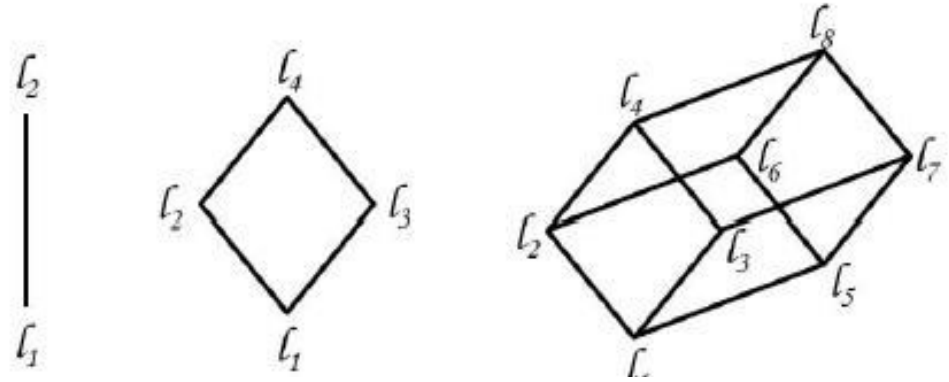
Denning's Information Flow Model (DFM)

DFM = (S,O, SC, ≤, ⊕) S

- S is a set of subjects /principals (active agents responsible for all info. flow),
- O is a set of objects (info. containers),
- SC is a set of security classes ,
- ≤ is a binary relation on security classes that specifies permissible info. flows .
- sc1≤ sc2 means: info. in security class sc1 is allowed/permitted to flow into security class sc2,
- ⊕ is the class-combining binary operator (assoc. & comm.) that specifies, for any pair of operand classes, the class in which the result of any binary function on values from the operand classes belongs

Example 1. An example of security classes in DFM could be $SC = \{l_1, l_2\}$, with $l_1 < l_2$ as the ordering. This means that information at security class l_1 is allowed to flow to security class l_2, but not vice-versa.



The pictorial representations are to be interpreted as follows: if there is an upward path from class l to l', then information is allowed to flow from class l to l'. For example, in SC_1, information is allowed to flow from class l_1 to l_2 but not vice-versa. Similarly, in SC_2, information is allowed to flow from l_2 to l_4 but not to l_1 or l_3. In SC_3, information is allowed to flow from l_3 to l_4, l_7 and l_8 but to no others. □

Example 2. Consider the security lattice SC_1 given in Example 1. Let s_1 and s_2 be the only subjects in the system i.e., $S = \{s_1, s_2\}$. Similarly, let $O = \{o_1, o_2\}$.

An example information-flow policy is given by $\lambda_1(s_1) = \lambda_1(o_1) = l_1$ and $\lambda_1(s_2) = \lambda_1(o_2) = l_2$. According to λ_1, s_1 can read o_1, because $\lambda_1(o_1) \preccurlyeq \lambda_1(s_1)$ is satisfied by λ_1. Similarly, it is easy to verify that λ_1 permits s_1 to write o_1 and o_2 (because $\lambda_1(s_1) = l_1 \preccurlyeq \lambda_1(o_2) = l_2$) but not read o_2; and s_2 can read and write o_2 and can read but not write o_1.

Another information-flow policy could be defined by $\lambda_2(o_1) = l_1$ and $\lambda_2(s_1) = \lambda_2(s_2) = \lambda_2(o_2) = l_2$. If policy λ_2 is enforced, then both s_1 and s_2 are allowed to read and write o_2 and read but not write o_1. □

Security of Flows

- A system enforcing Denning's flow model *DFM is secure if and only if execution of any sequence of operations* of the system cannot give rise to a flow that violates the permissible information flow relation.
- Further, the natural conditions required of information flow force the structure

