May 12: Information Flow

- Static (compile-time) mechanisms
- Dynamic (run-time) mechanisms
Array Elements

• Information flowing out:
  \[ \ldots := a[i] \]
  Value of \( i \), \( a[i] \) both affect result, so class is \( \text{lub}\{ a[i], i \} \)

• Information flowing in:
  \[ a[i] := \ldots \]

• 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 \( lub(y, z) \leq x \)

More generally:

\[ y := f(x_1, \ldots, x_n) \]

- the relation \( lub(x_1, \ldots, x_n) \leq y \) must hold
Compound Statements

\[ x := y + z; \quad a := b \times c - x; \]

- First statement: \( \text{lub}(y, z) \leq x \)
- Second statement: \( \text{lub}(b, c, x) \leq a \)
- So, both must hold (i.e., be secure)

More generally:

\[ S_1; \ldots; S_n; \]

- Each individual \( S_i \) must be secure
Conditional Statements

\[
\text{if } x + y < z \text{ then } a := b \text{ else } d := b \times c - x;
\]

- The statement executed reveals information about \(x, y, z\), so \(\text{lub}(x, y, z) \leq \text{glb}(a, d)\)

More generally:

\[
\text{if } f(x_1, \ldots, x_n) \text{ then } S_1 \text{ else } S_2; \text{ end}
\]

- \(S_1, S_2\) must be secure
- \(\text{lub}(x_1, \ldots, x_n) \leq \text{glb}(y | y \text{ target of assignment in } S_1, S_2)\)
Iterative Statements

\begin{code}
\textbf{while } i < n \textbf{ do begin }
\begin{align*}
a[i] &:= b[i]; \\
i &:= i + 1;
\end{align*}
\textbf{end}
\end{code}

\begin{itemize}
\item Same ideas as for “if”, but must terminate
\end{itemize}

More generally:
\begin{code}
\textbf{while } f(x_1, \ldots, x_n) \textbf{ do } S;
\end{code}

\begin{itemize}
\item Loop must terminate;
\item S must be secure
\end{itemize}

\begin{align*}
lub(x_1, \ldots, x_n) &\leq \\
glb(y | y \text{ target of assignment in } S)
\end{align*}
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

\begin{verbatim}
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
    i := 1;
    L2: if i > 10 then goto L7;
    j := 1;
    L4: if j > 10 then goto L6;
    y[j][i] := x[i][j]; j := j + 1; goto L4;
    i := i + 1; goto L2;
end;
\end{verbatim}
Flow of Control

\[ b_1 \xrightarrow{i > n} b_2 \xrightarrow{i \leq n} b_3 \xrightarrow{j > n} b_4 \xrightarrow{j \leq n} b_5 \xrightarrow{} b_6 \xrightarrow{} b_7 \]
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 a basic block $b$ (written $\text{IFD}(b)$) is the 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 the set of basic blocks along an execution path from $b_i$ to $\text{IFD}(b_i)$
  - 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 being secure:
  - All statements in each basic blocks are secure
  - $\text{lub}(x_{i1}, \ldots, x_{in}) \leq \text{glb}\{ y \mid y \text{ target of assignment in } B_i \}$
Example of Requirements

- Within each basic block:
  \( b_1: \text{Low} \leq i \)  
  \( b_3: \text{Low} \leq j \)  
  \( b_6: \text{lub}\{ \text{Low}, i \} \leq i \)  
  \( b_5: \text{lub}(x[i][j], i, j) \leq y[j][i]; \text{lub}(\text{Low}, j) \leq j \)  
  - 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:

\[
\text{proc } pn(i_1, \ldots, i_m: \text{int}; \text{var } o_1, \ldots, o_n: \text{int})
\begin{align*}
\text{begin} & \quad S \\
\text{end} & \end{align*}
\]

