Lecture 21
November 15, 2024

Basics of Information Flow

- Bell-LaPadula Model embodies information flow policy
  - Given compartments *A*, *B*, info can flow from *A* to *B* iff *B dom A*
- So does Biba Model
  - Given compartments *A*, *B*, info can flow from *A* to *B* iff *A dom B*
- Variables *x*, *y* assigned compartments $\underline{x}$, $\underline{y}$ as well as values
  - Confidentiality (Bel-LaPadula): if $\underline{x}$ = *A*, $\underline{y}$ = *B*, and *B dom A*, then *y* := *x* allowed but not *x* := *y*
  - Integrity (Biba): if $\underline{x}$ = *A*, $\underline{y}$ = *B*, and *A dom B*, then *x* := *y* allowed but not *y* := *x*
- For now, focus on confidentiality (Bell-LaPadula)
  - We'll get to integrity later

Entropy and Information Flow

- Idea: information flows from *x* to *y* as a result of a sequence of commands *c* if you can deduce information about *x* before *c* from the value in *y* after *c*
- Formally:
  - *s* time before execution of *c*, *t* time after
  - $H(x_s \mid y_t) < H(x_s \mid y_s)$
  - If no *y* at time *s*, then $H(x_s \mid y_t) < H(x_s)$

Example 1

- Command is $x := y + z$; where:
  - x does not exist initially (that is, has no value)
  - $0 \le y \le 7$, equal probability
  - $z = 1$ with probability 1/2, $z = 2$ or 3 with probability 1/4 each
- s state before command executed; t, after; so
  - $H(y_s) = H(y_t) = -8(1/8) \lg (1/8) = 3$
- You can show that $H(y_s \mid x_t) = (3/32) \lg 3 + 9/8 \approx 1.274 < 3 = H(y_s)$
  - Thus, information flows from y to x

Example 2

- Command is

  $$\textbf{if } x = 1 \textbf{ then } y := 0 \textbf{ else } y := 1;$$

  where x, y equally likely to be either 0 or 1
- $H(x_s) = 1$ as x can be either 0 or 1 with equal probability
- $H(x_s \mid y_t) = 0$ as if $y_t = 1$ then $x_s = 0$ and vice versa
  - Thus, $H(x_s \mid y_t) = 0 < 1 = H(x_s)$
- So information flowed from x to y

Implicit Flow of Information

- Information flows from x to y without an *explicit* assignment of the form $y := f(x)$
  - $f(x)$ an arithmetic expression with variable x
- Example from previous slide:

if $x = 1$ **then** $y := 0$ **else** $y := 1$;

- So must look for implicit flows of information to analyze program

Notation

- $\underline{x}$ means class of *x*
  - In Bell-LaPadula based system, same as "label of security compartment to which *x* belongs"
- $\underline{x} \le \underline{y}$ means "information can flow from an element in class of *x* to an element in class of *y*
  - Or, "information with a label placing it in class $\underline{x}$ can flow into class $\underline{y}$"

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:

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

  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

$$i_p\texttt{: } \textbf{\textit{type}}\ \textbf{class}\ \{\ \mathrm{i}_\mathrm{p}\ \}$$

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:

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

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:

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

- 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(x1, …, xn) then S1 else S2; 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{ $\underline{x}_1$, …, $\underline{x}_n$ } ≤ glb{$\underline{y}$ | *y* target of assignment in *S* }

