and can be entered either from a jump to the label or from the previous line. The last block is always entered by a jump.

Control within a basic block flows from the first line to the last. Analyzing the flow of control within a program is therefore equivalent to analyzing the flow of control among the program's basic blocks. Figure 15–1 shows the flow of control among the basic blocks of the body of the procedure *transmatrix*.

When a basic block has two exit paths, the block reveals information implicitly by the path along which control flows. When these paths converge later in the program, the (implicit) information flow derived from the exit path from the basic block becomes either explicit (through an assignment) or irrelevant. Hence, the class

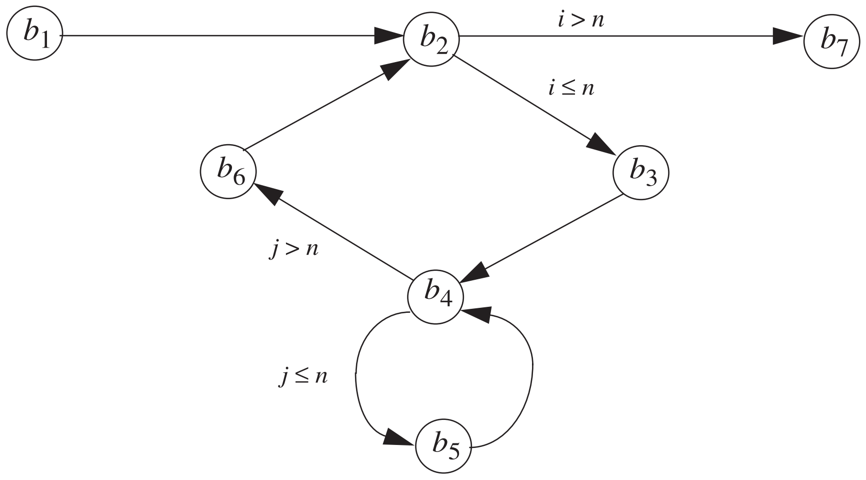

Figure 15–1 The control flow graph of the procedure *transmatrix*. The basic blocks are labeled b_1 through b_7. The conditions under which branches are taken are shown over the edges corresponding to the branches.

of the expression that causes a particular execution path to be selected affects the required classes of the blocks along the path up to the block at which the divergent paths converge.

> **Definition 15–5.** An *immediate forward dominator* of a basic block b (written $IFD(b)$) is the first block that lies on all paths of execution that pass through b.

EXAMPLE: In the procedure *transmatrix*, the immediate forward dominators of each block are $IFD(b_1) = b_2$, $IFD(b_2) = b_7$, $IFD(b_3) = b_4$, $IFD(b_4) = b_6$, $IFD(b_5) = b_4$, and $IFD(b_6) = b_2$.

Computing the information flow requirement for the set of blocks along the path is now simply applying the logic for the conditional statement. Each block along the path is taken because of the value of an expression. Information flows from the variables of the expression into the set of variables assigned in the blocks. Let B_i be the set of blocks along an execution path from b_i to $IFD(b_i)$, but excluding these endpoints. (See Exercise 3.) Let $x_{i1}, ..., x_{in}$ be the set of variables in the expression that selects the execution path containing the blocks in B_i. The requirements for the program's information flows to be secure are

- All statements in each basic block secure
- $lub\{\underline{x}_{i1}, ..., \underline{x}_{in}\} \le glb\{ \underline{y} \mid y$ is the target of an assignment in B_i $\}$

EXAMPLE: Consider the body of the procedure *transmatrix*. We first state requirements for information flow within each basic block:

b_1: $Low \le \underline{i} \Rightarrow$ secure

b_3: $Low \le \underline{j} \Rightarrow$ secure

b_5: $lub\{ \underline{x[i][j]}, \underline{i}, \underline{j} \} \le \underline{y[j][i]}$; $\underline{j} \le \underline{j} \Rightarrow lub\{ \underline{x[i][j]}, \underline{i}, \underline{j} \} \le \underline{y[j][i]}$

b_6: $lub\{ Low, \underline{i} \} \le \underline{i} \Rightarrow$ secure

The requirement for the statements in each basic block to be secure is, for $i = 1, ..., n$ and $j = 1, ..., n$, $lub\{ \underline{x[i][j]}, \underline{i}, \underline{j} \} \le \underline{y[j][i]}$. By the declarations, this is true when $lub\{\underline{x}, \underline{i}\} \le \underline{y}$.

