Chapter 16: Information Flow

- Entropy and analysis
- Non-lattice information flow policies
- Compiler-based mechanisms
- Execution-based mechanisms
- Examples

Overview

- Basics and background
  - Entropy
- Nonlattice flow policies
- Compiler-based mechanisms
- Execution-based mechanisms
- Examples
  - Security Pipeline Interface
  - Secure Network Server Mail Guard

Basics

- Bell-LaPadula Model embodies information flow policy
  - Given compartments *A*, *B*, info can flow from *A* to *B* iff *B dom A*
- Variables *x*, *y* assigned compartments $\underline{x}$, $\underline{y}$ as well as values
  - If $\underline{x}$ = *A* and $\underline{y}$ = B, and *A dom B*, then *y* := *x* allowed but not *x* := *y*

Entropy and Information Flow

- Idea: info flows from x to y as a result of a sequence of commands c if you can deduce information about x before c from the value in y after c
- Formally:
  - s time before execution of c, t time after
  - $H(x_s \mid y_t) < H(x_s \mid y_s)$
  - If no y at time s, then $H(x_s \mid y_t) < H(x_s)$

Example 1

- Command is $x := y + z$; where:
  - $0 \le y \le 7$, equal probability
  - $z = 1$ with prob. 1/2, $z = 2$ or 3 with prob. 1/4 each
- *s* state before command executed; *t*, after; so
  - $H(y_s) = H(y_t) = -8(1/8) \lg (1/8) = 3$
  - $H(z_s) = H(z_t) = -(1/2) \lg (1/2) - 2(1/4) \lg (1/4) = 1.5$
- If you know x_t, y_s can have at most 3 values, so $H(y_s \mid x_t) = -3(1/3) \lg (1/3) = \lg 3$

Example 2

- Command is
  - **if** $x = 1$ **then** $y := 0$ **else** $y := 1$;

  where:
  - x, y equally likely to be either 0 or 1
- $H(x_s) = 1$ as x can be either 0 or 1 with equal probability
- $H(x_s \mid y_t) = 0$ as if $y_t = 1$ then $x_s = 0$ and vice versa
  - Thus, $H(x_s \mid y_t) = 0 < 1 = H(x_s)$
- So information flowed from x to y

Implicit Flow of Information

- Information flows from x to y without an *explicit* assignment of the form $y := f(x)$
  - $f(x)$ an arithmetic expression with variable x
- Example from previous slide:
  - **if** $x = 1$ **then** $y := 0$
    else $y := 1$;
- So must look for implicit flows of information to analyze program

Notation

- $\underline{x}$ means class of x
  - In Bell-LaPadula based system, same as "label of security compartment to which x belongs"
- $\underline{x} \le \underline{y}$ means "information can flow from an element in class of x to an element in class of y
  - Or, "information with a label placing it in class $\underline{x}$ can flow into class $\underline{y}$"

Information Flow Policies

Information flow policies are usually:

- reflexive
  - So information can flow freely among members of a single class
- transitive
  - So if information can flow from class 1 to class 2, and from class 2 to class 3, then information can flow from class 1 to class 3

Non-Transitive Policies

- Betty is a confident of Anne
- Cathy is a confident of Betty
  - With transitivity, information flows from Anne to Betty to Cathy
- Anne confides to Betty she is having an affair with Cathy's spouse
  - Transitivity undesirable in this case, probably

Non-Lattice Transitive Policies

- 2 faculty members co-PIs on a grant
  - Equal authority; neither can overrule the other
- Grad students report to faculty members
- Undergrads report to grad students
- Information flow relation is:
  - Reflexive and transitive
- But some elements (people) have no “least upper bound” element
  - What is it for the faculty members?

