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

```
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

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 } ≤ $\underline{x}$
- Distinguished classes *Low*, *High*
  - Constants are always *Low*

Input Parameters

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

```
ip: type class { ip }
```

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: ***type*** **class** { r_1, …, r_n }

  where r_i is class of *i*th input or input/output argument

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{ $\underline{a[i]}$, $\underline{i}$ }

- Information flowing in:

```
a[i] := …
```

- Only value of $a[i]$ affected, so class is $\underline{a[i]}$

Assignment Statements

$x := y + z;$

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

More generally:

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

- the relation lub{ $\underline{x}_1, \ldots, x_n$ } ≤ $\underline{y}$ must hold

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; … S_n;

- Each individual S_i must be secure

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, …, x_n) then S_1 else S_2; end`

- S_1, S_2 must be secure
- lub{ $\underline{x}_1, \ldots, \underline{x}_n$ } ≤ glb{ $\underline{y}$ | y 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:

$\texttt{while}\ f(x_1, \ldots, x_n)\ \texttt{do}\ S;$

- Loop must terminate;
- *S* must be secure
- lub{ $\underline{x}_1, \ldots, \underline{x}_n$ } ≤
  glb{$\underline{y}$ | y 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`, …,` x_n`) do` S`;`

- Loop must terminate;
- S must be secure
- $\text{lub}\{\, \underline{x}_1, \ldots, \underline{x}_n \,\} \leq$ $\text{glb}\{\underline{y} \mid y \text{ target of assignment in } S \,\}$

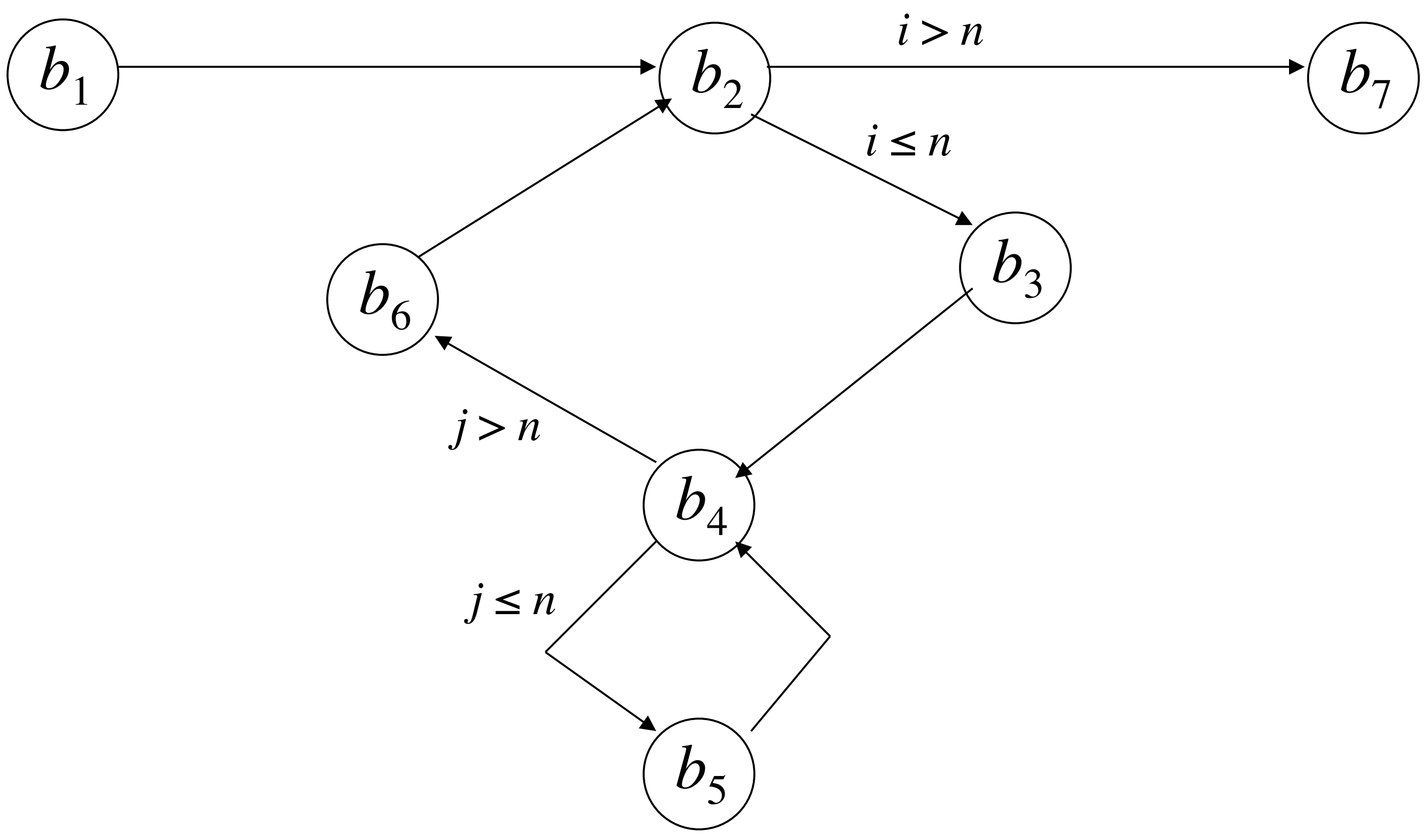
Goto Statements

- 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

