Lecture 24
November 17, 2021

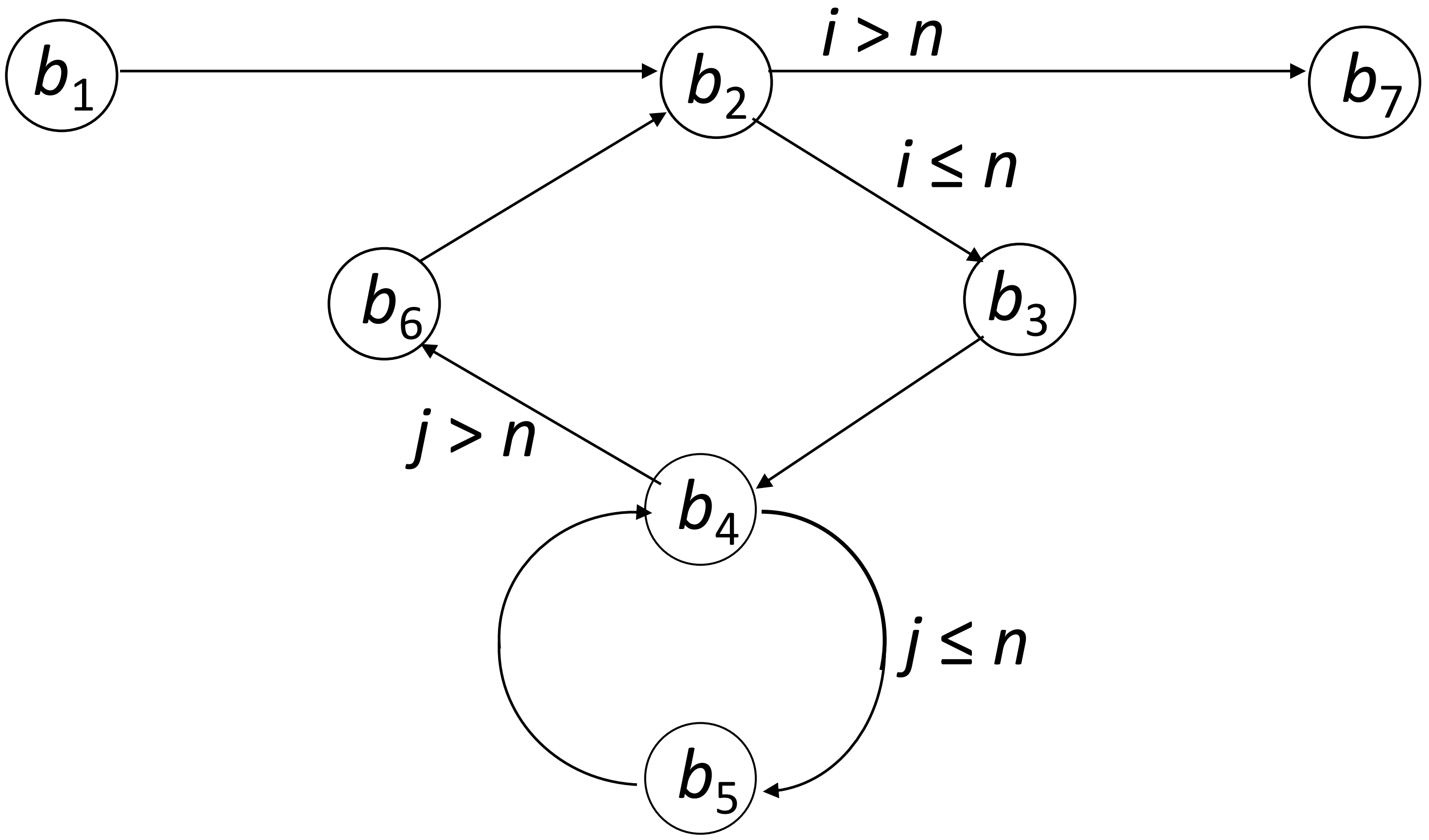
Goto Statements

- 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
$b_1$      i := 1;
$b_2$ L2:  if i > 10 goto L7;
$b_3$      j := 1;
$b_4$ L4:  if j > 10 then goto L6;
$b_5$      y[j][i] := x[i][j]; j := j + 1; goto L4;
$b_6$ L6:  i := i + 1; goto L2;
$b_7$ 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*