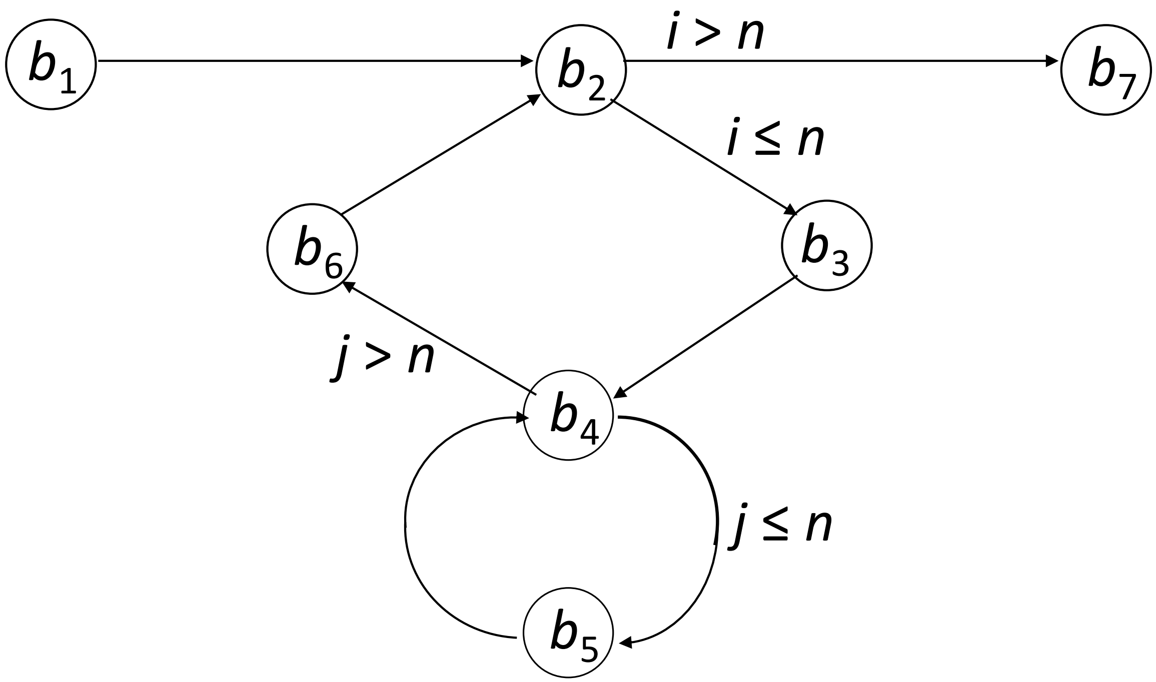
Goto Statements

- 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



Immediate Forward Dominators

- 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*

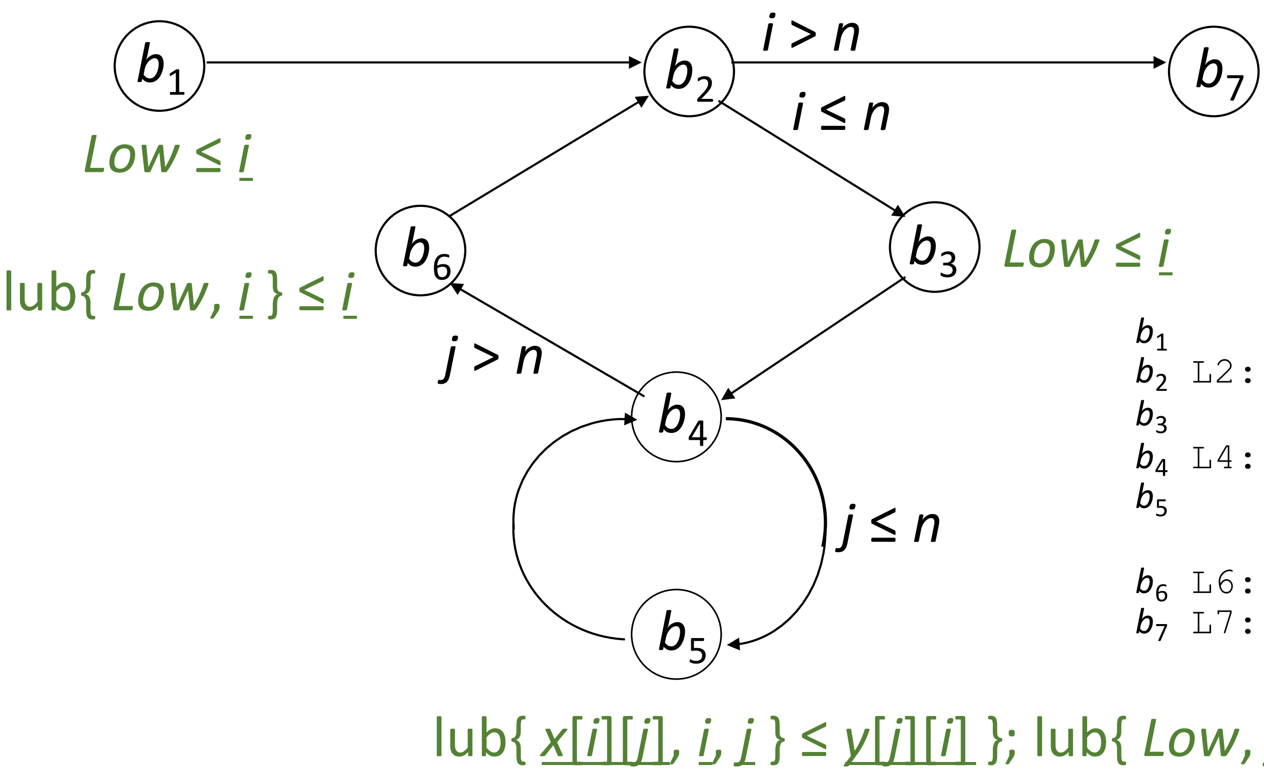
IFD Example

- 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}, …, 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{ $\underline{x}_{i1}$, …, $\underline{x}_{in}$ } ≤ glb{ $\underline{y}$ | y target of assignment in B_i }

Example of Requirements




```
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:
```

Example of Requirements

- Within each basic block:

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

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

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

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 *sum* is MAXINT/*y*
  - Information flows from *y* to *sum*, but $\underline{sum} \leq \underline{y}$ 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)

Infinite Loops

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

- If *x* = 0 initially, infinite loop
- If *x* = 1 initially, terminates with *y* set to 1
- No explicit flows, but implicit flow from *x* to *y*

Semaphores

Use these constructs:

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

- *x* is semaphore, a shared variable
- Both executed atomically

Consider statement

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

- Implicit flow from *sem* to *x*
  - Certification must take this into account!

Flow Requirements

- Semaphores in *signal* irrelevant
  - Don't affect information flow in that process
- Statement S is a *wait*
  - shared(S): set of shared variables read
    - Idea: information flows out of variables in shared(S)
  - fglb(S): glb of assignment targets *following S*
  - So, requirement is shared(S) ≤ fglb(S)
- begin S_1; … S_n end
  - All S_i must be secure
  - For all i, shared(S_i) ≤ fglb(S_i)

Example

```
begin
    x := y + z;          (* S1 *)
