16.3. 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 16–6.** 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 $\underline{\mathbf{a}} \le \underline{\mathbf{y}}$, $\underline{\mathbf{b}} \le \underline{\mathbf{y}}$, and $\underline{\mathbf{x}} \le \underline{\mathbf{y}}$, then the information flow is secure. But if $\underline{\mathbf{a}} \le \underline{\mathbf{y}}$ only when some other variable $\mathbf{z} = 1$, the compiler-based mechanism must determine whether $\mathbf{z} = 1$ 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 [274] and Denning [269].

16.3.1. 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**} ≤ **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** }

Because information can flow from any input parameter to any output parameter, the declaration must capture this:

o_s: **type class** { **i_1**, ..., **i_k**, **io_1**, ..., **io_k** }

(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:

io_s: **type class** { **i_1**, ..., **i_k**, **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** ≤ **out** and **out** ≤ **out** (the latter holding because ≤ 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**{ **a**[**i**], **i** }; for information flows into **a**[**i**], the class involved is **a**[**i**].

16.3.2. 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.

16.3.2.1. Assignment Statements

An assignment statement has the form

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

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

- $\mathbf{lub}\{\underline{x_1}, \ldots, \underline{x_n}\} \leq \underline{y}$

EXAMPLE: Consider the statement

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

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

16.3.2.2. Compound Statements

A compound statement has the form

```
begin
S1;
   ...
   Sn;
end;
```

where each of the $\mathbf{S_i}$'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

- $\mathbf{S_1}$ secure
- ...
- $\mathbf{S_n}$ 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**{ **y**, **z** } ≤ **x** for $\mathbf{S_1}$ and **lub**{ **b**, **c**, **x** } ≤ **a** for $\mathbf{S_2}$. So, the requirements for secure information flow are **lub**{ **y**, **z** } ≤ **x** and **lub**{ **b**, **c**, **x** } ≤ **a**.

16.3.2.3. Conditional Statements

A conditional statement has the form

```
if
f(x1, ..., xn) then
S1;

S2;
else

end;
```

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

- $\mathbf{S}_1$ secure
- $\mathbf{S}_2$ secure
- **lub**{ $\underline{\mathbf{x}}_1$, ..., $\underline{\mathbf{x}}_\mathbf{n}$ } ≤ **glb**{ $\underline{\mathbf{y}}$ | **y** is the target of an assignment in $\mathbf{S}_1$ and $\mathbf{S}_2$ }

As a degenerate case, if statement $\mathbf{S}_2$ is empty, it is trivially secure and has no assignments.

EXAMPLE: Consider the statements

```
if
x + y < z then
a := b;

d := b * c - x;
```

```
else

end;
```

Then the requirements for the information flow to be secure are $\underline{\mathbf{b}} \leq \underline{\mathbf{a}}$ for $\mathbf{S}_1$ and **lub**{ $\underline{\mathbf{b}}$, $\underline{\mathbf{c}}$, $\underline{\mathbf{x}}$ } ≤ $\underline{\mathbf{d}}$ for $\mathbf{S}_2$. 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**{ $\underline{\mathbf{y}}$, $\underline{\mathbf{z}}$ } ≤ $\underline{\mathbf{x}}$, $\underline{\mathbf{b}} \leq \mathbf{a}$, and **lub**{ $\underline{\mathbf{x}}$, $\underline{\mathbf{y}}$, $\underline{\mathbf{z}}$ } ≤ **glb**{ $\underline{\mathbf{a}}$, $\underline{\mathbf{d}}$ }.

16.3.2.4. Iterative Statements

An iterative statement has the form

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

where $\mathbf{x}_1$, ..., $\mathbf{x}_\mathbf{n}$ 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 $\mathbf{x}_1$, ..., $\mathbf{x}_\mathbf{n}$, 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 $\mathbf{x_1}$, ..., $\mathbf{x_n}$ 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**{ $\underline{\mathbf{x}}_1$, ..., $\underline{\mathbf{x}}_n$ } ≤ **glb**{ $\underline{\mathbf{y}}$ | **y** 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** ≤ **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 $\underline{\mathbf{i}} \le \underline{\mathbf{a}}[\underline{\mathbf{i}}]$ and $\underline{\mathbf{b}}[\underline{\mathbf{i}}] \le \underline{\mathbf{a}}[\underline{\mathbf{i}}]$; the second statement is secure because $\underline{\mathbf{i}} \le \underline{\mathbf{i}}$. Hence, the compound statement is secure if **lub**{ $\underline{\mathbf{i}}$, $\underline{\mathbf{b}}[\underline{\mathbf{i}}]$ } ≤ $\underline{\mathbf{a}}[\underline{\mathbf{i}}]$.

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**{ $\underline{\mathbf{i}}$, $\underline{\mathbf{n}}$ } ≤ **glb**{ $\underline{\mathbf{a}}[\underline{\mathbf{i}}]$, $\underline{\mathbf{i}}$ }. Putting these together, the requirement for the information flow to be secure is **lub**{ $\underline{\mathbf{i}}$, $\underline{\mathbf{n}}$ } ≤ **glb**{ $\underline{\mathbf{a}}[\underline{\mathbf{i}}]$, $\underline{\mathbf{i}}$ } (see Exercise 5).

16.3.2.5. Goto Statements

no 16.3.2.5

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

Definition 16–7. A **basic block** is a sequence of statements in a program that has one entry point and one exit point.

16.3.2.6. Procedure Calls

A procedure call has the form

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

where each of the $\mathbf{i_j}$'s is an input parameter and each of the $\mathbf{o_j}$'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 $\mathbf{x_1}$, ..., $\mathbf{x_m}$ and $\mathbf{y_1}$, ..., $\mathbf{y_n}$ 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 $\underline{\mathbf{i_j}} \le \underline{\mathbf{o_k}}$ then $\underline{\mathbf{x_j}} \le \underline{\mathbf{y_k}}$
- For **j** = 1, ..., **n** and **k** = 1, ..., **n**, if $\underline{\mathbf{o_j}} \le \underline{\mathbf{o_k}}$ then $\underline{\mathbf{y_j}} \le \underline{\mathbf{y_k}}$

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 $\mathbf{lub}\{\underline{\mathbf{x}}, \underline{\mathbf{tmp}}\} \le \underline{\mathbf{y}}$. This indicates that the formal parameters **x** and **y** have the information flow relationship $\underline{\mathbf{x}} \le \underline{\mathbf{y}}$. 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 $\underline{\mathbf{a}} \le \underline{\mathbf{b}}$.

16.3.3. 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]

[1] From Denning [269], p. 306.

```
var  sum: int

     z: int


begin

     sum := 0;
     y := 0;
     while


           y := y + 1;
     end
```

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

class { x };

class
Low;

z := 0;



z = 0 do begin
sum := sum + x;


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 **x** ≤ **y** checked.

---

[1] From Denning [269], p. 306.

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 **sum** ≤ **z**. Because **z** is **Low** and **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 };

y: int 0..1 class
Low);

y := 0;

x = 0 do
```

```
          var


begin

     while

          (* 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.

16.3.4. Concurrency

no 16.3.4

Of the many concurrency control mechanisms that are available, we choose to study information flow using semaphores [298]. Their operation is simple, and they can be used to express many higher-level constructs [148, 805]. The specific semaphore constructs are

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

```
signal(x): x := x + 1;
```

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

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

16.3.5. Soundness

Denning and Denning [274], Andrews and Reitman [34], 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 [1023] 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. (The astute reader will note that this is a form of noninterference; see Chapter 8.) 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.