(SC, ≤) to be a lattice with $\oplus$ as the least upper bound operator

Flow Policy: Lattice Structure

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

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

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

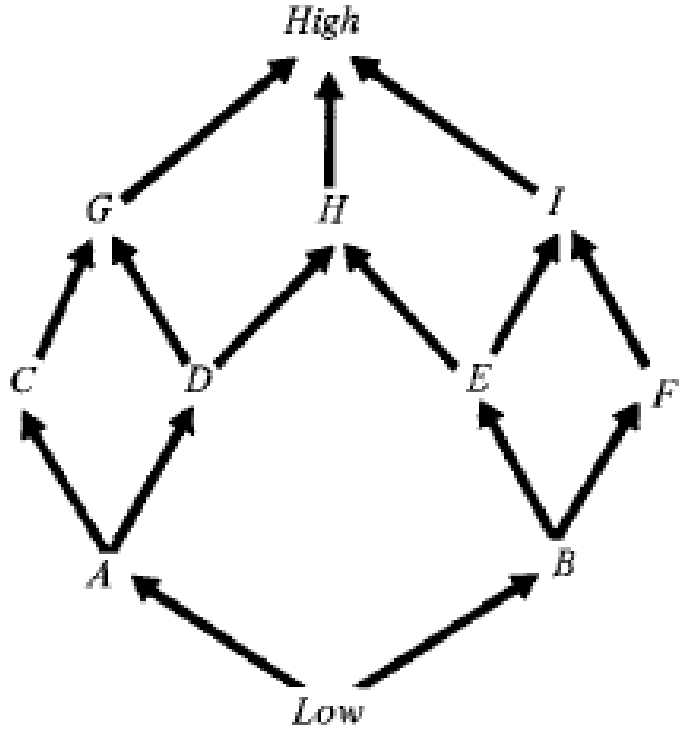
1. $A \preceq C$ and $B \preceq C$, and
2. $A \preceq D$ and $B \preceq D$ implies $C \preceq D$ for all D in SC.