```
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



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 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
- $x_{i1}, \ldots, x_{in}$ 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{ $\underline{x}_{i1}, \ldots, \underline{x}_{in}$ } ≤ glb{ $\underline{y}$ | y target of assignment in B_i }

Example of Requirements

- Within each basic block:

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

  b_5: $\text{lub}\{\ \underline{x[i][j]}, \underline{i}, \underline{j}\ \} \le \underline{y[j][i]}$ }; $\text{lub}\{\ \mathit{Low}, \underline{j}\ \} \le \underline{j}$

  – Combining, $\text{lub}\{\ \underline{x[i][j]}, \underline{i}, \underline{j}\ \} \le \underline{y[j][i]}$ }

  – From declarations, true when $\text{lub}\{\ \underline{x}, \underline{i}\ \} \le \underline{y}$

- $B_2 = \{b_3, b_4, b_5, b_6\}$

  – Assignments to i, j, $y[j][i]$; conditional is $i \le 10$

  – Requires $\underline{i} \le \text{glb}\{\ \underline{i}, \underline{j}, \underline{y[j][i]}\ \}$

  – From declarations, true when $\underline{i} \le \underline{y}$

Example (continued)

- $B_4 = \{ b_5 \}$
  - Assignments to j, $y[j][i]$; conditional is $j \le 10$
  - Requires $\underline{j} \le \text{glb}\{ \underline{j}, \underline{y[j][i]} \}$
  - From declarations, means $\underline{i} \le \underline{y}$
- Result:
  - Combine $\text{lub}\{ \underline{x}, \underline{i} \} \le \underline{y}$; $\underline{i} \le \underline{y}$; $\underline{i} \le \underline{y}$
  - Requirement is $\text{lub}\{ \underline{x}, \underline{i} \} \le \underline{y}$

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 $\underline{a} \le \underline{b}$

More generally:

proc $pn(i_1, \ldots, i_m$: int; var $o_1, \ldots, o_n$: 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 $\underline{y_j} \le \underline{y_k}$

Exceptions

```
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 $\underline{x} \le \underline{y}$ 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 $\underline{sum} \le \underline{z}$
  - This is false ($\underline{sum}$ = { x } dominates $\underline{z}$ = Low)

Infinite Loops

```
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:

```
wait(x):   if x = 0 then block until x > 0; x := x – 1;
signal(x): x := x + 1;
```

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

Consider statement

```
wait(sem); 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
  - 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) ≤ fglb(S)
- begin S_1; … S_n end
  - All S_i must be secure
  - For all i, shared(S_i) ≤ fglb(S_i)

Example

```
begin
    x := y + z;          (* S1 *)
    wait(sem);           (* S2 *)
    a := b * c – x;      (* S3 *)
end
```

- Requirements:
  - lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
  - lub{ $\underline{b}$, $\underline{c}$, $\underline{x}$ } ≤ $\underline{a}$
  - $\underline{sem}$ ≤ $\underline{a}$
    - Because fglb(S_2) = $\underline{a}$ and 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
  - lub{ $\underline{\text{shared}(S_1)}, \ldots, \underline{\text{shared}(S_n)}$ } $\leq$ 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] := item;        (* S1 *)
    wait(sem);           (* S2 *)
    i := i + 1;          (* S3 *)
end
```

- Conditions for this to be secure:
  - Loop terminates, so this condition met
  - S_1 secure if lub{ $\underline{i}$, $\underline{item}$ } ≤ $\underline{a[i]}$
  - S_2 secure if $\underline{sem} \leq \underline{i}$ and $\underline{sem} \leq \underline{a[i]}$
  - S_3 trivially secure

cobegin/coend

```
cobegin
      x := y + z;           (* S1 *)
      a := b * c – y;       (* S2 *)
coend
```

- No information flow among statements
  - For S_1, lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
  - For S_2, lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } ≤ $\underline{a}$
- Security requirement is both must hold
  - So this is secure if lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$ ∧ lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } ≤ $\underline{a}$

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, $\underline{x} \leq \underline{y}$

```
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!

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, $\underline{x}$) means push variable x and its security class $\underline{x}$ onto program stack
- *pop*(x, $\underline{x}$) means pop top value and security class from program stack, assign them to variable x and its security class $\underline{x}$ respectively

Instructions

- `x := x + 1` (increment)
  - Same as:

    `if PC ≤ 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 := lub{PC, x}; PC := n;
end else if PC ≤ 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 ≤ 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:

    `pop(`*PC*`, `<u>*PC*</u>`);`
- `halt` (stop)
  – Same as:

    `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

x	y	z	PC	$\underline{PC}$	*stack*	*check*
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low ≤ $\underline{x}$
0	0	0	6	$\underline{z}$	(3, Low)	
0	1	0	7	$\underline{z}$	(3, Low)	$\underline{PC} \leq \underline{y}$
0	1	0	3	Low	—	

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 $\underline{PC}$
  - On assignment of form $y := f(x_1, \ldots, x_n)$, $\underline{y}$ changed to lub$\{ \underline{x}_1, \ldots, \underline{x}_n \}$
  - 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;
```

- $\underline{z}$ changes when z assigned to
- Assume $\underline{y} < \underline{x}$

Analysis of Example

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

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 $\underline{z} \le \underline{y}$, fails, as $\underline{z} = \underline{x}$ from previous statement, and $\underline{y} \le \underline{x}$

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