ECS 235B, Lecture 19

February 22, 2019

Composition of Policies

- Two organizations have two security policies
- They merge
    - How do they combine security policies to create one security policy?
    - Can they create a coherent, consistent security policy?

The Problem

- Single system with 2 users
  - Each has own virtual machine
  - Holly at system high, Lara at system low so they cannot communicate directly
- CPU shared between VMs based on load
  - Forms a *covert channel* through which Holly, Lara can communicate

Example Protocol

- Holly, Lara agree:
  - Begin at noon
  - Lara will sample CPU utilization every minute
  - To send 1 bit, Holly runs program
    - Raises CPU utilization to over 60%
  - To send 0 bit, Holly does not run program
    - CPU utilization will be under 40%
- Not “writing” in traditional sense
  - But information flows from Holly to Lara

Policy vs. Mechanism

- Can be hard to separate these
- In the abstract: CPU forms channel along which information can be transmitted
    - Violates *-property
    - Not “writing” in traditional sense
- Conclusion:
    - Bell-LaPadula model does not give sufficient conditions to prevent communication, *or*
    - System is improperly abstracted; need a better definition of “writing”

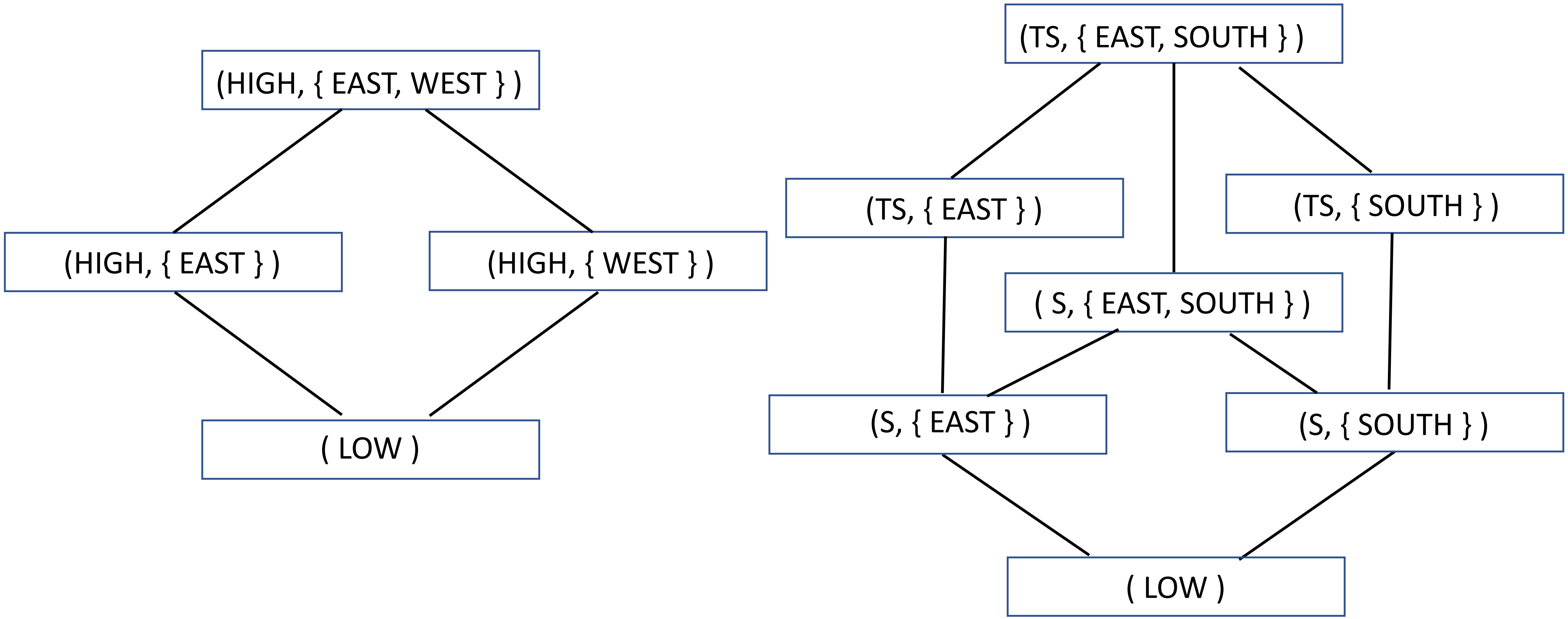
Composition of Bell-LaPadula

- Why?
  - Some standards require secure components to be connected to form secure (distributed, networked) system
- Question
  - Under what conditions is this secure?
- Assumptions
  - Implementation of systems precise with respect to each system's security policy