- \( S \) 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: int class { x };
    var y: int class Low)

var sum: int 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 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 $\text{sum} \leq z$
  - This is false ($\text{sum} = \{ x \}$ dominates $z = \text{Low}$)
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

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

- 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
  – $\text{shared}(S)$: 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 $S_1; \ldots; S_n$ 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:
  – \( \text{lub}(y, z) \leq x \)
  – \( \text{lub}(b, c, x) \leq a \)
  – \( \text{sem} \leq a \)
  • Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = \text{sem} \)
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
  – $\operatorname{lub}(\text{shared}(S_1), \ldots, \text{shared}(S_n)) \leq \operatorname{glb}(t_1, \ldots, t_m)$
    • Where $t_1, \ldots, t_m$ are variables assigned to in loop
Loop Example

\[
\text{while } i < n \text{ do begin } \\
\hspace{1cm} a[i] := \text{item}; \quad (* S_1 *) \\
\hspace{1cm} \text{wait}(sem); \quad (* S_2 *) \\
\hspace{1cm} i := i + 1; \quad (* S_3 *) \\
\text{end }
\]

- 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
\begin{verbatim}
\textbf{cobegin/\textit{coend}}

\textbf{cobegin}
\begin{align*}
  x & := y + z; & (* S_1 *) \\
  a & := b * c - y; & (* S_2 *)
\end{align*}
\textbf{coend}

\begin{itemize}
  \item No information flow among statements
    \begin{itemize}
      \item For $S_1$, $\text{lub}(\underline{y}, \underline{z}) \leq \underline{x}$
      \item For $S_2$, $\text{lub}(\underline{b}, \underline{c}, \underline{y}) \leq \underline{a}$
    \end{itemize}
  \item Security requirement is both must hold
    \begin{itemize}
      \item So this is secure if $\text{lub}(\underline{y}, \underline{z}) \leq \underline{x} \land \text{lub}(\underline{b}, \underline{c}, \underline{y}) \leq \underline{a}$
    \end{itemize}
\end{itemize}
\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 \( x \neq 1, \_ = \text{High}, \_ = \text{Low}, a = \text{Low}, \) 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:
    
    $$\text{if } PC \leq x \text{ then } x := x + 1 \text{ else skip}$$

• $\text{if } x = 0 \text{ then goto } n \text{ else } x := x - 1$ (branch and save PC on stack)
  – Same as:
    
    $$\text{if } x = 0 \text{ then begin}
    \quad \text{push}(PC, PC); \; PC := \text{lub}\{PC, x\}; \; PC := n;
    \text{end else if } PC \leq x \text{ then}
    \quad x := x - 1
    \text{else}
    \quad \text{skip;}$$
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 ≤ PC then PC := n else skip
    else
      if PC ≤ 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:
    \[ \text{if program stack empty then halt} \]
  – Note stack empty to prevent user obtaining information from it after halting
Example Program

1  if  $x = 0$  then  goto  4  else  $x := x - 1$
2  if  $z = 0$  then  goto  6  else  $z := z - 1$
3        halt
4  $z := z + 1$
5        return
6  $y := y + 1$
7        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 \( lub(x_1, \ldots, x_n) \)
  – 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
Handling This (2)

- Raise class of variables assigned to in conditionals even when branch not taken
- Also, verify information flow requirements even when branch not taken
- Example:
  - In \(\text{if } x = 0 \text{ then } z := 1\), \(z\) raised to \(x\) whether or not \(x = 0\)
  - Certification check in next statement, that \(z \leq y\), fails, as \(z = x\) from previous statement, and \(y \leq x\)
Handling This (3)

- Change classes only when explicit flows occur, but _all_ flows (implicit as well as explicit) force certification checks

- Example
  - When $x = 0$, first “if” sets $z$ to Low then checks $x \leq z$
  - When $x = 1$, first “if” checks that $x \leq z$
  - This holds if and only if $x = \text{Low}$
    - Not possible as $y < x = \text{Low}$ and there is no such class