Lecture 18: Information Flow

- Compiler-based mechanisms
- Execution-based mechanisms
- Examples
  - Security Pipeline Interface
  - Secure Network Server Mail Guard
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 flows in set of statements do not violate that policy
Example

\[\text{if } x = 1 \text{ then } y := a;\]
\[\text{else } y := b;\]

- Info flows from \(x\) and \(a\) to \(y\), or from \(x\) and \(b\) to \(y\)
- Certified only if \(x \leq y\) and \(a \leq y\) and \(b \leq y\)
  - Note flows for both branches must be true unless compiler can determine that one branch will never be taken
Declarations

• Notation:

\[ x: \text{int class} \{ A, B \} \]

means \( x \) is an integer variable with security class at least \( \text{lub}\{ A, B \} \), so \( \text{lub}\{ A, B \} \leq x \)

• Distinguished classes \( \text{Low}, \text{High} \)
  – Constants are always \( \text{Low} \)
Input Parameters

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

\[ i_p: \text{type class} \{ i_p \} \]
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:

  \[ o_p: \text{type class}\{r_1, ..., r_n\} \]

  where \( r_i \) is class of \( i \)th input or input/output argument
Example

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

• Require $x \leq \text{out}$ and $\text{out} \leq \text{out}$
```
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 \( \text{lub}\{ y, z \} \leq x \)

More generally:

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

• the relation \( \text{lub}\{ x_1, \ldots, x_n \} \leq y \) 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

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

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

More generally:

if $f(x_1, \ldots, x_n)$ then $S_1$ else $S_2$; end

• $S_1, S_2$ must be secure

• $\text{lub}\{ x_1, \ldots, x_n \} \leq \text{glb}\{ y \mid y \text{ target of assignment in } S_1, S_2 \}$
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
• $\text{lub}\{x_1, \ldots, x_n\} \leq \text{glb}\{y \mid y \text{ target of assignment in } S\}$
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
• $\text{lub}\{ x_1, \ldots, x_n \} \leq \\
  \quad \text{glb}\{ y \mid y \text{ target of assignment in } S \}$
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 int class {x});
    var y: array[1..10][1..10] of int class {y});
var i, j: int {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

$\begin{align*}
&b_1 \quad \rightarrow \quad b_2 \\
&b_2 \quad \rightarrow \quad b_7 \\
&b_6 \quad \rightarrow \quad b_4 \\
&b_4 \quad \rightarrow \quad b_3 \\
&b_3 \quad \rightarrow \quad b_5 \\
&b_5 \quad \rightarrow \quad b_6
\end{align*}$

$i > n$  
$i \leq n$  
$j > n$  
$j \leq n$
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 $\text{IFD}(b)$) 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($b_2$) = $b_7$  $b_2 \rightarrow b_7$ or $b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_7$
  - IFD($b_3$) = $b_4$  one path
  - IFD($b_4$) = $b_6$  $b_4 \rightarrow b_6$ or $b_4 \rightarrow b_5 \rightarrow b_6$
  - 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
- $x_{i_1}, \ldots, x_{i_n}$ 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
  - $\text{lub}\{ x_{i_1}, \ldots, x_{i_n} \} \leq \text{glb}\{ y \mid y \text{ target of assignment in } B_i \}$
Example of Requirements

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

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

March 2, 2009

ECS 235B, Winter Quarter 2009

Matt Bishop, UC Davis
Exceptions

proc \textit{copy}(x: \text{int class } \{ x \} ;
  \quad \text{var } y: \text{int class Low})
\var \text{sum: int class } \{ x \} ;
  \quad z: \text{int class Low};
begin
  \quad y := z := \text{sum} := 0;
  \quad \text{while } z = 0 \text{ do begin begin}
  \quad \quad \text{sum} := \text{sum} + x;
  \quad \quad y := y + 1;
  \quad \quad \text{end}
  \quad \text{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 $sum \leq z$
  - This is false ($\underline{sum} = \{ x \}$ dominates $\underline{z} = \text{Low}$)
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:

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

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

Consider statement

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

• Implicit flow from \( \text{sem} \) 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($S$) $\leq$ fglb($S$)

• begin $S_1$; … $S_n$ end
  – All $S_i$ must be secure
  – For all $i$, shared($S_i$) $\leq$ fglb($S_i$)
Example

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

• **Requirements:**
  - \( \text{lub}\{ y, z \} \leq x \)
  - \( \text{lub}\{ b, c, x \} \leq a \)
  - \( \underline{sem} \leq a \)
    - 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 $S_1, \ldots, S_n$ 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

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

wait(\( sem \)); \quad (*) \quad S_2 (*)

\[ i := i + 1; \quad (*) \quad S_3 (*) \]

end

• Conditions for this to be secure:
  
  – Loop terminates, so this condition met
  
  – \( S_1 \) secure if \( \text{lub\{i, item\}} \leq a[i] \)
  
  – \( S_2 \) secure if \( \text{sem} \leq i \) and \( \text{sem} \leq a[i] \)
  
  – \( S_3 \) trivially secure
\begin{cobegin}
\begin{align*}
x &:= y + z; \quad \text{\texttt{(* S_1 *)}} \\
a &:= b \times c - y; \quad \text{\texttt{(* S_2 *)}}
\end{align*}
\end{cobegin}

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

•  if \( x = 0 \) then goto \( n \) else \( x := x - 1 \) (branch and save PC on stack)
  - Same as:
    \[
    \text{if } x = 0 \text{ then begin}
    \text{push}(PC, PC); \ PC := \text{lub}(PC, x); \ PC := n;
    \text{end else if } PC \leq x \text{ then}
    \ x := x - 1
    \text{else}
    \skip;
    \]
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:
    \[ \text{if program stack empty then halt} \]
  – 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></th>
<th></th>
<th></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>Low ≤ x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td></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 \( \bar{z} \) 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 \( \lub\{\text{Low}, \bar{z}\} \leq y \)
  - 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 if $x = 0$ 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