Issues

- Compose the lattices
- What is relationship among labels?
  - If the same, trivial
  - If different, new lattice must reflect the relationships among the levels

Example


(HIGH, { EAST, WEST })
(HIGH, { EAST })
(HIGH, { WEST })
(LOW)
(TS, { EAST, SOUTH })
(TS, { EAST })
(TS, { SOUTH })
(S, { EAST, SOUTH })
(S, { EAST })
(S, { SOUTH })
(LOW)

Analysis

- Assume S < HIGH < TS
- Assume SOUTH, EAST, WEST different
- Resulting lattice has:
  - 4 clearances (LOW < S < HIGH < TS)
  - 3 categories (SOUTH, EAST, WEST)

Same Policies

- If we can change policies that components must meet, composition is trivial (as above)
- If we *cannot*, we must show composition meets the same policy as that of components; this can be very hard

Different Policies

- What does “secure” now mean?
- Which policy (components) dominates?
- Possible principles:
  - Any access allowed by policy of a component must be allowed by composition of components (*autonomy*)
  - Any access forbidden by policy of a component must be forbidden by composition of components (*security*)

Implications

- Composite system satisfies security policy of components as components' policies take precedence
- If something neither allowed nor forbidden by principles, then:
  - Allow it (Gong & Qian)
  - Disallow it (Fail-Safe Defaults)

Example

- System X: Bob can’t access Alice’s files
- System Y: Eve, Lilith can access each other’s files
- Composition policy:
  - Bob can access Eve’s files
  - Lilith can access Alice’s files
- Question: can Bob access Lilith’s files?

Solution (Gong & Qian)

- Notation:
  - (*a*, *b*): *a* can read *b*'s files
  - AS(*x*): access set of system *x*
- Set-up:
  - AS(X) = $\varnothing$
  - AS(Y) = { (Eve, Lilith), (Lilith, Eve) }
  - AS(X$\cup$Y) = { (Bob, Eve), (Lilith, Alice), (Eve, Lilith), (Lilith, Eve) }

Solution (Gong & Qian)

- Compute transitive closure of AS($X \cup Y$):
  - AS($X \cup Y$)$^+$ = { (Bob, Eve), (Bob, Lilith), (Bob, Alice), (Eve, Lilith), (Eve, Alice), (Lilith, Eve), (Lilith, Alice) }
- Delete accesses conflicting with policies of components:
  - Delete (Bob, Alice)
- (Bob, Lilith) in set, so Bob can access Lilith's files

Idea

- Composition of policies allows accesses not mentioned by original policies
- Generate all possible allowed accesses
  - Computation of transitive closure
- Eliminate forbidden accesses
  - Removal of accesses disallowed by individual access policies
- Everything else is allowed
- Note: determining if access allowed is of polynomial complexity

Information Flow

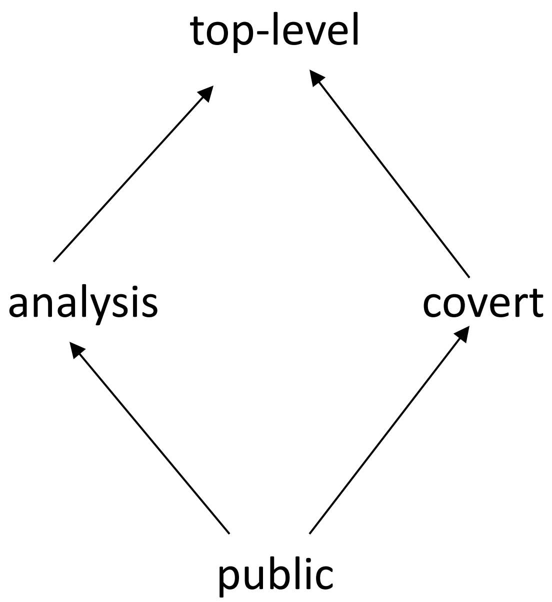
- Basics and background
    - Entropy
- Non-lattice flow policies
- Compiler-based mechanisms
- Execution-based mechanisms
- Examples
    - Privacy and cell phones
    - Firewalls