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

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

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

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

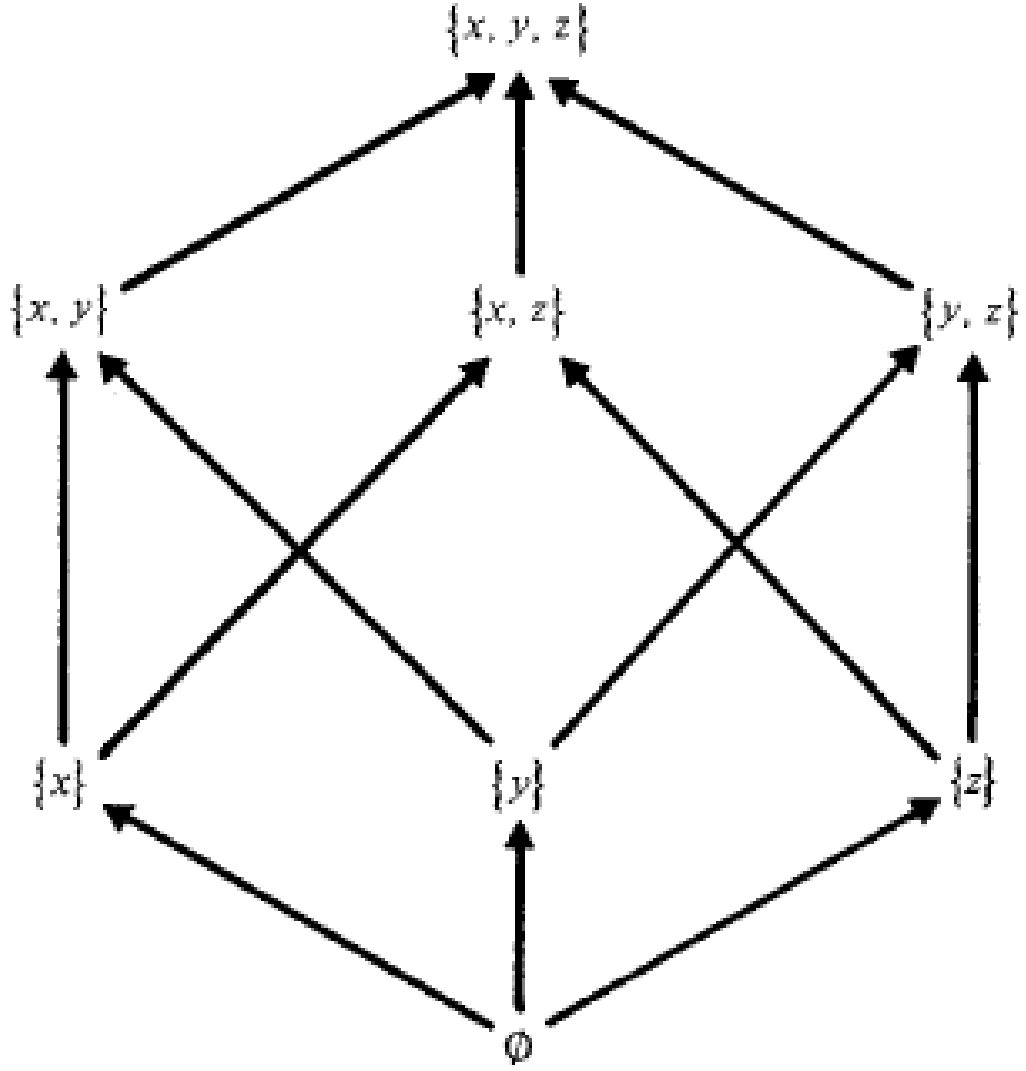


$A \oplus C = C$, $A \otimes C = A$
$C \oplus D = G$, $C \otimes D = A$
$C \oplus D \oplus E = High$, $C \otimes D \otimes E = Low$. ■