Confidentiality Policy Model

- Lattice model fails in previous 2 cases
- Generalize: policy $I = (SC_I, \leq_I, join_I)$:
  - SC_I set of security classes
  - $\leq_I$ ordering relation on elements of SC_I
  - $join_I$ function to combine two elements of SC_I
- Example: Bell-LaPadula Model
  - SC_I set of security compartments
  - $\leq_I$ ordering relation *dom*
  - $join_I$ function *lub*

Confinement Flow Model

- $(I, O, confine, \rightarrow)$
  - $I = (SC_I, \leq_I, join_I)$
  - O set of entities
  - $\rightarrow$: $O \times O$ with $(a, b) \in \rightarrow$ (written $a \rightarrow b$) iff information can flow from a to b
  - for $a \in O$, $confine(a) = (a_L, a_U) \in SC_I \times SC_I$ with $a_L \leq_I a_U$
    - Interpretation: for $a \in O$, if $x \leq_I a_U$, info can flow from x to a, and if $a_L \leq_I x$, info can flow from a to x
    - So a_L lowest classification of info allowed to flow out of a, and a_U highest classification of info allowed to flow into a

Assumptions, *etc*.

- Assumes: object can change security classes
  - So, variable can take on security class of its data
- Object x has security class $\underline{x}$ currently
- Note transitivity *not* required
- If information can flow from a to b, then b dominates a under ordering of policy I:

$$(\forall\, a, b \in O)[\, a \rightarrow b \Rightarrow a_L \leq_I b_U \,]$$

Example 1

- $SC_I = \{ \text{U, C, S, TS} \}$, with U $\leq_I$ C, C $\leq_I$ S, and S $\leq_I$ TS
- $a, b, c \in O$
  - confine(a) = [C, C]
  - confine(b) = [S, S]
  - confine(c) = [TS, TS]
- Secure information flows: $a \rightarrow b$, $a \rightarrow c$, $b \rightarrow c$
  - As $a_L \leq_I b_U$, $a_L \leq_I c_U$, $b_L \leq_I c_U$
  - Transitivity holds

Example 2

- SC_I, $\leq_I$ as in Example 1
- $x, y, z \in O$
  - confine(x) = [C, C]
  - confine(y) = [S, S]
  - confine(z) = [C, TS]
- Secure information flows: $x \rightarrow y$, $x \rightarrow z$, $y \rightarrow z$, $z \rightarrow x$, $z \rightarrow y$
  - As $x_L \leq_I y_U$, $x_L \leq_I z_U$, $y_L \leq_I z_U$, $z_L \leq_I x_U$, $z_L \leq_I y_U$
  - Transitivity does not hold
    - $y \rightarrow z$ and $z \rightarrow x$, but $y \rightarrow z$ is false, because $y_L \leq_I x_U$ is false

Transitive Non-Lattice Policies

- Q = (S_Q, $\leq_Q$) is a *quasi-ordered set* when $\leq_Q$ is transitive and reflexive over S_Q
- How to handle information flow?
  - Define a partially ordered set containing quasi-ordered set
  - Add least upper bound, greatest lower bound to partially ordered set
  - It's a lattice, so apply lattice rules!

In Detail …

- $\forall x \in S_Q$: let $f(x) = \{ y \mid y \in S_Q \wedge y \leq_Q x \}$
  - Define $S_{QP} = \{ f(x) \mid x \in S_Q \}$
  - Define $\leq_{QP} = \{ (x, y) \mid x, y \in S_Q \wedge x \subseteq y \}$
    - S_{QP} partially ordered set under $\leq_{QP}$
    - f preserves order, so $y \leq_Q x$ iff $f(x) \leq_{QP} f(y)$
- Add upper, lower bounds
  - $S_{QP}' = S_{QP} \cup \{ S_Q, \varnothing \}$
  - Upper bound $ub(x, y) = \{ z \mid z \in S_{QP} \wedge x \subseteq z \wedge y \subseteq z \}$
  - Least upper bound $lub(x, y) = \cap ub(x, y)$
    - Lower bound, greatest lower bound defined analogously

