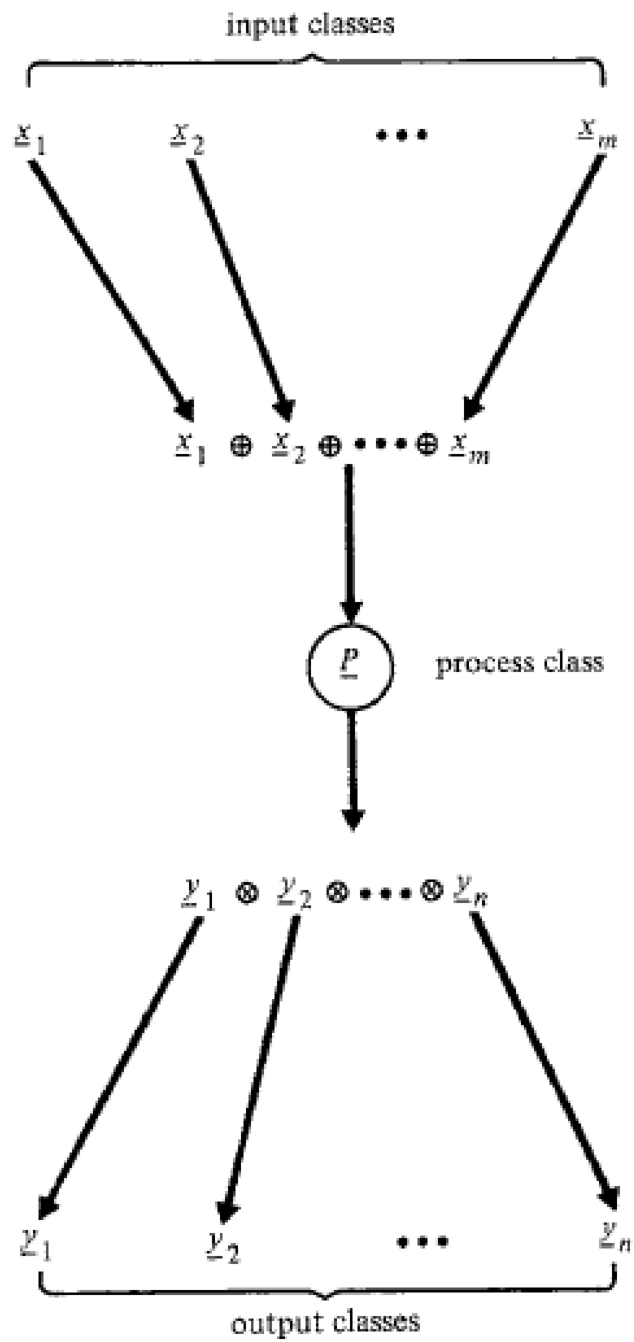
Two Lectures
Language Based Security

Flow-Secure Access Controls

- Integration of Simple flow controls into the access control mechanisms of say operating systems.
- General:
  - p can read from x1,. . . , xm and write into Y1, ... Yn only if

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

- This automatically guarantees the security of all flows, explicit or implicit, internal to the process.


input classes
$\underline{x}_1$ $\underline{x}_2$ • • • $\underline{x}_m$
$\underline{x}_1 \oplus \underline{x}_2 \oplus \cdots \oplus \underline{x}_m$
$\underline{P}$
process class
$\underline{y}_1 \otimes \underline{y}_2 \otimes \cdots \otimes \underline{y}_n$
$\underline{y}_1$ $\underline{y}_2$ • • • $\underline{y}_n$
output classes

Military Systems

- Access controls enforce both a nondiscretionary policy of information flow based on the military classification scheme(MLS), 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 subject to* integrity constraints of writing into *Top Secret objects.*

Dynamically determining Determining Labels

- OS (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 $\underline{p}$.
- Privacy: Privacy Restriction Processor is similar, except that $\underline{p}$ is determined by **lub** the of the classes opened for read, and
- whenever the process writes into a file f, the file's class is changed to $\underline{f} = \underline{f}\ \underline{\text{lub}}\ \underline{p}$

Problem of Dynamic determinations

- Suppose the procedure *copy1 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=O then z:= 1***
- ***p2: if z=O then y:= 1 .***

```
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
```

- P1 and P2 are are set to the **lub** of the classes of all objects opened for read or write operations,
- y and z are initially 0 and in class *Low,*
- *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 $\underline{p2} < \underline{y}$ is never verified.