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

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

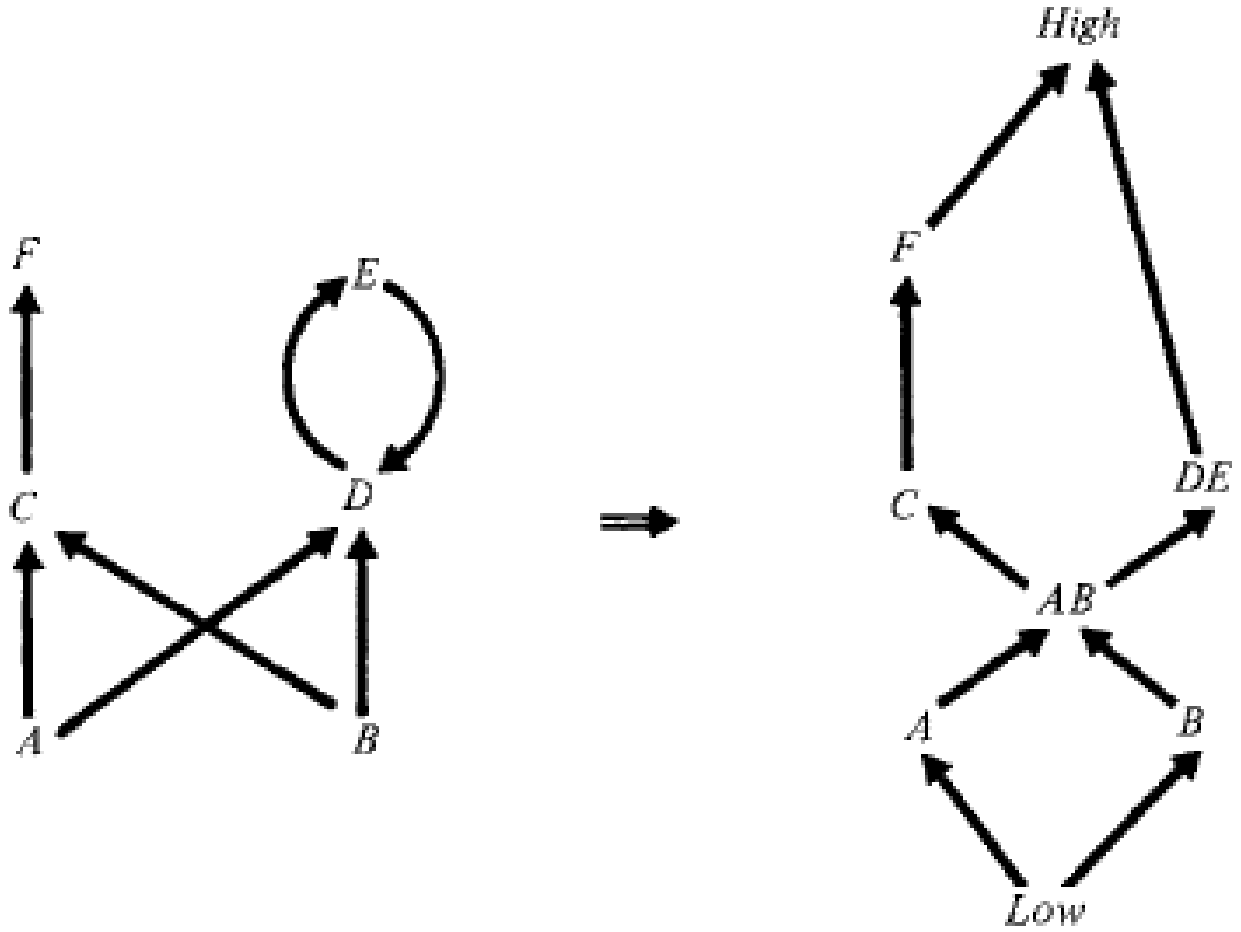
Subset lattice



Transformation of nonlattice policy into a lattice

- Take an arbitrary flow policy P = *(SC, ≤) and transform* it into a lattice P' = *(SC', ≤');*
- *classes A and B in SC have corresponding* classes A' and B' in *SC' such that A ≤ B in P if and only if A' ≤' B' in P'*
- Flow is authorized under P if and only if it is authorized under P', where objects bound to class A in P are bound to class A' in P'.
  - Requires only that the relation ≤ be reflexive and transitive.
  - To derive a relation ≤' that is also antisymmetric, classes forming cycles are compressed into single classes.
  - To provide least upper and greatest lower bound operators, new classes are added.

Example



Flow Properties

Transitivity of the relation ≤ implies any indirect flow

x → y resulting from a sequence of flows

X = Z 0 → Z 1 . . . → Zn_ 1 → Zn = y

is permitted if each flow *Zi_1 → Zi (1 ≤ i ≤ n) is permitted, because*

- $\underline{x} = \underline{z0} \leq \underline{Z} \leq \ldots \leq \underline{Zn\text{-}1} \leq \underline{Zn} = \underline{y}$ implies $x \leq y$ *(here reference is to security classes of the variables/objects)*

- *Thus:* an enforcement mechanism need only verify direct flows.

Examples

- *Example:*
- The security of the indirect flow x → y caused by executing the sequence of statements
- *z := x; y:=z*
- automatically follows from the security of the individual statements; that is,
- $\underline{x} \leq \underline{z}$ and $\underline{z} \leq \underline{y}$ implies $\underline{x} \leq \underline{y}$. (refers to security classes of x,y,z)

- *Example:*
- Consider the sequence (x =0 or 1 initially)

  z := o;

  if x= 1 then z:= 1; Indirect (xto z)