In this procedure, $B_2 = \{ b_3, b_4, b_5, b_6 \}$ and $B_4 = \{ b_5 \}$. Thus, in B_2, statements assign values to i, j, and $y[j][i]$. In B_4, statements assign values to j and $y[j][i]$. The expression controlling which basic blocks in B_2 are executed is $i \le 10$; the expression controlling which basic blocks in B_4 are executed is $j \le 10$. Secure information flow requires that $\underline{i} \le glb\{ \underline{i}, \underline{j}, \underline{y[j][i]}\}$ and $\underline{j} \le glb\{ \underline{j}, \underline{y[j][i]} \}$. In other words, $\underline{i} \le glb\{ \underline{i}, \underline{y} \}$ and $\underline{i} \le glb\{ \underline{i}, \underline{y} \}$, or $\underline{i} \le \underline{y}$.

Combining these requirements, the requirement for the body of the procedure to be secure with respect to information flow is $lub\{\underline{x}, \underline{i}\} \le \underline{y}$.

15.2.2.6 Procedure Calls

A procedure call has the form

```
proc procname(i₁, ..., iₘ : int; var o₁, ..., oₙ : int);
begin
     S;
end;
```

where each of the i_j's is an input parameter and each of the o_j's is an input/output parameter. The information flow in the body S must be secure. As discussed earlier, information flow relationships may also exist between the input parameters and the output parameters. If so, these relationships are necessary for S to be secure. The actual parameters (those variables supplied in the call to the procedure) must also satisfy these relationships for the call to be secure. Let $x_1, ..., x_m$ and $y_1, ..., y_n$ be the actual input and input/output parameters, respectively. The requirements for the information flow to be secure are

- S secure
- For $j = 1, ..., m$ and $k = 1, ..., n$, if $\underline{i_j} \le \underline{o_k}$ then $\underline{x_j} \le \underline{y_k}$
- For $j = 1, ..., n$ and $k = 1, ..., n$, if $\underline{o_j} \le \underline{o_k}$ then $\underline{y_j} \le \underline{y_k}$

EXAMPLE: Consider the procedure *transmatrix* from the preceding section. As we showed there, the body of the procedure is secure with respect to information flow when $lub\{\underline{x}, \underline{tmp}\} \le \underline{y}$. This indicates that the formal parameters x and y have the information flow relationship $\underline{x} \le \underline{y}$. Now, suppose a program contains the call

```
transmatrix(a, b)
```

The second condition asserts that this call is secure with respect to information flow if and only if $\underline{a} \le \underline{b}$.

15.2.3 Exceptions and Infinite Loops

Exceptions can cause information to flow.

EXAMPLE: Consider the following procedure, which copies the (approximate) value of x to y.[1]

```
proc copy(x: int class { x }; var y: int class Low);
var  sum: int class { x };
     z: int class Low;
```

[1] From Denning [242], p. 306.

```
begin
     z := 0;
     sum := 0;
     y := 0;
     while z = 0 do begin
           sum := sum + x;
           y := y + 1;
     end
end
```

When *sum* overflows, a trap occurs. If the trap is not handled, the procedure exits. The value of *x* is *MAXINT* / *y*, where *MAXINT* is the largest integer representable as an *int* on the system. At no point, however, is the flow relationship $\underline{x} \le \underline{y}$ checked.

If exceptions are handled explicitly, the compiler can detect problems such as this. Denning again supplies such a solution.