- When x= 1, *p1* terminates with $\underline{p1} = \underline{x}$;
- *however, because z is never opened for writing,* $(z, \underline{z}) = x$ remains (0, *Low);*
- *thus,* $\underline{p2}$ *= Low, y becomes 1, and the relation Low <=* $\underline{y}$ *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 $\underline{x} \not\leq \underline{y}$.

•problem does not arise when objects and processes have fixed security classes.

•suppose *p1 runs in the minimal class needed to* read x; $\underline{p1} = \underline{x}$.

•Then *pl 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

- 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, p, reads both confidential *(High) and nonconfidential (Low) data,* then 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.*

- To enforce security within processes that handle different classes of information, the information flow internal to a process must be examined.

Flow Specifications

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* ,

- 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 | u → v is allowed}*
- The class of an input/output parameter y will be of the form (y, u1, … uk) where *u1 . . . . , u k are other inputs to y.*

- If y is an output only, the class of y will be of the form (u1, . . . , u k) (i.e., y not in <u>y</u>);
- 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;

Flow Specifications

- The class declarations are used to form a subset lattice of allowable input/output relations
- 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;

Flow Specifications: Simple Certification

Specify the flow (assuming no global variables)

```
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 .
```

Security Class implied: $\underline{x}$ <= $\underline{m}$ and $\underline{m}$ >= $\underline{y}$

```
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

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 $\underline{u} <= \underline{y}$.

Sufficient conditions for security

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

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

Security conditions for assignment:
Execution of an assignment

$$b := e$$

is secure if $\underline{e} \leq \underline{b}$.

Security conditions for compound:
Execution of the statement

$$\textbf{begin } S_1; \ldots; S_n \textbf{ end}$$

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

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
$\underline{S}_1 = \otimes\{\underline{b} \mid b \text{ is a target of an assignment in } S_1\}$,
$\underline{S}_2 = \otimes\{\underline{b} \mid b \text{ is a target of an assignment in } S_2\}$

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 $\underline{x} \oplus \underline{y} \leq \underline{z} \otimes \underline{i}$.

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
$\underline{S}_1 = \otimes \{\underline{b} \mid b \text{ is a target of a possible flow in } S_1\}$.

- Nonterminating loops can cause additional implicit flows, because execution of the remaining statements is conditioned on the loop terminating
- Even terminating loops can cause covert flows, because the execution time of a procedure depends on the number of iterations performed.
- No Good Solution

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.