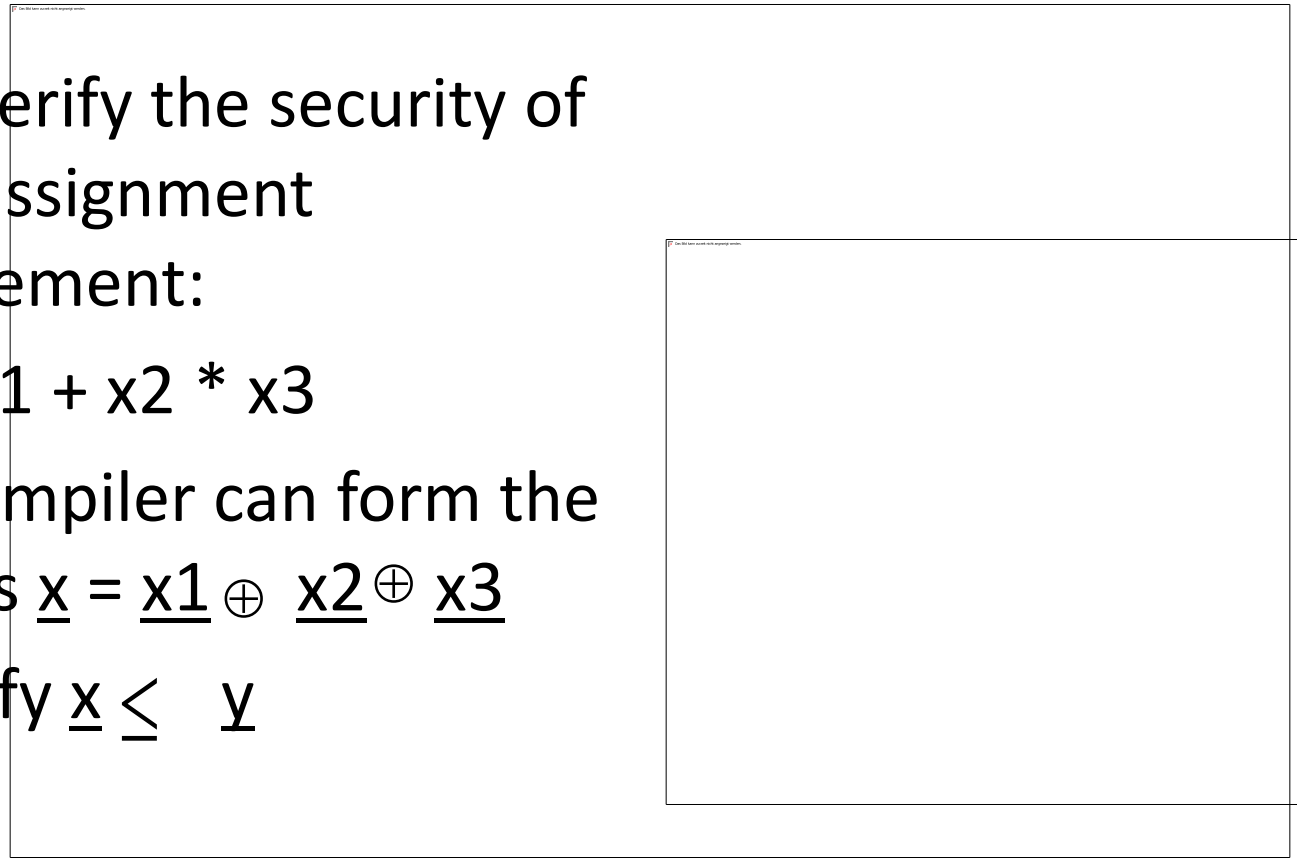
  y:=z, Direct z to y

Examples

- To verify the security of an assignment statement:

y := x1 + x2 * x3

- a compiler can form the class $\underline{x} = \underline{x1} \oplus \underline{x2} \oplus \underline{x3}$
- Verify $\underline{x} \leq \underline{y}$



Examples

- To verify the security of an if statement

```
if x then
begin
Y1:= O;
Y2 := O;
Y3 := 0
End
```