EXAMPLE: Suppose the system ignores all exceptions unless the programmer specifically handles them. Ignoring the exception in the preceding example would cause the program to loop indefinitely. So, the programmer would want the loop to terminate when the exception occurred. The following line does this.

```
on overflowexception sum do z := 1;
```

This line causes information to flow from *sum* to *z*, meaning that $\underline{sum} \le \underline{z}$. Because $\underline{z}$ is *Low* and $\underline{sum}$ is { *x* }, this is incorrect and the procedure is not secure with respect to information flow.

Denning also notes that infinite loops can cause information to flow in unexpected ways.

EXAMPLE: The following procedure copies data from *x* to *y*. It assumes that *x* and *y* are either 0 or 1.

```
proc copy(x: int 0..1 class { x };
            var y: int 0..1 class Low);
begin
     y := 0;
     while x = 0 do
          (* nothing *);
     y := 1;
end.
```

If *x* is 0 initially, the procedure does not terminate. If *x* is 1, it does terminate, with *y* being 1. At no time is there an explicit flow from *x* to *y*. This is an example of a *covert channel*, which we will discuss in detail in the next chapter.

15.2.4 Concurrency

Of the many concurrency control mechanisms that are available, we choose to study information flow using semaphores [270]. Their operation is simple, and they can be used to express many higher-level constructs [135, 718]. The specific semaphore constructs are

```
wait(x): if x = 0 then block until x > 0; x := x - 1;
signal(x): x := x + 1;
```

where x is a semaphore. As usual, the *wait* and the *signal* are indivisible; once either one has started, no other instruction will execute until the *wait* or *signal* finishes.

Reitman and his colleagues [33, 748] point out that concurrent mechanisms add information flows when values common to multiple processes cause specific actions. For example, in the block

```
begin
     wait(sem);
     x := x + 1;
end;
```

the program blocks at the *wait* if *sem* is 0, and executes the next statement when *sem* is nonzero. The earlier certification requirement for compound statements is not sufficient because of the implied flow between *sem* and x. The certification requirements must take flows among local and shared variables (semaphores) into account.

Let the block be

```
begin
     $S_1$;
     ...
     $S_n$;
end;
```

Assume that each of the statements $S_1, ..., S_n$ is certified. Semaphores in the *signal* do not affect information flow in the program in which the *signal* occurs, because the *signal* statement does not block. But following a *wait* statement, which may block, information implicitly flows from the semaphore in the *wait* to the targets of successive assignments.

Let statement S_i be a *wait* statement, and let $shared(S_i)$ be the set of shared variables that are read (so information flows from them). Let $\underline{g(S_i)}$ be the greatest lower bound of the targets of assignments following S_i. A requirement that the block be secure is that $\underline{shared(S_i)} \leq \underline{g(S_i)}$. Thus, the requirements for certification of a compound statement with concurrent constructs are

- S_1 secure
- ...
- S_n secure
- For $i = 1, ..., n$ [$\underline{shared(S_i)} \leq \underline{g(S_i)}$]

EXAMPLE: Consider the statements

```
begin
     x := y + z;
     wait(sem);
     a := b * c - x;
end;
```

The requirements that the information flow be secure are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$ for S_2. Information flows implicitly from *sem* to *a*, so $\underline{sem} \leq \underline{a}$. The requirements for certification are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$, $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$, and $\underline{sem} \leq \underline{a}$.

Loops are handled similarly. The only difference is in the last requirement, because after completion of one iteration of the loop, control may return to the beginning of the loop. Hence, a semaphore may affect assignments that precede the *wait* statement in which the semaphore is used. This simplifies the last condition in the compound statement requirement considerably. Information must be able to flow from all shared variables named in the loop to the targets of all assignments. Let $shared(S_i)$ be the set of shared variables read, and let $t_1, ..., t_m$ be the targets of assignments in the loop. Then the certification conditions for the iterative statement

```
while f(x₁, ..., xₙ) do
     S;
```

are

- Iterative statement terminates
- S secure
- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq glb\{ t_1, ..., \underline{t}_m \}$
- $lub\{\underline{shared(S_1)}, ,,,, \underline{shared(S_n)} \} \leq glb\{ \underline{t}_1, ..., \underline{t}_m \}$

EXAMPLE: Consider the statements

```
while i < n do
begin
     a[i] := item;
     wait(sem);
     i := i + 1;
end;
```

This loop terminates. If $n \le i$ initially, the loop is never entered. If $i < n$, i is incremented by a positive integer, 1, and so increases, at each iteration. Hence, after $n - i$ iterations, $n = i$, and the loop terminates.

