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 {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

IFD Example

- In previous procedure:
  - IFD($b_1$) = $b_2$ one path
  - IFD($b_2$) = $b_7$ $b_2$→$b_7$ or $b_2$→$b_3$→$b_6$→$b_2$→$b_7$
  - IFD($b_3$) = $b_4$ one path
  - IFD($b_4$) = $b_6$ $b_4$→$b_6$ or $b_4$→$b_5$→$b_6$
  - IFD($b_5$) = $b_4$ one path
  - IFD($b_6$) = $b_2$ one path
Example of Requirements

- Within each basic block:
  \( b_1: \text{Low} \leq i \) \( b_5: \text{Low} \leq j \) \( b_6: \text{lub}\{ \text{Low}, i \} \leq i \)
  \( b_5: \text{lub}\{ x[j][i], i, j \} \leq y[j][i] \}; \text{lub}\{ \text{Low}, i \} \leq i \)
  - Combining, \( \text{lub}\{ x[j][i], i, j \} \leq y[j][i] \}
  - From declarations, true when \( \text{lub}\{ x, i \} \leq y \)

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

Example (continued)

- \( B_4 = \{ b_5 \} \)
  - Assignments to \( j, y[j][i] \); conditional is \( j \leq 10 \)
  - Requires \( j \leq \text{glb}\{ j, y[j][i] \} \)
  - From declarations, means \( i \leq y \)

- Result:
  - Combine \( \text{lub}\{ x, i \} \leq y; i \leq y; i \leq y \)
  - Requirement is \( \text{lub}\{ x, i \} \leq y \)
Procedure Calls

\[ tm(a, b); \]

From previous slides, to be secure, lub\{ x, i \} \leq y must hold
- In call, \( x \) corresponds to \( a, y \) to \( b \)
- Means that \( \text{lub}(a, i) \leq b \), or \( a \leq b \)

More generally:
\[
\text{proc } p\!n(i_1, \ldots, i_m: \text{int}; \text{var } o_1, \ldots, o_n: \text{int})
\begin{align*}
\text{begin} & \\
\text{S must be secure} & \\
\text{For all } j \text{ and } k, \text{ if } i_j \leq o_k, \text{ then } x_j \leq y_k & \\
\text{For all } j \text{ and } k, \text{ if } o_j \leq o_k, \text{ then } y_j \leq y_k & \\
\text{end}
\end{align*}
\]

Exceptions

\[
\text{proc } \text{copy}(x: \text{int class } \{ x \};
\quad \text{var } y: \text{int class Low})
\]
\[
\text{var } \text{sum}: \text{int class } \{ x \};
\quad \text{z: int class Low};
\begin{align*}
\text{begin} & \\
y := z := \text{sum} := 0; & \\
\text{while } z = 0 \text{ do begin} & \\
\text{sum} := \text{sum} + x; & \\
y := y + 1; & \\
\text{end} & \\
\text{end}
\end{align*}
\]
Exceptions (cont)

- When sum overflows, integer overflow trap
  - Procedure exits
  - Value of $x$ is MAXINT/y
  - Info flows from $y$ to $x$, but $x \leq y$ never checked
- Need to handle exceptions explicitly
  - Idea: on integer overflow, terminate loop
    
    ```
    on integer_overflow_exception sum do z := 1;
    ```
  - Now info flows from $sum$ to $z$, meaning $sum \leq z$
  - This is false ($sum = \{ x \}$ dominates $z = Low$)