- Form $\underline{y1} \otimes \underline{y2} \otimes \underline{y3}$
- verify the implicit flows x → yi (i = 1, 2, 3)

  by checking $\underline{x} \leq \underline{y}$

Examples

- X := 1
- X:=x+1
- x := 'On a clear disk you can seek forever‘

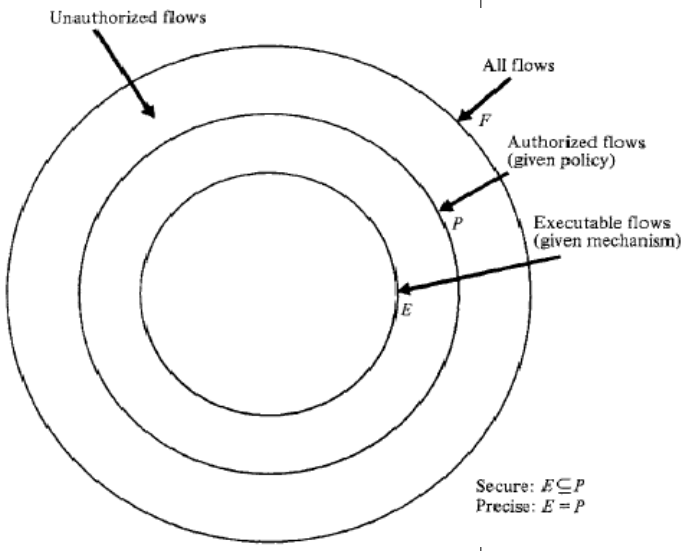
Is always authorized

Because

- *<u>Low</u> is an identity on ⊕ the class of the expression*
- "x + 1" is simply *<u>x</u> ⊕ Low = <u>x</u>.*

Security and Precision

- F be the set of all possible flows in an information flow system,
- P be the subset of F authorized by a given flow policy,
- E be the subset of F "executable“ given the flow control mechanisms in operation.
- The system is secure if E⊆ P;
  - that is, all executable flows are authorized.
- A secure system is precise if E = P;
  - that is, all authorized flows are executable.



Enforcements

Example

- Y:= k* x
- Policy: $\underline{k} \leq \underline{y}$ but $\underline{x} \not\leq \underline{y}$
- A mechanism that always prohibits execution of this statement will provide security.
  - k = 0 or H(x) =0

- To design a mechanism that verifies $\underline{x} \leq \underline{y}$ the relation only for actual flows x → y is considerably more difficult than designing one that verifies the relation $\underline{x} \leq \underline{y}$ for any operation that can potentially cause a flow x → y

Un-decidability

- Showing whether a system is secure or precise
- if f(n) *halts* ***then y := x else y := 0***

where f is an arbitrary function and $\underline{x}$ $!\leq \underline{y}$

- it is theoretically impossible to construct a mechanism that is both secure and precise

Example

```
y:=x;
z:=y;
y:=0
```

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

Channels of Flow

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

- Example:
- Process p can transfer a value x to a process q through the lock bit of a shared file f
- p arranges regular intervals of use and nonuse of x according to the binary representation of x;
- q requests use of the file each interval, and determines the corresponding bit of x according to whether the request is granted.

Dynamically Enforcing Security for Flows

Explicit Flows

- Occurs due to assignment
- Y := y (x1, … , xn)
- xi → y can be verified by
- $\underline{x1} \oplus \ldots \oplus \underline{xn} \leq \underline{y}$
- If not an error can be generated (skipped/aborted)

Implicit Flows

- Verifying implicit flows to an object only at the time of an explicit assignment to the object is insecure.
- Verifying the requirement when doing y:= 1 in " if x=1 then y:=1 fi" would be insecure!

- What is to be done?