Now consider the compound statement that makes up the body of the loop. The first statement is secure if $\underline{i} \le \underline{a[i]}$ and $\underline{item} \le \underline{a[i]}$.The third statement is secure because $\underline{i} \le \underline{i}$. The second statement induces an implicit flow, so $\underline{sem} \le \underline{a[i]}$ and $\underline{sem} \le \underline{i}$. The requirements are thus $\underline{i} \le \underline{a[i]}$, $\underline{item} \le \underline{a[i]}$, $\underline{sem} \le \underline{a[i]}$, and $\underline{sem} \le \underline{i}$.

Finally, concurrent statements have no information flow among them per se. Any such flows occur because of semaphores and involve compound statements (discussed above). The certification conditions for the concurrent statement

```
cobegin
     S₁;
     ...
     Sₙ;
coend;
```

are

- S_1 secure
- ...
- S_n secure

EXAMPLE: Consider the statements

```
cobegin
     x := y + z;
     a := b * c - y;
coend;
```

The requirements that the information flow be secure are $lub\{ \underline{y}, \underline{z} \} \le \underline{x}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{y} \} \le \underline{a}$ for S_2. The requirement for certification is simply that both of these requirements hold.

15.2.5 Soundness

Denning and Denning [247], Andrews and Reitman [33], and others build their argument for security on the intuition that combining secure information flows produces a secure information flow, for some security policy. However, they never formally prove this intuition. Volpano, Irvine, and Smith [920] express the semantics of the above-mentioned information on flow analysis as a set of types, and equate certification that a certain flow can occur to the correct use of types. In this context, checking for valid information flows is equivalent to checking that variable and expression types conform to the semantics imposed by the security policy.

Let x and y be two variables in the program. Let x's label dominate y's label. A set of information flow rules is sound if the value in x cannot affect the value in y during the execution of the program. Volpano, Irvine, and Smith use language-based techniques to prove that, given a type system equivalent to the certification rules discussed above, all programs without type errors have the noninterference property described above. Hence, the information flow certification rules of Denning and of Andrews and Reitman are sound.

15.3 Execution-Based Mechanisms

The goal of an execution-based mechanism is to prevent an information flow that violates policy. Checking the flow requirements of explicit flows achieves this result for statements involving explicit flows. Before the assignment

```
y = f(x1, ..., xn)
```

is executed, the execution-based mechanism verifies that

$$lub(\underline{x}_1, ..., \underline{x}_n) \leq \underline{y}$$

If the condition is true, the assignment proceeds. If not, it fails. A naïve approach, then, is to check information flow conditions whenever an explicit flow occurs.

Implicit flows complicate checking.

EXAMPLE: Let x and y be variables. The requirement for certification for a particular statement y *op* x is that $\underline{x} \leq \underline{y}$. The conditional statement

```
if x = 1 then y := a;
```

causes a flow from x to y. Now, suppose that when $x \neq 1$, $\underline{x} = High$ and $\underline{y} = Low$. If flows were verified only when explicit, and $x \neq 1$, the implicit flow would not be checked. The statement may be incorrectly certified as complying with the information flow policy.

Fenton explored this problem using a special abstract machine.

15.3.1 Fenton's Data Mark Machine

Fenton [313] created an abstract machine called the *Data Mark Machine* to study handling of implicit flows at execution time. Each variable in this machine had an associated security class, or tag. Fenton also included a tag for the program counter (PC).