```
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 .
```

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 $\underline{x} \le \underline{m}$ and $\underline{y} \le \underline{m}$ for output m, execution of a call "$max(a, b, c)$" is secure if $\underline{a} \le \underline{c}$ and $\underline{b} \le \underline{c}$.

procedure *copy2*(x: **integer class** $\langle x \rangle$;	
var y: **integer class** $\langle x \rangle$);	
"copy x to y"	
var z: **integer class** $\langle x \rangle$;	
begin	
$z := 1$;	$Low \le \underline{z}$
$y := -1$;	$Low \le \underline{y}$
while $z = 1$ **do**	$\underline{z} \le \underline{y} \otimes \underline{z}$
begin	
$y := y + 1$;	$\underline{y} \le \underline{y}$
if $y = 0$	$\underline{y} \le \underline{z}$
then $z := x$	$\underline{x} \le \underline{z}$
else $z := 0$	$Low \le \underline{z}$
end	
end	
end *copy2*	

- The flow x → y is indirect:
  - an explicit flow x →z occurs during the first iteration;
  - this is followed by an implicit flow z → y during the second iteration due to the iteration being conditioned on z.

Certification Semantics

Certification semantics.

Expression e	Semantic Actions
$f(a_1, \ldots, a_n)$	$\underline{e} := \underline{a}_1 \oplus \ldots \oplus \underline{a}_n$
Statement S	
$b := e$	$\underline{S} := \underline{b}$; 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 $\underline{y}_i \leq \underline{y}_j$ $\underline{S} := \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$

Certifications Semantics (2)

- The certification mechanism is sufficiently simple that it can be easily integrated into the analysis phase of a compiler.
- As an expression e = *f (a 1 , . . . , an) is parsed, the class* <u>e</u> *= <u>a1</u> <u>lub</u> ... <u>lub</u> <u>an</u>* computed and associated with the expression.
  – This facilitates verification of explicit and implicit flows from *a1 . . . . . an.*

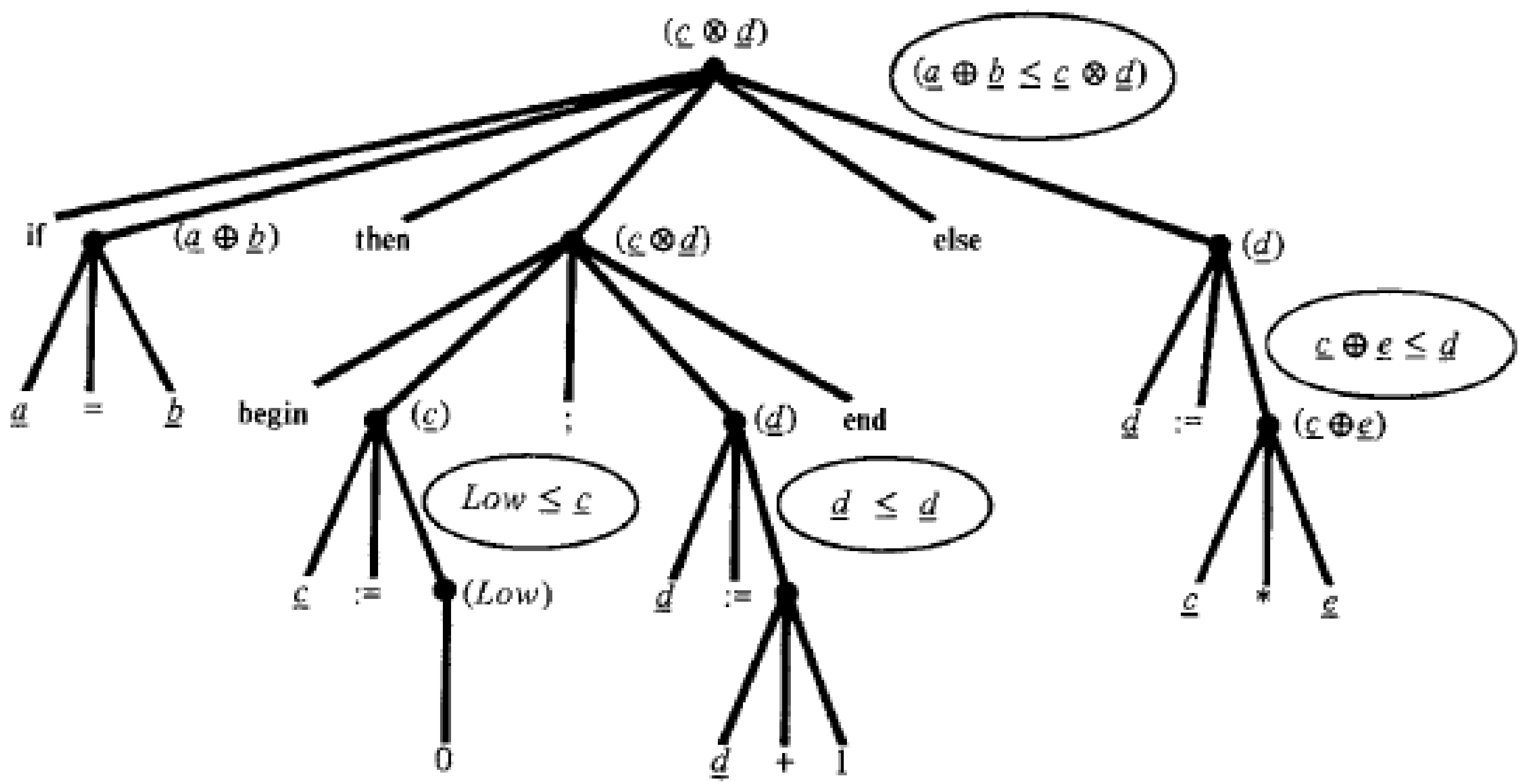
Certifications Semantics (3)

Example:

- 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 $\underline{e} = \underline{a}\ \underline{\text{lub}}\ \underline{b}\ \underline{\text{lub}}\ \underline{c}$