Handling Implicit Flows (Execution Based Mechanisms)

- Security can be enforced by verifying flows to an object only at the time of explicit assignments to the object.
- Except that:
  - attempted security violations cannot generally be reported.
  - I.e., if an unauthorized implicit flow is detected at the time of an assignment, not only must that assignment be skipped, but the error must not be reported to the user,
  - and the program must keep running as though nothing has happened.
- The program cannot abort or even generate an error message to the user unless the user's clearance is at least that of the information causing the flow violation. I
  - It would otherwise be possible to use the error message or abnormal termination to leak high security data to a user with a low security clearance.
- Note that the program must terminate in a low security state;
  - that is, any information encoded in the program counter from tests must belong to the class ***Low***.

Example

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

if x = 1 then y := 1

is described by

if x= 1

then if $\underline{x} \leq \underline{y}$ then

y := 1 else

skip

else skip .

- Suppose x is 0 or 1, y is initially *O,*
- *$\underline{x}$ = High, and $\underline{y}$= Low;*

  thus, the flow x → y is not secure.
- Because the assignment to y is skipped both when x = 1 (because the security check fails) and when x = 0 (because the test "x = 1“ fails), y is always 0 when the statement terminates, thereby giving no information about x.
- Note that if an error flag E is set to 1 when the security check fails, then the value of x is encoded in the flag (E = 1 implies x = 1, E = 0 implies x =0).

Conditional Assignment

- Y := f(x1,…,xm)

Secure because y is not changed;
Thus no implicit flow

Conditioned on

x(m+1)… xn

- Then the explicit and implicit flow to y can be validated by checking that the relation

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

1.Simpler to construct a run-time enforcement mechanism if all implicit and explicit flows can be validated only at the time of actual assignments.

2. Such a mechanism is likely to be more precise than one that checks implicit flows that occur in the absence of explicit assignments.

Example

- If x=1 then y:=1 else z:=1 --- x is High

- x=1 , y = high and z = Low
- x ≠1 , y = low and z = high
- What happens if you test
- **x ≤ y and x ≤ z**

Error Flagging (1)

- An error flag can be securely logged in a record having a security class **at least that of the information**.
- Although it is insecure to report it to a user in a lower class, the capacity of the leakage channel is at most 1 bit (because at most 1 bit can be encoded in a 1-bit flag).
- The error message can, however, potentially disclose the exact values of all variables in the system.

- Suppose information in the system is encoded in variables x1, ... xn, y is zero and $\underline{xi} \not\leq \underline{y}$
- Then
- If (x1 = val1) & … & (xn=valn)
  then y :=1
- Generates an error message when

(x1 = val1) & … & (xn=valn)

Then all values are known due to error message. Assume that only one xi != vali then little can be deduced from y

Error Flagging (2)

- If not [(x1 = val1) & … & (xn=valn)]
  then y :=1
- terminates successfully without causing security error when x1 = *val1 . . . x n= valn, leaking the exact values of x1 . . . x n.*
- Note, however, that the values of x1, . . . , x n are not encoded in y, because y will be 0 even when the test succeeds.
- The values are encoded in the termination status of the program, which is why only 1 bit of information can be leaked.

Error Flags (3)

- It is clearly unsatisfactory not to report errors, but errors can be logged, and offending programs removed from the system.
- This solution may be satisfactory in most cases.
- It is not satisfactory, however, if the 1 bit of information is sufficiently valuable (e.g., a signal to attack).

- Consider the execution of the procedure *copy1.*
- *Suppose* the local variable z has a variable class (initially *Low), z is changed whenever* z is assigned a value, and flows into y are verified whenever y is assigned a value.

- Procedure -- executed with x = 0, the test "x = 0“ succeeds and
- (z, z) becomes (1, x); hence the test "z = 0" fails, and y remains 0.
- If it is executed with x = 1, the test "x = 0" fails, so (z, z) remains (0, *Low); hence the test "z = 0" succeeds, y is assigned the value 1,* and the relation *Low ≤ y is verified.*
- In either case, execution terminates with y = x, but without verifying the relation x ≤ y

```
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
```