And the Policy Is …

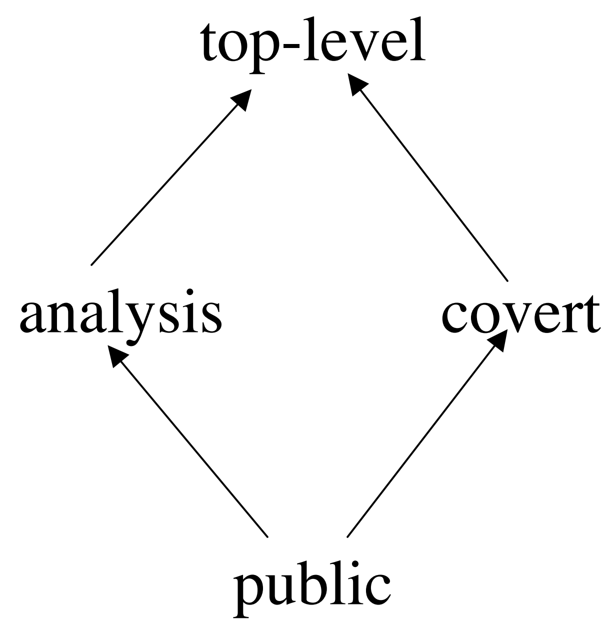
- Now $(S_{QP}', \leq_{QP})$ is lattice
- Information flow policy on quasi-ordered set emulates that of this lattice!

Nontransitive Flow Policies

- Government agency information flow policy (on next slide)
- Entities public relations officers PRO, analysts A, spymasters S
  - *confine*(PRO) = { public, analysis }
  - *confine*(A) = { analysis, top-level }
  - *confine*(S) = { covert, top-level }

Information Flow

- By confinement flow model:
  - PRO ≤ A, A ≤ PRO
  - PRO ≤ S
  - A ≤ S, S ≤ A
- Data *cannot* flow to public relations officers; not transitive
  - S ≤ A, A ≤ PRO
  - S ≤ PRO is *false*



Transforming Into Lattice

- Rough idea: apply a special mapping to generate a subset of the power set of the set of classes
  - Done so this set is partially ordered
  - Means it can be transformed into a lattice
- Can show this mapping preserves ordering relation
  - So it preserves non-orderings and non-transitivity of elements corresponding to those of original set

Dual Mapping

- $R = (SC_R, \leq_R, join_R)$ reflexive info flow policy
- $P = (S_P, \leq_P)$ ordered set
  - Define *dual mapping* functions l_R, h_R: $SC_R \rightarrow S_P$
    - $l_R(x) = \{ x \}$
    - $h_R(x) = \{ y \mid y \in SC_R \wedge y \leq_R x \}$
  - S_P contains subsets of SC_R; $\leq_P$ subset relation
  - Dual mapping function *order preserving* iff

$$(\forall a, b \in SC_R)[\ a \leq_R b \Leftrightarrow l_R(a) \leq_P h_R(b)\ ]$$

Theorem

Dual mapping from reflexive info flow policy R to ordered set P order-preserving

Proof sketch: all notation as before

(⇒) Let $a \leq_R b$. Then $a \in l_R(a)$, $a \in h_R(b)$, so $l_R(a) \subseteq h_R(b)$, or $l_R(a) \leq_P h_R(b)$

(⇐) Let $l_R(a) \leq_P h_R(b)$. Then $l_R(a) \subseteq h_R(b)$. But $l_R(a) = \{ a \}$, so $a \in h_R(b)$, giving $a \leq_R b$

Info Flow Requirements

- Interpretation: let *confine*(x) = { $\underline{x}_L$, $\underline{x}_U$ }, consider class $\underline{y}$
  - Information can flow from x to element of $\underline{y}$ iff $\underline{x}_L \leq_R \underline{y}$, or $l_R(\underline{x}_L) \subseteq h_R(\underline{y})$
  - Information can flow from element of $\underline{y}$ to x iff $y \leq_R \underline{x}_U$, or $l_R(\underline{y}) \subseteq h_R(\underline{x}_U)$

Revisit Government Example

- Information flow policy is *R*
- Flow relationships among classes are:

public $\leq_R$ public
public $\leq_R$ analysis
public $\leq_R$ covert
public $\leq_R$ top-level
analysis $\leq_R$ top-level
analysis $\leq_R$ analysis
covert $\leq_R$ covert
covert $\leq_R$ top-level
top-level $\leq_R$ top-level