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

Certification of a statement.



Array Statement (1)

- *Example:*

  b := a[e].

- If e is known but *a[e] is not, then execution of this statement causes a flow*
  - *a[e] → b.*
- *If a[i] is known for i = 1 . . . . , n but e is not, it can cause a flow e* → b
  - (e.g., if *a[i] = i for i = 1, ... n, then b = e).*

Array Statement (2)

- 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.

Array Statement (3)

- If all elements *a[i] belong to the same class* <u>*a,*</u> *the certification mechanism is* easily extended to verify flows to and from arrays.
- For an array reference "a[e]", the class <u>a</u> <u>lub</u> <u>e</u> can be associated with the reference to verify flows from a and e.
- For an array assignment "a[e] := b", the relation <u>e</u> <u>≤</u> <u>a</u> can be verified along with the relation <u>b</u> <u>≤</u> <u>a</u>.
- If the elements belong to different classes, it is necessary to check only the classes *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).

What about?

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

$$\textbf{if } x \le 2 \textbf{ then } b[x] := a[x]$$

requires only that

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

NEED Further Analysis

❑As a general rule, a mechanism is needed to ensure addresses refer to the objects assumed during certification.

• Otherwise, a "a[e] := b" might cause an invalid flow b →c, where c is an object addressed by *a[e] when e is* out of range.

❑Several possible approaches

➢ check the bounds of array subscripts and pointer variables.

➢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.

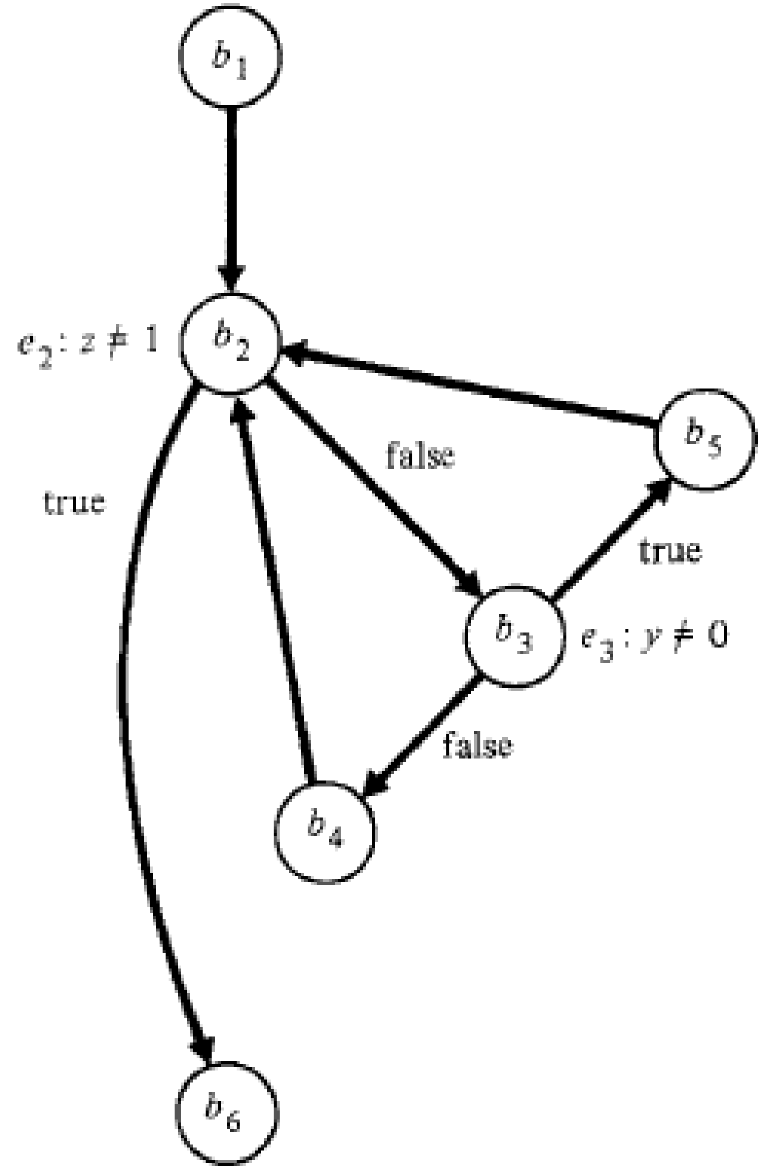
➢ 3rd method is to prove that all subscripts and pointer variables are within their bounds;

Control Structures with Unrestricted Goto's

Certifying a program with unrestricted gotos, requires a control flow analysis of the program to determine the objects receiving implicit flows.