Infinite Loops

```plaintext
proc copy(x: int 0..1 class { x });
    var y: int 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:

\[
\text{wait}(x): \quad \text{if } x = 0 \text{ then block until } x > 0; \quad x := x - 1;
\]

\[
\text{signal}(x): \quad x := x + 1;
\]

– x is semaphore, a shared variable
– Both executed atomically

Consider statement

\[
\text{wait}(\text{sem}); \quad x := x + 1;
\]

• Implicit flow from \text{sem} to \text{x}
  – Certification must take this into account!

Flow Requirements

• Semaphores in \text{signal} irrelevant
  – Don’t affect information flow in that process

• Statement \text{S} is a wait
  – \text{shared}(\text{S}): \text{set of shared variables read}
    • Idea: information flows out of variables in \text{shared}(\text{S})
  – \text{fglb}(\text{S}): glb of assignment targets following \text{S}
  – So, requirement is \text{shared}(\text{S}) \leq \text{fglb}(\text{S})

• \text{begin} \text{S}_1; \ldots \text{S}_n \text{ end}
  – All \text{S}_i must be secure
  – For all \text{i}, \text{shared}(\text{S}_i) \leq \text{fglb}(\text{S}_i)
Example

begin
    \begin{align*}
    x &:= y + z; & \quad (* S_1 *) \\
    \text{wait}(\text{sem}); & \quad (* S_2 *) \\
    a &:= b * c - x; & \quad (* S_3 *) \\
    \end{align*}
end

• Requirements:
  - \text{lub}\{y, z\} \leq x
  - \text{lub}\{b, c, x\} \leq a
  - \text{sem} \leq a
    • Because \text{fglb}(S_2) = a \text{ and } \text{shared}(S_2) = \text{sem}

Concurrent Loops

• Similar, but wait in loop affects \textit{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}\{ \text{shared}(S_1), \ldots, \text{shared}(S_n)\} \leq \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

    $a[i] := \text{item}$; (* $S_1$ *)

    \text{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 lub{$i, \text{item}$} $\leq a[i]$
  – $S_2$ secure if sem $\leq i$ and sem $\leq a[i]$
  – $S_3$ trivially secure

\begin{verbatim}
May 12, 2006 ECS 289M, Foundations of Computer and Information Security Slide 15
\end{verbatim}

cobegin/coend
cobegin

    $x := y + z$; (* $S_1$ *)

    $a := b \ast c - y$; (* $S_2$ *)

coend

• No information flow among statements
  – For $S_1$, lub{$y, z$} $\leq x$
  – For $S_2$, lub{$b, c, y$} $\leq a$

• Security requirement is both must hold
  – So this is secure if lub{$y, z$} $\leq x \land$ lub{$b, c, y$} $\leq a$

\begin{verbatim}
May 12, 2006 ECS 289M, Foundations of Computer and Information Security Slide 16
\end{verbatim}
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, \( x \leq y \)
    \[
    \text{if } x = 1 \text{ then } y := a;
    \]
  - When \( x \neq 1, x = \text{High}, y = \text{Low}, a = \text{Low}, \) 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* means instruction not executed
• *push(x, x)* means push variable x and its security class x onto program stack
• *pop(x, x)* means 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 $PC \leq x$ 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($PC, PC$); $PC := \text{lub}(PC, x)$; PC := $n$;
    end else if $PC \leq x$ 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 $x \leq PC$ then $PC := n$ else skip
    else
        if $PC \leq x$ then $x := x - 1$ else skip
More Instructions

- **return** (go to just after last *if*)
  - Same as:
    - \( \text{pop}(PC, PC); \)
- **halt**(stop)
  - Same as:
    - \( \text{if } \text{program stack empty then halt} \)
  - Note stack empty to prevent user obtaining information from it after halting

### Example Program

1. \( \text{if } x = 0 \text{ then goto 4 else } x := x - 1 \)
2. \( \text{if } z = 0 \text{ then goto 6 else } z := z - 1 \)
3. \( \text{halt} \)
4. \( z := z - 1 \)
5. \( \text{return} \)
6. \( y := y - 1 \)
7. \( \text{return} \)

- Initially \( x = 0 \) or \( x = 1 \), \( y = 0 \), \( z = 0 \)
- Program copies value of \( x \) to \( y \)
Example Execution

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>PC</th>
<th>PC</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td>Low ≤ x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>z</td>
<td>(3, Low)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>z</td>
<td>(3, Low)</td>
<td>PC ≤ y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Handling Errors

• Ignore statement that causes error, but continue execution
  – If aborted or a visible exception taken, user could deduce information
  – Means errors cannot be reported unless user has clearance at least equal to that of the information causing the error
Variable Classes

• Up to now, classes fixed
  – Check relationships on assignment, etc.
• Consider variable classes
  – Fenton’s Data Mark Machine does this for \( PC \)
  – On assignment of form \( y := f(x_1, \ldots, x_n), y \) changed to \( \text{lub}\{ x_1, \ldots, x_n \} \)
  – Need to consider implicit flows, also

Example Program

(* Copy value from \( x \) to \( y \)
  * Initially, \( x \) is 0 or 1 *)
proc copy(x: int class { x });
    var y: int class { y }
    var z: int class variable { Low }; begin
    y := 0;
    z := 0;
    if x = 0 then z := 1;
    if z = 0 then y := 1;
end;

• \( z \) changes when \( z \) assigned to
• Assume \( y < x \)
Analysis of Example

- $x = 0$
  - $z := 0$ sets $z$ to Low
  - if $x = 0$ then $z := 1$ sets $z$ to 1 and $z$ to $x$
    - So on exit, $y = 0$
- $x = 1$
  - $z := 0$ sets $z$ to Low
  - if $z = 0$ then $y := 1$ sets $y$ to 1 and checks that $\text{lub}(\text{Low}, z) \leq y$
    - So on exit, $y = 1$
- Information flowed from $x$ to $y$ even though $y < x$

Handling This (1)

- Fenton’s Data Mark Machine detects implicit flows violating certification rules