    wait(sem);           (* S2 *)
    a := b * c - x;      (* S3 *)
end
```

- Requirements:
  - lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
  - lub{ $\underline{b}$, $\underline{c}$, $\underline{x}$ } ≤ $\underline{a}$
  - $\underline{sem}$ ≤ $\underline{a}$
    - Because fglb(S_2) = $\underline{a}$ and shared(S_2) = *sem*

Concurrent Loops

- Similar, but wait in loop affects *all* statements in loop
  - Because if flow of control loops, statements in loop before wait may be executed after wait
- Requirements
  - Loop terminates
  - All statements $S_1, \ldots, S_n$ in loop secure
  - lub{ $\underline{\text{shared}(S_1)}, \ldots, \underline{\text{shared}(S_n)}$ } ≤ glb($t_1, \ldots, t_m$)
    - Where $t_1, \ldots, t_m$ are variables assigned to in loop

Loop Example

```
while i < n do begin
    a[i] := item;          (* S1 *)
    wait(sem);             (* S2 *)
    i := i + 1;            (* S3 *)
end
```

- Conditions for this to be secure:
  - Loop terminates, so this condition met
  - S_1 secure if lub{ $\underline{i}$, $\underline{item}$ } ≤ $\underline{a[i]}$
  - S_2 secure if $\underline{sem}$ ≤ $\underline{i}$ and $\underline{sem}$ ≤ $\underline{a[i]}$
  - S_3 trivially secure

cobegin/coend

```
cobegin
        x := y + z;          (* S1 *)
        a := b * c - y;      (* S2 *)
coend
```

- No information flow among statements
  - For S_1, lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
  - For S_2, lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } ≤ $\underline{a}$
- Security requirement is both must hold
  - So this is secure if lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$ ∧ lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } ≤ $\underline{a}$

Soundness

- Above exposition intuitive
- Can be made rigorous:
  - Express flows as types
  - Equate certification to correct use of types
  - Checking for valid information flows same as checking types conform to semantics imposed by security policy