```
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
```

- A control flow graph is constructed, showing transitions among basic blocks;
- associated with block bi is an expres., *ei that selects the successor of bi in* the graph
- The security class of block *bi is the* <u>*glb*</u> *of the classes* of all objects that are the targets of flows in bi (if there are no such objects, this class is *High).*
- The **immediate forward dominator** ***IFD(bi) is computed for each*** block bi
    - It is the closest block to *bi among the set of blocks that lie on all paths* from bi to the program exit and, therefore, is the point where the divergent execution paths conditioned on ei converge.
- Define *Bi as the set of blocks on some path* from *bi to IFD(bi) excluding bi and IFD(bi).*
- The security class of *Bi is* <u>*Bi*</u> *=* <u>*glb*</u> *{*<u>*bj | bj in Bi*</u>*}*



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

Because the only blocks directly conditioned on the selector expression ei of *bi are those in Bi, the program is secure if each block bi is independently secure* and $\underline{ei} \leq \underline{Bi}$ for all i.

Concurrency and Synchronization

```
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
```

- 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 → y in is caused by wait(s) and signal(s), which read and modify the value of s as follows

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

- 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
- However, that information can flow along synchronization channels even when the procedures terminate normally

```
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
```

- When x = 0, process 2 executes before process 3, so the final value of y is 0;
- when x != 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, we require the class of every such variable y satisfy the relation

$\underline{s} \leq \underline{y}$.

Concurrency: flow over Global channels

```
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
```

We require <u>e0</u> ≤ <u>y</u> and <u>e1</u> ≤ <u>y</u>

e0 and *e1 playing the role of* semaphores *s0 and s1*

- A signaling process can covertly leak a value by making the length of delay proportional to the loop.

- Similar problem of loops

Abnormal terminations

```
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.
- ❑ x→ y , as y:=1 is conditioned on x. Thus there is a flow even though that is not the target
- ❖ Such covert channels are not necessarily 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.

```
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 .
```

The problem of abnormal termination can be handled by inhibiting all traps except those for which actions have been explicitly defined in the program

- 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 → y because execution of the assignment to y is conditioned on the value of *sum, and thus* x, but we do not require that $\underline{x} \leq \underline{y}$.

Adding a condition statement

- Example: If the statement

  on overflow sum *do* z "= 1

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