The inclusion of the PC allowed Fenton to treat implicit flows as explicit flows, because branches are merely assignments to the PC. He defined the semantics of the Data Mark Machine. In the following discussion, *skip* means that the instruction is not executed, *push*(x, $\underline{x}$) means to push the variable x and its security class $\underline{x}$ onto the program stack, and *pop*(x, $\underline{x}$) means to pop the top value and security class off the program stack and assign them to x and $\underline{x}$, respectively.

Fenton defined five instructions. The relationships between execution of the instructions and the classes of the variables are as follows.

1. The increment instruction

   ```
   x := x + 1
   ```

   is equivalent to

   $$\texttt{if } \underline{\texttt{PC}} \le \underline{\texttt{x}} \texttt{ then x := x + 1; else skip}$$

2. The conditional instruction

   ```
   if x = 0 then goto n else x := x – 1
   ```

   is equivalent to

   $$\begin{array}{ll}\texttt{if x = 0 then} & \texttt{\{ push(PC, }\underline{\texttt{PC}}\texttt{); }\underline{\texttt{PC}}\texttt{ = lub(}\underline{\texttt{PC}}\texttt{, }\underline{\texttt{x}}\texttt{); PC := n; \}} \\ \texttt{else} & \texttt{\{ if }\underline{\texttt{PC}} \le \underline{\texttt{x}}\texttt{ then \{ x := x – 1; \} else skip \}}\end{array}$$

   This branches, and pushes the PC and its security class onto the program stack. (As is customary, the PC is incremented so that when it is popped, the instruction following the *if* statement is executed.) This captures the PC containing information from x (specifically, that x is 0) while following the **goto**.
3. The return

   ```
   return
   ```

   is equivalent to

   $$\texttt{pop(PC, }\underline{\texttt{PC}}\texttt{);}$$

This returns control to the statement following the last *if* statement. Because the flow of control would have arrived at this statement, the PC no longer contains information about *x*, and the old class can be restored.

4. The branch instruction

```
if’ x = 0 then goto n else x := x – 1
```

is equivalent to

```
if x = 0 then { if x ≤ PC then { PC := n; } else skip }
else          { if PC ≤ x then { x := x – 1; } else skip }
```

This branches without saving the PC on the stack. If the branch occurs, the PC is in a higher security class than the conditional variable *x*, so adding information from *x* to the PC does not change the PC’s security class.

5. The halt instruction

```
halt
```

is equivalent to

```
if program stack empty then halt execution
```

The program stack being empty ensures that the user cannot obtain information by looking at the program stack after the program has halted (for example, to determine which *if* statement was last taken).

EXAMPLE: Consider the following program, in which *x* initially contains 0 or 1.[2]

```
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
```

This program copies the value of *x* to *y*. Suppose that *x* = 1 initially. The following table shows the contents of memory, the security class of the PC at each step, and the corresponding certification check.

[2] From Denning [242], Figure 5.7, p. 290.

x	y	z	PC	$\underline{\text{PC}}$	*stack*	*certification check*
1	0	0	1	*Low*	—	
0	0	0	2	*Low*	—	$Low \le \underline{x}$
0	0	0	6	$\underline{x}$	(3, *Low*)	
0	1	0	7	$\underline{x}$	(3, *Low*)	$\underline{\text{PC}} \le \underline{y}$
0	1	0	3	*Low*	—	

Fenton's machine handles errors by ignoring them. Suppose that, in the program above, $\underline{y} \le \underline{x}$. Then at the fifth step, the certification check fails (because $\underline{\text{PC}} = \underline{x}$). So, the assignment is skipped, and at the end $y = 0$ regardless of the value of x. But if the machine reports errors, the error message informing the user of the failure of the certification check means that the program has attempted to execute step 6. It could do so only if it had taken the branch in step 2, meaning that $z = 0$. If $z = 0$, then the *else* branch of statement 1 could not have been taken, meaning that $x = 0$ initially.