Dual Mapping of *R*

- Elements l_R, h_R:

  l_R(public) = { public }

  h_R(public = { public }

  l_R(analysis) = { analysis }

  h_R(analysis) = { public, analysis }

  l_R(covert) = { covert }

  h_R(covert) = { public, covert }

  l_R(top-level) = { top-level }

  h_R(top-level) = { public, analysis, covert, top-level }

confine

- Let *p* be entity of type PRO, *a* of type A, *s* of type S
- In terms of *P* (not *R*), we get:
  - *confine*(*p*) = [{ public }, { public, analysis }]
  - *confine*(*a*) = [{ analysis },
    { public, analysis, covert, top-level }]
  - *confine*(*s*) = [{ covert },
    { public, analysis, covert, top-level }]

And the Flow Relations Are …

- $p \rightarrow a$ as $l_R(p) \subseteq h_R(a)$
  - $l_R(p)$ = { public }
  - $h_R(a)$ = { public, analysis, covert, top-level }
- Similarly: $a \rightarrow p$, $p \rightarrow s$, $a \rightarrow s$, $s \rightarrow a$
- ***But*** $s \rightarrow p$ ***is false*** as $l_R(s) \not\subseteq h_R(p)$
  - $l_R(s)$ = { covert }
  - $h_R(p)$ = { public, analysis }

Analysis

- $(S_P, \leq_P)$ is a lattice, so it can be analyzed like a lattice policy
- Dual mapping preserves ordering, hence non-ordering and non-transitivity, of original policy
  - So results of analysis of $(S_P, \leq_P)$ can be mapped back into $(SC_R, \leq_R, join_R)$

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

i_p: ***type*** **class { $\mathrm{i_p}$ }**

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**: *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 <u>*x*</u> ≤ <u>*out*</u> and <u>*out*</u> ≤ <u>*out*</u>

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:

```
y := f(x1, …, xn)
```

- the relation lub{ $\underline{x}_1$, …, 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:

$S_1; \; \ldots \; S_n;$

- 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` $f(x_1, \ldots, x_n)$ `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:

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

- Loop must terminate;
- S must be secure
- $\text{lub}\{ \underline{x}_1, \ldots, \underline{x}_n \} \leq$

  $\text{glb}\{\underline{y} \mid y \text{ target of assignment in } S \}$

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(x1, …, xn) 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* }

