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

```pascal
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;
```

November 17, 2021
ECS 235A, Computer and Information Security
Flow of Control

\[ b_1 \rightarrow b_2 \quad i > n \rightarrow b_7 \]
\[ b_2 \rightarrow b_6 \quad i \leq n \rightarrow b_3 \]
\[ b_6 \rightarrow b_4 \quad j > n \rightarrow b_3 \]
\[ b_4 \rightarrow b_5 \quad j \leq n \rightarrow b_4 \]
\[ b_5 \rightarrow b_1 \]
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 $\text{IFD}(b)$) is first basic block lying on all paths of execution passing through $b$
IFD Example

• In previous procedure:
  • $\text{IFD}(b_1) = b_2$ one path
  • $\text{IFD}(b_2) = b_7$ $b_2 \rightarrow b_7$ or $b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_7$
  • $\text{IFD}(b_3) = b_4$ one path
  • $\text{IFD}(b_4) = b_6$ $b_4 \rightarrow b_6$ or $b_4 \rightarrow b_5 \rightarrow b_6$
  • $\text{IFD}(b_5) = b_4$ one path
  • $\text{IFD}(b_6) = b_2$ one path
Requirements

• $B_i$ is set of basic blocks along an execution path from $b_i$ to $\text{IFD}(b_i)$
  • 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
  • $\text{lub}\{x_{i1}, ..., x_{in}\} \leq \text{glb}\{y \mid y \text{ target of assignment in } B_i\}$
Example of Requirements

\[ \text{lub}\{ \text{Low, } i \} \leq i \]

\[ \text{lub}\{ \text{x[i][j], } i, j \} \leq \text{y[j][i]} \}; \text{lub}\{ \text{Low, } j \} \leq j \]

\[ \begin{align*}
& b_1 & \quad i := 1; \\
& b_2 & \quad \text{if } i > 10 \text{ goto L7;} \\
& b_3 & \quad j := 1; \\
& b_4 & \quad \text{if } j > 10 \text{ then goto L6;} \\
& b_5 & \quad y[j][i] := x[i][j]; \\
& b_6 & \quad j := j + 1; \text{ goto L4;} \\
& b_7 & \quad i := i + 1; \text{ goto L2;} \\
& \end{align*} \]
Example of Requirements

• Within each basic block:
  
  \[b_1: \text{Low} \leq i \quad b_3: \text{Low} \leq j \quad b_6: \text{lub}\{ \text{Low}, i \} \leq i\]
  
  \[b_5: \text{lub}\{ x[i][j], i, j \} \leq y[j][i] \}; \text{lub}\{ \text{Low}, i \} \leq i\]
  
  • Combining, \text{lub}\{ x[i][j], 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 \(i \leq \text{glb}\{ i, j, y[j][i] \}\)
  
  • 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 lub\{a, i\} \leq b, or a \leq 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 i_j \leq o_k, then x_j \leq y_k
• For all j and k, if o_j \leq o_k, then y_j \leq y_k
Exceptions

```plaintext
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 sum ≤ 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 sum ≤ z
• This is false (sum = { x } dominates 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:

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

\texttt{signal}(x): \quad x := x + 1;

- \( x \) is semaphore, a shared variable
- Both executed atomically

Consider statement

\texttt{wait}(sem); \quad x := x + 1;

- Implicit flow from \textit{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*
  • $\text{shared}(S)$: set of shared variables read
    • Idea: information flows out of variables in $\text{shared}(S)$
  • $\text{fglb}(S)$: glb of assignment targets *following* $S$
  • So, requirement is $\text{shared}(S) \leq \text{fglb}(S)$

• begin $S_1$; ... $S_n$ end
  • All $S_i$ must be secure
  • For all $i$, $\text{shared}(S_i) \leq \text{fglb}(S_i)$
Example

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

• Requirements:
  • \( \text{lub}\{ y, z \} \leq x \)
  • \( \text{lub}\{ b, c, x \} \leq a \)
  • \( \overline{sem} \leq a \)
    • Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = sem \)
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 $S_1, ..., S_n$ in loop secure
  • $\text{lub}\{\text{shared}(S_1), ..., \text{shared}(S_n)\} \leq \text{glb}(t_1, ..., t_m)$
    • Where $t_1, ..., t_m$ are variables assigned to in loop
Loop Example

```plaintext
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 $\text{lub}\{i, \text{item}\} \leq a[i]$
  • $S_2$ secure if $\text{sem} \leq i$ and $\text{sem} \leq a[i]$
  • $S_3$ trivially secure
cobegin/coend

cobegin

\[ x := y + z; \quad (* S_1 *) \]
\[ a := b \times c - y; \quad (* S_2 *) \]

coend

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

• Security requirement is both must hold
  • So this is secure if \( \text{lub}\{ y, z \} \leq x \land \text{lub}\{ b, c, y \} \leq a \)
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, $x \leq y$
    
    ```
    if x = 1 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*: 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:
    \[
    \text{if } PC \leq x \text{ then } x := x + 1 \text{ else } \text{skip}
    \]

• if \( x = 0 \) then goto \( n \) else \( x := x - 1 \) (branch and save PC on stack)
  • Same as:
    \[
    \text{if } x = 0 \text{ then begin}
    \hspace{1em} \text{push}(PC, PC);
    \hspace{1em} PC := \text{lub}(PC, x); \hspace{1em} PC := n;
    \hspace{1em} \text{end else if } PC \leq x \text{ then}
    \hspace{1em} x := x - 1
    \hspace{1em} \text{else}
    \hspace{2em} \text{skip;}
    \]
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:
    ```
    pop(PC, PC);
    ```
- **halt** (stop)
  - Same as:
    ```
    if program stack empty then halt
    ```
  - Note stack empty to prevent user obtaining information from it after halting