To prevent this type of deduction, Fenton's machine continues executing in the face of errors, but ignores the statement that would cause the violation. This satisfies the requirements. Aborting the program, or creating an exception visible to the user, would also cause information to flow against policy.

The problem with reporting of errors is that a user with lower clearance than the information causing the error can deduce the information from knowing that there has been an error. If the error is logged in such a way that the entries in the log, and the action of logging, are visible only to those who have adequate clearance, then no violation of policy occurs. But if the clearance of the user is sufficiently high, then the user can see the error without a violation of policy. Thus, the error can be logged for the system administrator (or other appropriate user), even if it cannot be displayed to the user who is running the program. Similar comments apply to any exception action, such as abnormal termination.

15.3.2 Variable Classes

The classes of the variables in the examples above are fixed. Fenton's machine alters the class of the PC as the program runs. This suggests a notion of dynamic classes, wherein a variable can change its class. For explicit assignments, the change is straightforward. When the assignment

```
y := f(x1, …, xn)
```

occurs, y's class is changed to $lub(\underline{x}_1, \ldots, \underline{x}_n)$. Again, implicit flows complicate matters.

EXAMPLE: Consider the following program (which is the same as the program in the example for the Data Mark Machine).[3]

[3] From Denning [242], Figure 5.5, p. 285.

```
proc copy(x : integer class { x };
                 var y : integer class { y });
var z : integer class variable { Low };
begin
        y := 0;
        z := 0;
        if x = 0 then z := 1;
        if z = 0 then y := 1;
end;
```

In this program, $\underline{z}$ is variable and initially *Low*. It changes when something is assigned to z. Flows are certified whenever anything is assigned to y. Suppose $\underline{y} < \underline{x}$.

If $x = 0$ initially, the first statement checks that $Low \leq \underline{y}$ (trivially true). The second statement sets z to 0 and $\underline{z}$ to *Low*. The third statement changes z to 1 and $\underline{z}$ to $lub(Low, \underline{x}) = \underline{x}$. The fourth statement is skipped (because $z = 1$). Hence, y is set to 0 on exit.

If $x = 1$ initially, the first statement checks that $Low \leq \underline{y}$ (trivially true). The second statement sets z to 0 and $\underline{z}$ to *Low*. The third statement is skipped (because $x = 1$). The fourth statement assigns 1 to y and checks that $lub(Low, \underline{z}) = Low \leq \underline{y}$ (again, trivially true). Hence, y is set to 1 on exit.

Information has therefore flowed from x to y even though $\underline{y} < \underline{x}$. The program violates the policy but is nevertheless certified.

Fenton's Data Mark Machine would detect the violation (see Exercise 4).

Denning [239] suggests an alternative approach. She raises the class of the targets of assignments in the conditionals and verifies the information flow requirements, even when the branch is not taken. Her method would raise $\underline{z}$ to $\underline{x}$ in the third statement (even when the conditional is false). The certification check at the fourth statement then would fail, because $lub(Low, \underline{z}) = \underline{x} \leq \underline{y}$ is false.

Denning ([242], p. 285) credits Lampson with another mechanism. Lampson suggested changing classes only when explicit flows occur. But all flows force certification checks. For example, when $x = 0$, the third statement sets $\underline{z}$ to *Low* and then verifies $\underline{x} \leq \underline{z}$ (which is true if and only if $\underline{x} = Low$).

15.4 Example Information Flow Controls

Like the program-based information flow mechanisms discussed above, both special-purpose and general-purpose computer systems have information flow controls at the system level. File access controls, integrity controls, and other types of access controls are mechanisms that attempt to inhibit the flow of information within a system, or between systems.

The first example is a special-purpose computer that checks I/O operations between a host and a secondary storage unit. It can be easily adapted to other purposes.

A mail guard for electronic mail moving between a classified network and an unclassified one follows. The goal of both mechanisms is to prevent the illicit flow of information from one system unit to another.