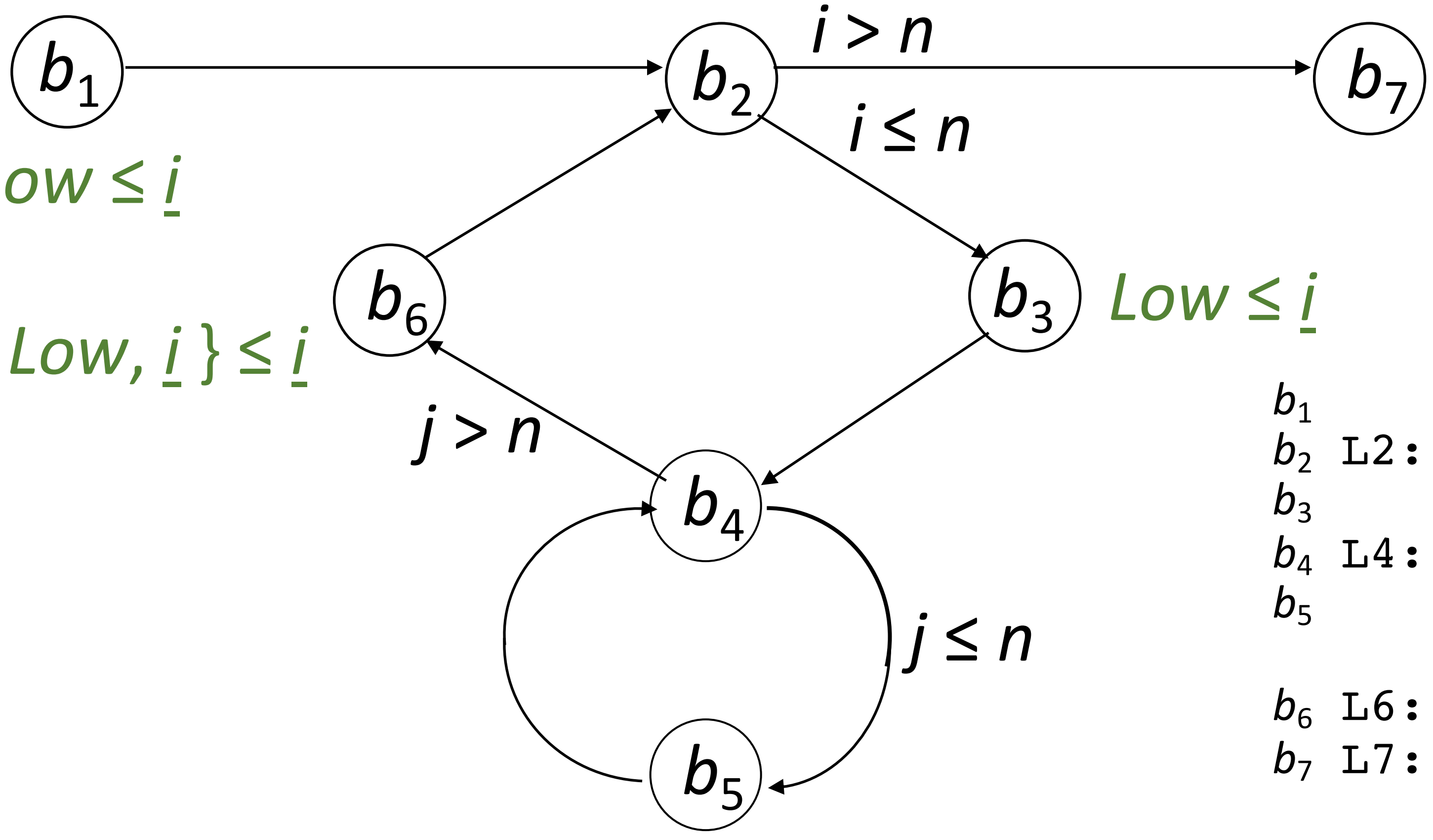
IFD Example

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



$Low \le \underline{i}$

$Low \le \underline{i}$

$\text{lub}\{\ Low, \underline{i}\ \} \le \underline{i}$

```
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:
```