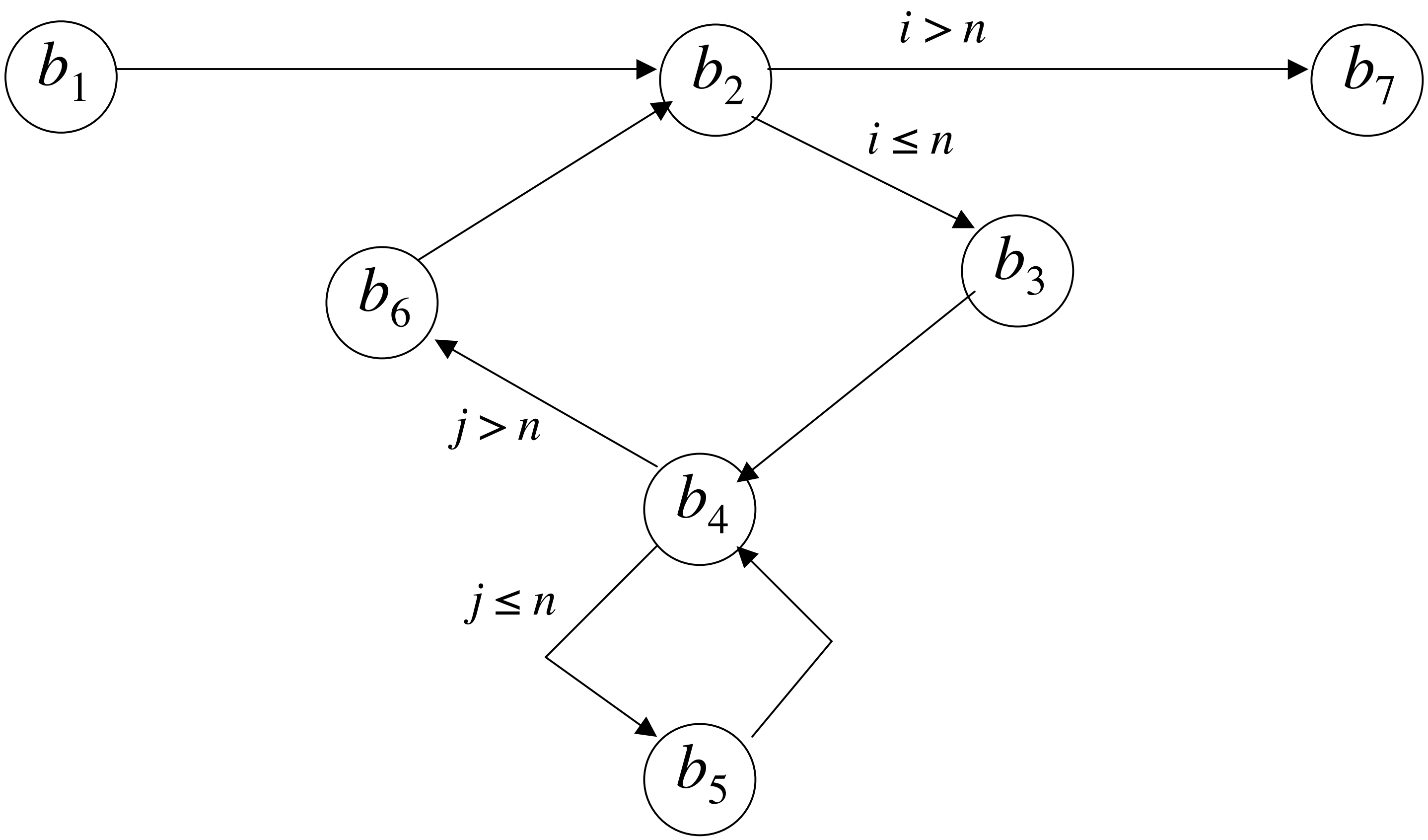
Goto Statements

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

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

  – 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 $\underline{a} \leq \underline{b}$

More generally:

```
proc pn(i1, …, im: int; var o1, …, on: int)
begin S end;
```

- *S* must be secure
- For all *j* and *k*, if $\underline{i}_j \leq \underline{o}_k$, then $\underline{x}_j \leq \underline{y}_k$
- For all *j* and *k*, if $\underline{o}_j \leq \underline{o}_k$, then $\underline{y}_j \leq \underline{y}_k$

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, <u>shared(S_i)</u> ≤ 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{ $\underline{i}$, $\underline{item}$ } $\leq$ $\underline{a[i]}$
  - 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, $\underline{x} \le \underline{y}$

```
if x = 1 then y := a;
```

  - When $x \ne 1$, $\underline{x}$ = 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` $\underline{x} \le \underline{PC}$ `then` PC `:=` n `else` *skip*

    `else`

    `if` $\underline{PC} \le \underline{x}$ `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)	$\underline{PC}$ ≤ $\underline{y}$
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 $\underline{z} \leq \underline{y}$, fails, as $\underline{z} = \underline{x}$ from previous statement, and $\underline{y} \leq \underline{x}$