15.4.1 Security Pipeline Interface

Hoffman and Davis [428] propose adding a processor, called a *security pipeline interface* (SPI), between a host and a destination. Data that the host writes to the destination first goes through the SPI, which can analyze the data, alter it, or delete it. But the SPI does not have access to the host's internal memory; it can only operate on the data being output. Furthermore, the host has no control over the SPI. Hoffman and Davis note that SPIs could be linked into a series of SPIs, or be run in parallel.

They suggest that the SPI could check for corrupted programs. A host requests a file from the main disk. An SPI lies on the path between the disk and the host (see Figure 15–2.) Associated with each file is a cryptographic checksum that is stored on a second disk connected to the first SPI. When the file reaches the first SPI, it computes the cryptographic checksum of the file and compares it with the checksum stored on the second disk. If the two match, it assumes that the file is uncorrupted. If not, the SPI requests a clean copy from the second disk, records the corruption in a log, and notifies the user, who can update the main disk.

The information flow being restricted here is an integrity flow, rather than the confidentiality flow of the other examples. The inhibition is not to prevent the corrupt data from being seen, but to prevent the system from trusting it. This emphasizes that, although information flow is usually seen as a mechanism for maintaining confidentiality, its application in maintaining integrity is equally important.

15.4.2 Secure Network Server Mail Guard

Consider two networks, one of which has data classified SECRET[4] and the other of which is a public network. The authorities controlling the SECRET network need to

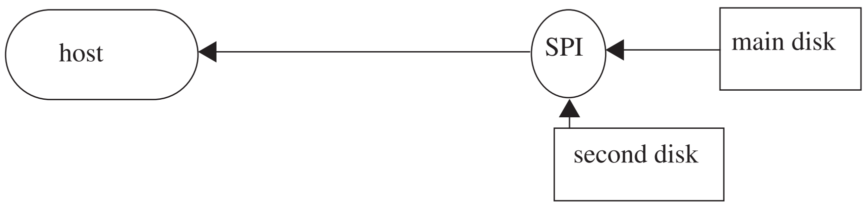

Figure 15–2 Use of an SPI to check for corrupted files.

[4] For this example, assume that the network has only one category, which we omit.

allow electronic mail to go to the unclassified network. They do not want SECRET information to transit the unclassified network, of course. The Secure Network Server Mail Guard (SNSMG) [844] is a computer that sits between the two networks. It analyzes messages and, when needed, sanitizes or blocks them.

The SNSMG accepts messages from either network to be forwarded to the other. It then applies several filters to the message; the specific filters may depend on the source address, destination address, sender, recipient, and/or contents of the message. Examples of the functions of such filters are as follows.

- Check that the sender of a message from the SECRET network is authorized to send messages to the unclassified network.
- Scan any attachments to messages coming from the unclassified network to locate, and eliminate, any computer viruses.
- Require all messages moving from the SECRET to the unclassified network to have a clearance label, and if the label is anything other than UNCLASS (unclassified), encipher the message before forwarding it to the unclassified network.

The SNSMG is a computer that runs two different message transfer agents (MTAs), one for the SECRET network and one for the unclassified network (see Figure 15–3). It uses an assured pipeline [700] to move messages from the MTA to the filter, and vice versa. In this pipeline, messages output from the SECRET network's MTA have type *a*, and messages output from the filters have a different type, type *b*. The unclassified network's MTA will accept as input only messages of type *b*. If a message somehow goes from the SECRET network's MTA to the unclassified network's MTA, the unclassified network's MTA will reject the message as being of the wrong type.

The SNSMG is an information flow enforcement mechanism. It ensures that information cannot flow from a higher security level to a lower one. It can perform other functions, such as restricting the flow of untrusted information from the unclassified network to the trusted, SECRET network. In this sense, the information flow is an integrity issue, not a confidentiality issue.