Nontransitive Flow Policies

- Government agency information flow policy (on next slide)
- Entities public relations officers PRO, analysts A, spymasters S
  - *confine*(PRO) = [public, analysis]
  - *confine*(A) = [analysis, top-level]
  - *confine*(S) = [covert, top-level]

Information Flow

- By confinement flow model:
  - PRO ≤ A, A ≤ PRO
  - PRO ≤ S
  - A ≤ S, S ≤ A
- Data *cannot* flow to public relations officers; not transitive
  - S ≤ A, A ≤ PRO
  - S ≤ PRO is *false*



Transforming Into Lattice

- Rough idea: apply a special mapping to generate a subset of the power set of the set of classes
  - Done so this set is partially ordered
  - Means it can be transformed into a lattice
- Can show this mapping preserves ordering relation
  - So it preserves non-orderings and non-transitivity of elements corresponding to those of original set

Dual Mapping

- $R = (SC_R, \leq_R, join_R)$ reflexive info flow policy
- $P = (S_P, \leq_P)$ ordered set
  - Define *dual mapping* functions $l_R, h_R: SC_R \rightarrow S_P$
    - $l_R(x) = \{ x \}$
    - $h_R(x) = \{ y \mid y \in SC_R \wedge y \leq_R x \}$
  - S_P contains subsets of SC_R; $\leq_P$ subset relation
  - Dual mapping function *order preserving* iff

$$(\forall a, b \in SC_R)[\, a \leq_R b \Leftrightarrow l_R(a) \leq_P h_R(b) \,]$$

Theorem

Dual mapping from reflexive information flow policy R to ordered set P order-preserving

Proof sketch: all notation as before

($\Rightarrow$) Let $a \leq_R b$. Then $a \in l_R(a)$, $a \in h_R(b)$, so $l_R(a) \subseteq h_R(b)$, or $l_R(a) \leq_P h_R(b)$

($\Leftarrow$) Let $l_R(a) \leq_P h_R(b)$. Then $l_R(a) \subseteq h_R(b)$. But $l_R(a) = \{ a \}$, so $a \in h_R(b)$, giving $a \leq_R b$

Information Flow Requirements

- Interpretation: let *confine*(x) = [$\underline{x}_L$, $\underline{x}_U$], consider class $\underline{y}$
  - Information can flow from x to element of $\underline{y}$ iff $\underline{x}_L \leq_R \underline{y}$, or $l_R(\underline{x}_L) \subseteq h_R(\underline{y})$
  - Information can flow from element of $\underline{y}$ to x iff y $\leq_R \underline{x}_U$, or $l_R(\underline{y}) \subseteq h_R(\underline{x}_U)$

Revisit Government Example

- Information flow policy is *R*
- Flow relationships among classes are:

public $\leq_R$ public
public $\leq_R$ analysis
public $\leq_R$ covert
public $\leq_R$ top-level
analysis $\leq_R$ top-level

analysis $\leq_R$ analysis
covert $\leq_R$ covert
covert $\leq_R$ top-level
top-level $\leq_R$ top-level

Dual Mapping of R

- Elements l_R, h_R:
  - l_R(public) = { public }
  - h_R(public = { public }
  - l_R(analysis) = { analysis }
  - h_R(analysis) = { public, analysis }
  - l_R(covert) = { covert }
  - h_R(covert) = { public, covert }
  - l_R(top-level) = { top-level }
  - h_R(top-level) = { public, analysis, covert, top-level }

confine

- Let *p* be entity of type PRO, *a* of type A, *s* of type S
- In terms of *P* (not *R*), we get:
  - *confine*(*p*) = [{ public }, { public, analysis }]
  - *confine*(*a*) = [{ analysis }, { public, analysis, covert, top-level }]
  - *confine*(*s*) = [{ covert }, { public, analysis, covert, top-level }]

And the Flow Relations Are …

- $p \to a$ as $l_R(p) \subseteq h_R(a)$
  - $l_R(p)$ = { public }
  - $h_R(a)$ = { public, analysis, covert, top-level }
- Similarly: $a \to p$, $p \to s$, $a \to s$, $s \to a$
- But $s \to p$ is false as $l_R(s) \not\subset h_R(p)$
  - $l_R(s)$ = { covert }
  - $h_R(p)$ = { public, analysis }