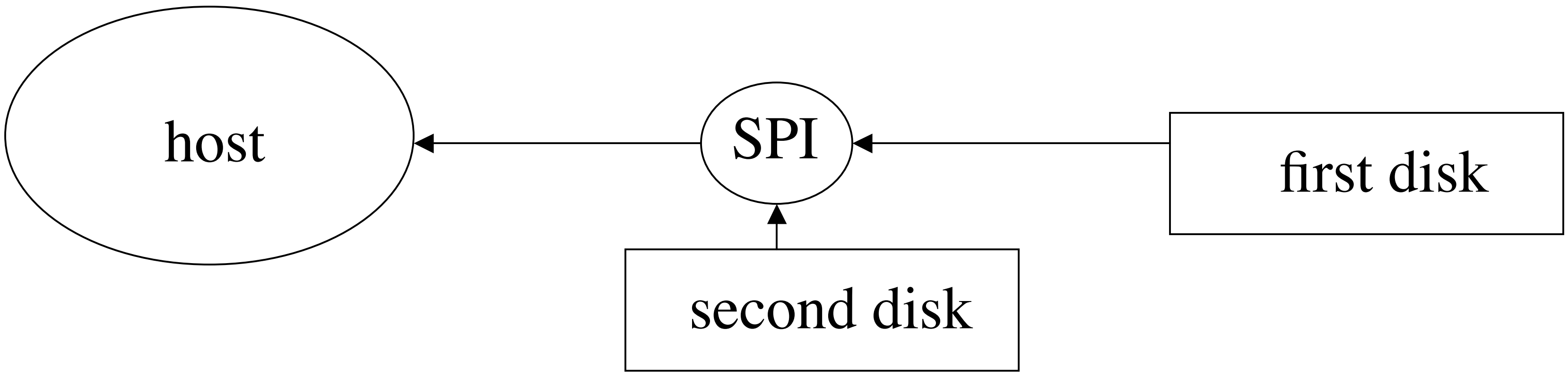
Handling This (3)

- 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

Example Information Flow Control Systems

- Use access controls of various types to inhibit information flows
- Security Pipeline Interface
  - Analyzes data moving from host to destination
- Secure Network Server Mail Guard
  - Controls flow of data between networks that have different security classifications

Security Pipeline Interface

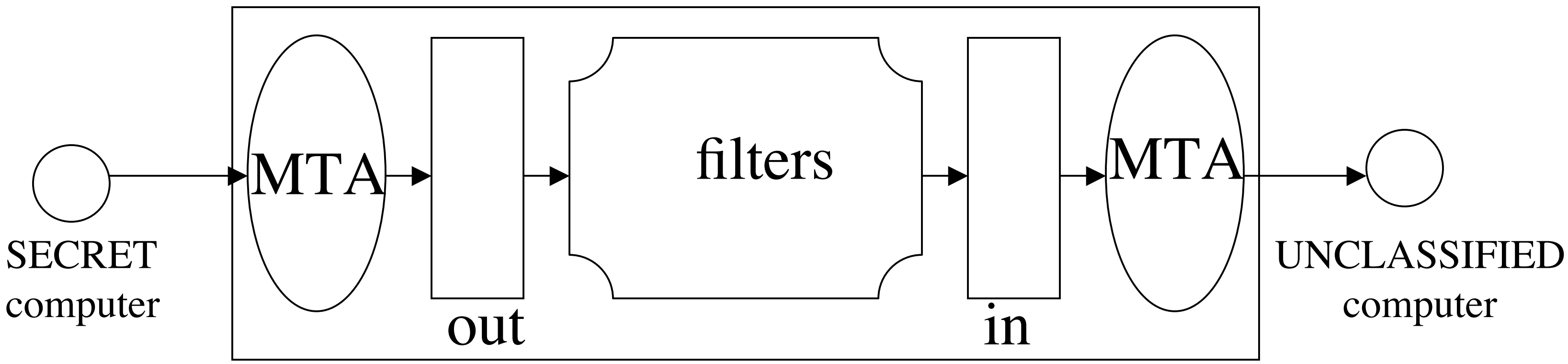


- SPI analyzes data going to, from host
  - No access to host main memory
  - Host has no control over SPI

Use

- Store files on first disk
- Store corresponding crypto checksums on second disk
- Host requests file from first disk
  - SPI retrieves file, computes crypto checksum
  - SPI retrieves file's crypto checksum from second disk
  - If a match, file is fine and forwarded to host
  - If discrepency, file is compromised and host notified
- Integrity information flow restricted here
  - Corrupt file can be seen but will not be trusted

Secure Network Server Mail Guard (SNSMG)



- Filters analyze outgoing messages
  - Check authorization of sender
  - Sanitize message if needed (words and viruses, etc.)
- Uses type checking to enforce this
  - Incoming, outgoing messages of different type
  - Only appropriate type can be moved in or out

Key Points

- Both amount of information, direction of flow important
  - Flows can be explicit or implicit
- Analysis assumes lattice model
  - Non-lattices can be embedded in lattices
- Compiler-based checks flows at compile time
- Execution-based checks flows at run time