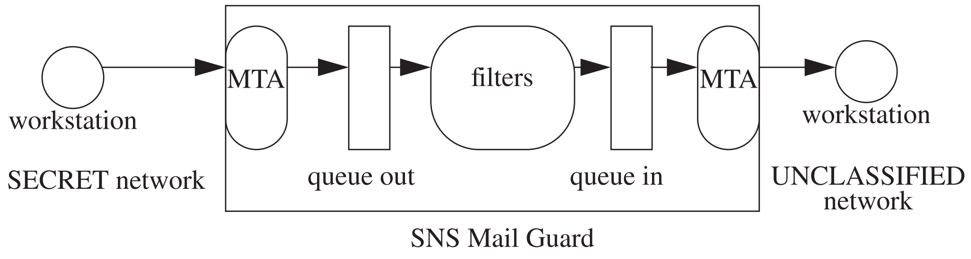

Figure 15–3 Secure Network Server Mail Guard. The SNSMG is processing a message from the SECRET network. The filters are part of a highly trusted system and perform checking and sanitizing of messages.

15.5 Summary

Two aspects of information flow are the amount of information flowing and the way in which it flows. Given the value of one variable, entropy measures the amount of information that one can deduce about a second variable. The flow can be explicit, as in the assignment of the value of one variable to another, or implicit, as in the antecedent of a conditional statement depending on the conditional expression.

Traditionally, models of information flow policies form lattices. Should the models not form lattices, they can be embedded in lattice structures. Hence, analysis of information flow assumes a lattice model.

A compiler-based mechanism assesses the flow of information in a program with respect to a given information flow policy. The mechanism either certifies that the program meets the policy or shows that it fails to meet the policy. It has been shown that if a set of statements meet the information flow policy, their combination (using higher-level language programming constructs) meets the information flow policy.

Execution-based mechanisms check flows at runtime. Unlike compiler-based mechanisms, execution-based mechanisms either allow the flow to occur (if the flow satisfies the information flow policy) or block it (if the flow violates the policy). Classifications of information may be static or dynamic.

Two example information flow control mechanisms, the Security Pipeline Interface and the Secure Network Server Mail Guard, provide information flow controls at the system level rather than at the program and program statement levels.

15.6 Further Reading

The Decentralized Label Model [660] allows one to specify information flow policies on a per-entity basis. Formal models sometimes lead to reports of flows not present in the system; Eckmann [290] discusses these reports, as well as approaches to eliminating them. Guttmann draws lessons from the failure of an information flow analysis technique [385].

Foley [327] presented a model of confinement flow suitable for nonlattice structures, and models nontransitive systems of infoormation flow. Denning [240] describes how to turn a partially ordered set into a lattice, and presents requirements for information flow policies.

The cascade problem is identified in the Trusted Network Interpretation [258]. Numerous studies of this problem describe analyses and approaches [320, 441, 631]; the problem of correcting it with minimum cost is *NP*-complete [440].

Gendler-Fishman and Gudes [351] examine a compile-time flow control mechanism for object-oriented databases. McHugh and Good describe a flow analysis tool [606] for the language Gypsy. Greenwald et al. [379], Kocher [522], Sands

[787], and Shore [826] discuss guards and other mechanisms for control of information flow.

A multithreaded environment adds to the complexity of constraints on information flow [842]. Some architectural characteristics can be used to enforce these constraints [462].

15.7 Exercises

1. Extend the semantics of the information flow security mechanism in Section 15.2.1 for records (structures).
2. Why can we omit the requirement $lub\{\ \underline{i}, \underline{b[i]}\ \} \le \underline{a[i]}$ from the requirements for secure information flow in the example for iterative statements (see Section 15.2.2.4)?
3. In the flow certification requirement for the *goto* statement in Section 15.2.2.5, the set of blocks along an execution path from b_i to IFD(b_i) excludes these endpoints. Why are they excluded?
4. Prove that Fenton's Data Mark Machine described in Section 15.3.1 would detect the violation of policy in the execution time certification of the *copy* procedure.
5. Discuss how the Security Pipeline Interface in Section 15.4.1 can prevent information flows that violate a confidentiality model. (*Hint:* Think of scanning messages for confidential data and sanitizing or blocking that data.)