Analysis

- (S_P, $\leq_P$) is a lattice, so it can be analyzed like a lattice policy
- Dual mapping preserves ordering, hence non-ordering and non-transitivity, of original policy
  - So results of analysis of (S_P, $\leq_P$) can be mapped back into (SC_R, $\leq_R$, $join_R$)

Compiler-Based Mechanisms

- Detect unauthorized information flows in a program during compilation
- Analysis not precise, but secure
  - If a flow *could* violate policy (but may not), it is unauthorized
  - No unauthorized path along which information could flow remains undetected
- Set of statements *certified* with respect to information flow policy if flows in set of statements do not violate that policy

Example

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

else *y* := *b*;

- Information flows from *x* and *a* to *y*, or from *x* and *b* to *y*
- Certified only if $\underline{x} \le \underline{y}$ and $\underline{a} \le \underline{y}$ and $\underline{b} \le \underline{y}$
  - Note flows for *both* branches must be true unless compiler can determine that one branch will *never* be taken

Declarations

- Notation:

  $$\mathtt{x}\texttt{: }\mathbf{int\ class}\ \{\ \mathtt{A},\ \mathtt{B}\ \}$$

  means *x* is an integer variable with security class at least *lub*{ A, B }, so *lub*{ A, B } ≤ $\underline{x}$
- Distinguished classes *Low*, *High*
  - Constants are always *Low*

Input Parameters

- Parameters through which data passed into procedure
- Class of parameter is class of actual argument

$$i_p\texttt{: }\textbf{\textit{type}}\ \textbf{class}\ \{\ \mathtt{i_p}\ \}$$

Output Parameters

- Parameters through which data passed out of procedure
  - If data passed in, called input/output parameter
- As information can flow from input parameters to output parameters, class must include this:

$$o_p\text{: } \textbf{\textit{type}} \ \textbf{class} \ \{ \ r_1, \ \ldots, \ r_n \ \}$$

  where r_i is class of *i*th input or input/output argument

Example

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

- Require <u>x</u> ≤ <u>out</u> and <u>out</u> ≤ <u>out</u>

Array Elements

- Information flowing out:

  `… := a[i]`

  Value of *i*, *a*[*i*] both affect result, so class is lub{ <u>*a*[*i*]</u>, <u>*i*</u> }
- Information flowing in:

  `a[i] := …`
- Only value of *a*[*i*] affected, so class is <u>*a*[*i*]</u>

Assignment Statements

x := *y* + *z*;

- Information flows from *y*, *z* to *x*, so this requires lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$

More generally:

`y := f(x₁, …, xₙ)`

- the relation lub{ $\underline{x}_1$, …, x_n } ≤ $\underline{y}$ must hold

Compound Statements

x := *y* + *z*; *a* := *b* * *c* – *x*;

- First statement: lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
- Second statement: lub{ $\underline{b}$, $\underline{c}$, $\underline{x}$ } ≤ $\underline{a}$
- So, both must hold (i.e., be secure)

More generally:

S_1; … S_n;

- Each individual S_i must be secure

Conditional Statements

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

- Statement executed reveals information about *x*, *y*, *z*, so lub{ $\underline{x}$, $\underline{y}$, $\underline{z}$ } ≤ glb{ $\underline{a}$, $\underline{d}$ }

More generally:

if $f(x_1, \ldots, x_n)$ then S_1 else S_2; end

- S_1, S_2 must be secure
- lub{ $\underline{x}_1$, …, $\underline{x}_n$ } ≤ glb{$\underline{y}$ | *y* target of assignment in S_1, S_2 }

Iterative Statements

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

- Same ideas as for "if", but must terminate

More generally:

while $f(x_1, \ldots, x_n)$ do S;

- Loop must terminate;
- *S* must be secure
- lub{ $\underline{x}_1, \ldots, \underline{x}_n$ } ≤ glb{$\underline{y}$ | y target of assignment in *S* }