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

Example of Requirements

- Within each basic block:

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

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

  - 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 $j \le 10$
  - Requires $\underline{j} \le \text{glb}\{ \underline{j}, \underline{y[j][i]} \}$
  - From declarations, means $\underline{i} \le \underline{y}$
- Result:
  - Combine $\text{lub}\{ \underline{x}, \underline{i} \} \le \underline{y}$; $\underline{i} \le \underline{y}$; $\underline{i} \le \underline{y}$
  - Requirement is $\text{lub}\{ \underline{x}, \underline{i} \} \le \underline{y}$

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 $\underline{a} \leq \underline{b}$

More generally:

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

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

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 $\underline{sum} \le \underline{y}$ 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 $\underline{sum} \le \underline{z}$
  - 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;         (* $S_1$ *)
    wait(sem);          (* $S_2$ *)
    a := b * c – x;     (* $S_3$ *)
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
  - $\text{lub}\{ \underline{\text{shared}(S_1)}, \ldots, \underline{\text{shared}(S_n)} \} \le \text{glb}(t_1, \ldots, t_m)$
    - Where $t_1, \ldots, t_m$ are variables assigned to in loop

Loop Example

```
while i < n do begin
    a[i] := item;       (* $S_1$ *)
    wait(sem);          (* $S_2$ *)
    i := i + 1;         (* $S_3$ *)
end
```

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

cobegin/coend

```
cobegin
        x := y + z;         (* $S_1$ *)
        a := b * c – y;     (* $S_2$ *)
coend
```

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

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

Execution-Based Mechanisms

- Detect and stop flows of information that violate policy
  - Done at run time, not compile time
- Obvious approach: check explicit flows
  - Problem: assume for security, $\underline{x} \leq \underline{y}$

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

  - When $x \neq 1$, $\underline{x}$ = High, $\underline{y}$ = Low, $\underline{a}$ = Low, appears okay—but implicit flow violates condition!

Fenton's Data Mark Machine

- Each variable has an associated class
- Program counter (PC) has one too
- Idea: branches are assignments to PC, so you can treat implicit flows as explicit flows
- Stack-based machine, so everything done in terms of pushing onto and popping from a program stack

Instruction Description

- *`skip`*: instruction not executed
- *`push(x, x)`*: push variable *x* and its security class *x* onto program stack
- *`pop(x, x)`* : pop top value and security class from program stack, assign them to variable *x* and its security class *x* respectively

Instructions

- *x* **:=** *x* + 1 (increment)
  - Same as:

    ```
    if $\underline{PC}$ ≤ $\underline{x}$ then x := x + 1 else skip
    ```

- **if** *x* = 0 **then goto** *n* **else** *x* **:=** *x* – 1 (branch and save PC on stack)
  - Same as:

    ```
    if x = 0 then begin
      push(PC, $\underline{PC}$); $\underline{PC}$ := lub{$\underline{PC}$, x}; PC := n;
     end else if $\underline{PC}$ ≤ $\underline{x}$ then
      x := x - 1
    else
      skip;
    ```

More Instructions

- **if′** x = 0 **then goto** n **else** x := x – 1 (branch without saving PC on stack)
  - Same as:

```
if $x$ = 0 then
  if $\underline{x} \le \underline{PC}$ then $PC$ := $n$ else $skip$
else
  if $\underline{PC} \le \underline{x}$ then $x$ := $x$ - 1 else $skip$
```

More Instructions

- **`return`** (go to just after last *if*)
  - Same as:

    pop(*PC*, <u>*PC*</u>);
- **`halt`** (stop)
  - Same as:

    if *program stack empty* **then** *halt*
  - Note stack empty to prevent user obtaining information from it after halting