Information Flow

- Overview
  - Basics and background
  - Compiler-based mechanisms
  - Execution-based mechanisms

Basics

- Bell-LaPadula Model embodies information flow policy
  - Given compartments A, B, info can flow from A to B iff B *dom* A
- Variables x, y assigned compartments $\underline{x}$, $\underline{y}$ as well as values
  - If $\underline{x} = A$ and $\underline{y}$ = B, and B *dom* A, then the assignment $y := x$ is allowed but $x := y$ is not

Information Flow

- Idea: info flows from x to y as a result of a sequence of commands c if you can deduce **information** about x (as simple as excluding possible values) before c from the value in y after c

Example 1

- Command is `x := y + z;` where:
  - $0 \le y \le 7$, equal probability
  - $z = 1$ with prob. 1/2, $z = 2$ or 3 with prob. 1/4 each
- If you know final value of x, initial value of y can have at most 3 values, so information flows from y to x

Example 2

- Command is
  - **if** *x* = 1 **then** *y* := 0 **else** *y* := 1;

  where:
  - *x*, *y* equally likely to be either 0 or 1
- But if *x* = 1 then *y* = 0, and vice versa, so value of *y* depends on *x*
- 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*)
- 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"
- $\underline{x} \leq \underline{y}$ 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 $\underline{y}$"

7

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 the flows in set of statements do not violate that policy

8

Example

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

- Info 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

9

Declarations

- Notation:

```
x: int class { A, B }
```

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

10

Input Parameters

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

$$i_p\text{: } \textbf{\textit{type}}\ \textbf{class}\ \{\ \mathtt{i}_p\ \}$$

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 ith input or input/output argument from which info. flows into output o_p

Example

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

- Require $\underline{x} \leq \underline{out}$ and $\underline{out} \leq \underline{out}$

Array Elements

- Information flowing out:

  `… := a[i]`

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

  `a[i] := …`
- Only value of a[i] affected, so class is a[i]

Assignment Statements

$x := y + z;$

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

More generally:

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

- Information flow from the input values to the result, so lub{ $\underline{x}_1$, …, x_n } ≤ $\underline{y}$ must hold

15

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:

$S_1; \ldots S_n;$

- Each individual S_i must be secure

16

Conditional Statements

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

- The 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 or S_2 }

17

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 }

18

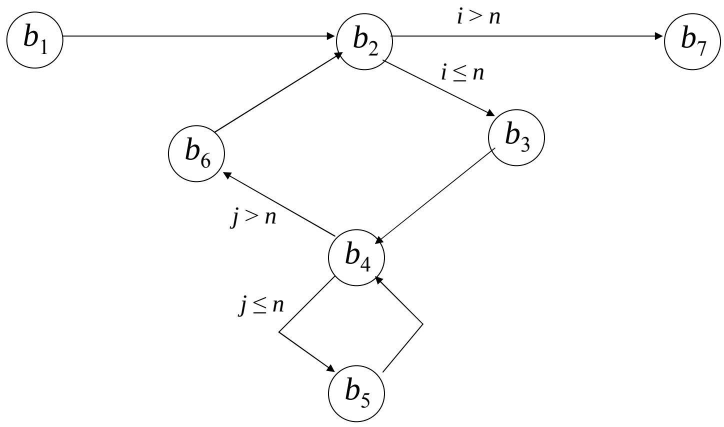
Goto Statements

- No assignments
  - Hence no explicit flows
- Need to detect implicit flows
- *Basic block* is a 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 int class {x};
    var y: array[1..10][1..10] of int class {y});
var i, j: int 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 the 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 the 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{ $\underline{x}_{i1}, \ldots, \underline{x}_{in}$ } $\leq$ glb{ $\underline{y}$ | y target of assignment in B_i }

Example of Requirements

- Within each basic block:

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

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

25

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 $\text{lub}\{ \underline{x}, \underline{i} \} \leq \underline{y}$

26

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 $\underline{y_j} \le \underline{y_k}$

27

Soundness

- Above exposition intuitive
- Can be (has been 1996) 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

28

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!

Key Points

- Both amount of information, direction of flow important
  - Flows can be explicit or implicit
- Compiler-based checks flows at compile time
- Execution-based checks flows at run time