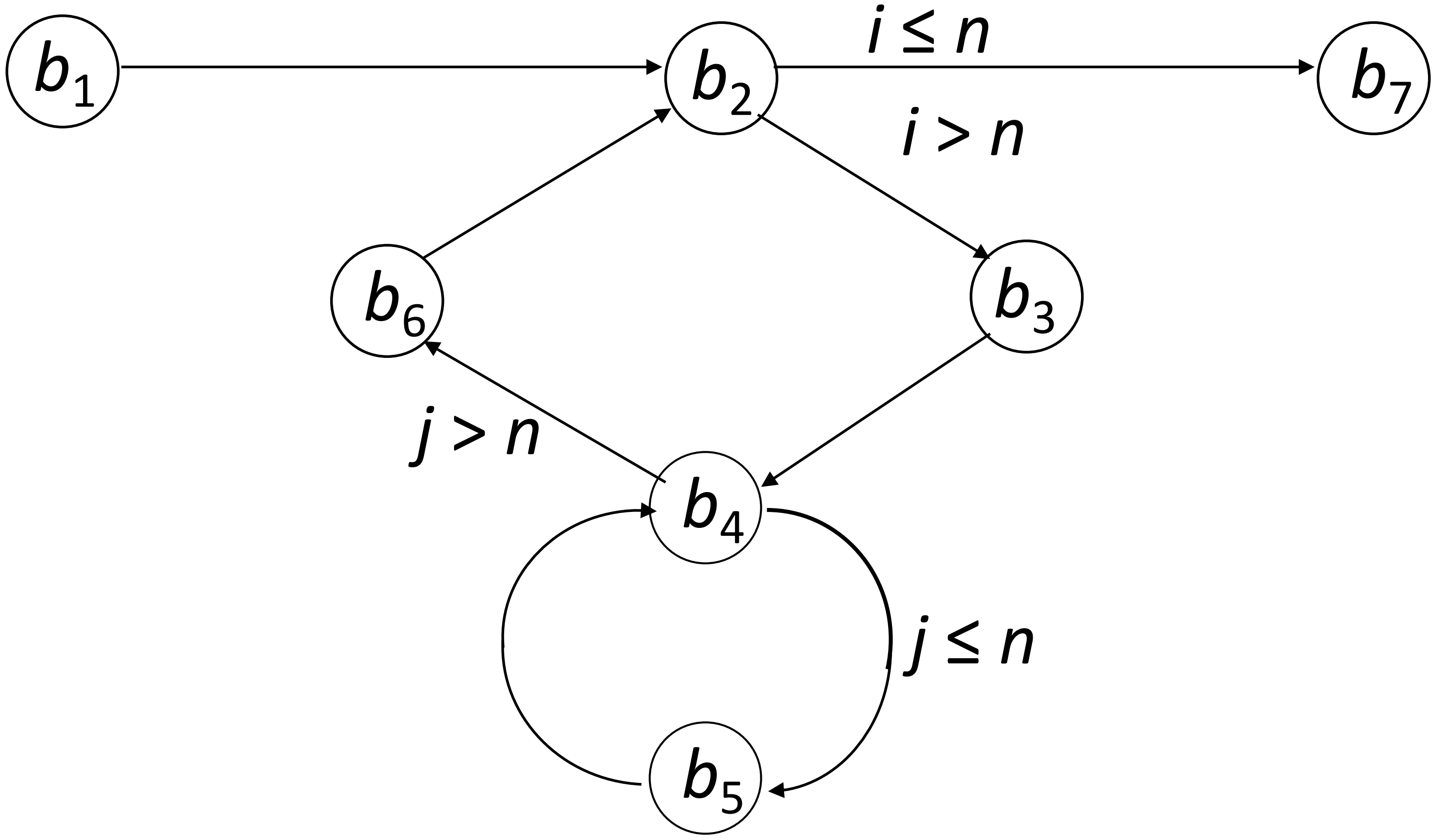
Goto Statements

- No assignments
  - Hence no explicit flows
- Need to detect implicit flows
- *Basic block* is sequence of statements that have one entry point and one exit point
  - Control in block *always* flows from entry point to exit point

Example Program

```
proc tm(x: array[1..10][1..10] of integer class {x};
                  var y: array[1..10][1..10] of integer class {y});
var i, j: integer class {i};
begin
b1     i := 1;
b2 L2: if i > 10 goto L7;
b3     j := 1;
b4 L4: if j > 10 then goto L6;
b5     y[j][i] := x[i][j]; j := j + 1; goto L4;
b6 L6: i := i + 1; goto L2;
b7 L7:
end;
```

Flow of Control



IFDs

- Idea: when two paths out of basic block, implicit flow occurs
  - Because information says *which* path to take
- When paths converge, either:
  - Implicit flow becomes irrelevant; or
  - Implicit flow becomes explicit
- *Immediate forward dominator* of basic block *b* (written IFD(*b*)) is first basic block lying on all paths of execution passing through *b*

IFD Example

- In previous procedure:
  - IFD(b_1) = b_2 one path
  - IFD(b_2) = b_7 $b_2 \rightarrow b_7$ or $b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_7$
  - IFD(b_3) = b_4 one path
  - IFD(b_4) = b_6 $b_4 \rightarrow b_6$ or $b_4 \rightarrow b_5 \rightarrow b_6$
  - IFD(b_5) = b_4 one path
  - IFD(b_6) = b_2 one path

Requirements

- B_i is set of basic blocks along an execution path from b_i to IFD(b_i)
  - Analogous to statements in conditional statement
- $x_{i1}, \ldots, x_{in}$ variables in expression selecting which execution path containing basic blocks in B_i used
  - Analogous to conditional expression
- Requirements for secure:
  - All statements in each basic blocks are secure
  - lub{ $\underline{x}_{i1}, \ldots, \underline{x}_{in}$ } ≤ glb{ $\underline{y}$ | y target of assignment in B_i }

Example of Requirements

- Within each basic block:

  b_1: $Low \le \underline{i}$ b_3: $Low \le \underline{j}$ b_6: lub{ Low, $\underline{i}$ } $\le \underline{i}$

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

  - Combining, lub{ $\underline{x[i][j]}$, $\underline{i}$, $\underline{j}$ } $\le \underline{y[j][i]}$ }
  - From declarations, true when lub{ $\underline{x}$, $\underline{i}$ } $\le \underline{y}$

- $B_2 = \{b_3, b_4, b_5, b_6\}$
  - Assignments to i, j, $y[j][i]$; conditional is $i \le 10$
  - Requires $\underline{i} \le$ glb{ $\underline{i}$, $\underline{j}$, $\underline{y[j][i]}$ }
  - From declarations, true when $\underline{i} \le \underline{y}$

Example (continued)

- $B_4 = \{ b_5 \}$
  - Assignments to j, $y[j][i]$; conditional is $j \leq 10$
  - Requires $\underline{j} \leq \text{glb}\{ \underline{j}, \underline{y[j][i]} \}$
  - From declarations, means $\underline{i} \leq \underline{y}$
- Result:
  - Combine $\text{lub}\{ \underline{x}, \underline{i} \} \leq \underline{y}$; $\underline{i} \leq \underline{y}$; $\underline{i} \leq \underline{y}$
  - Requirement is $\text{lub}\{ \underline{x}, \underline{i} \} \leq \underline{y}$

Procedure Calls

`tm(a, b);`

From previous slides, to be secure, lub{ $\underline{x}$, $\underline{i}$ } ≤ $\underline{y}$ must hold

- In call, *x* corresponds to *a*, *y* to *b*
- Means that lub{ $\underline{a}$, $\underline{i}$ } ≤ $\underline{b}$, or $\underline{a} \le \underline{b}$

More generally:

proc *pn*(i_1, …, i_m: **int**; **var** o_1, …, o_n: **int**); **begin** *S* **end**;

- *S* must be secure
- For all *j* and *k*, if $\underline{i}_j \le \underline{o}_k$, then $\underline{x}_j \le \underline{y}_k$
- For all *j* and *k*, if $\underline{o}_j \le \underline{o}_k$, then $\underline{y}_j \le \underline{y}_k$

Exceptions

```
proc copy(x: integer class { x };
                    var y: integer class Low);
var sum: integer class { x };
    z: int class Low;
begin
     y := z := sum := 0;
     while z = 0 do begin
          sum := sum + x;
          y := y + 1;
     end
end
```

Exceptions (*cont*)

- When sum overflows, integer overflow trap
  - Procedure exits
  - Value of *x* is MAXINT/*y*
  - Information flows from *y* to *x*, but $\underline{x} \leq \underline{y}$ never checked
- Need to handle exceptions explicitly
  - Idea: on integer overflow, terminate loop

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

  - Now information flows from *sum* to *z*, meaning $\underline{sum} \leq \underline{z}$
  - This is false ($\underline{sum}$ = { x } dominates $\underline{z}$ = Low)