Chapter 15: Information Flow

- Definitions
- Compiler-based mechanisms
- Execution-based mechanisms
- Examples

Overview

- Basics and background
- 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*

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

Example 1

- Command is $x := y + z$; where:
  - $0 \leq y \leq 7$, equal probability
  - $z = 1$ with prob. 1/2, $z = 2$ or 3 with prob. 1/4 each
- If you know final value of x, initial value of y can have at most 3 values, so information flows from y to x

Example 2

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

  where:
  - x, y equally likely to be either 0 or 1
- But if $x = 1$ then $y = 0$, and vice versa, so value of y depends on x
- 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"
- $\underline{x} \leq \underline{y}$ 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 $\underline{y}$"

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 } $\leq \underline{x}$
- 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 { 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 $\underline{x} \leq \underline{out}$ 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{ $\underline{y}$, $\underline{z}$ } $\leq \underline{x}$

More generally:

y **:=** f**(**x_1**, …,** x_n**)**

- the relation lub{ $\underline{x}_1, \ldots, x_n$ } $\leq \underline{y}$ 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{ $\underline{x}_1$, …, $\underline{x}_n$ } ≤
  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
- lub$\{ \underline{x}_1, \ldots, \underline{x}_n \} \leq$ glb$\{\underline{y} \mid y$ 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(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 }

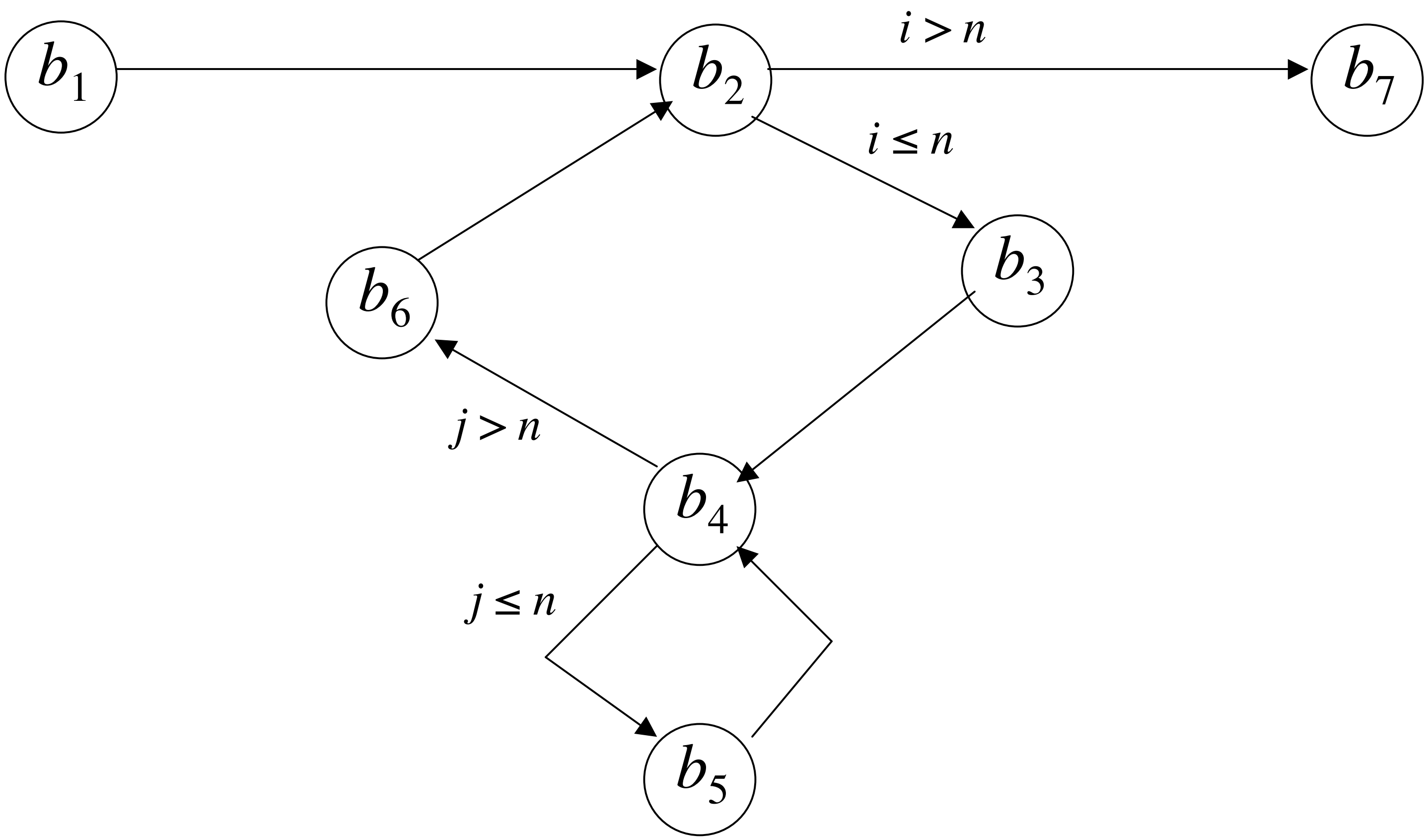
Goto Statements

- 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:
  - $\text{IFD}(b_1) = b_2$ 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 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: $Low \le \underline{i}$ $\quad$ b_3: $Low \le \underline{j}$ $\quad$ b_6: lub{ Low, $\underline{i}$ } $\le \underline{i}$

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

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

  – From declarations, true when 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 $\underline{i} \le$ 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 $j \leq 10$
  - Requires $\underline{j} \leq \text{glb}\{ \underline{j}, \underline{y[j][i]} \}$
  - From declarations, means $\underline{i} \leq \underline{y}$
- Result:
  - Combine $\text{lub}\{ \underline{x}, \underline{i} \} \leq \underline{y}$; $\underline{i} \leq \underline{y}$; $\underline{i} \leq \underline{y}$
  - Requirement is $\text{lub}\{ \underline{x}, \underline{i} \} \leq \underline{y}$

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(i₁, …, iₘ: int; var o₁, …, oₙ: 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, $\underline{\text{shared}(S_i)} \leq \text{fglb}(S_i)$

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{ $\underline{i}$, $\underline{item}$ } $\le$ $\underline{a[i]}$
  - S_2 secure if $\underline{sem} \le \underline{i}$ and $\underline{sem} \le \underline{a[i]}$
  - 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{ $\underline{y}$, $\underline{z}$ } $\leq \underline{x}$
  - For S_2, lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } $\leq \underline{a}$
- Security requirement is both must hold
  - So this is secure if lub{ $\underline{y}$, $\underline{z}$ } $\leq \underline{x} \wedge$ lub{ $\underline{b}$, $\underline{c}$, $\underline{y}$ } $\leq \underline{a}$

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 $\underline{PC} \le \underline{x}$ 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)	$\underline{PC} \le \underline{y}$
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 $\underline{y} \leq \underline{x}$

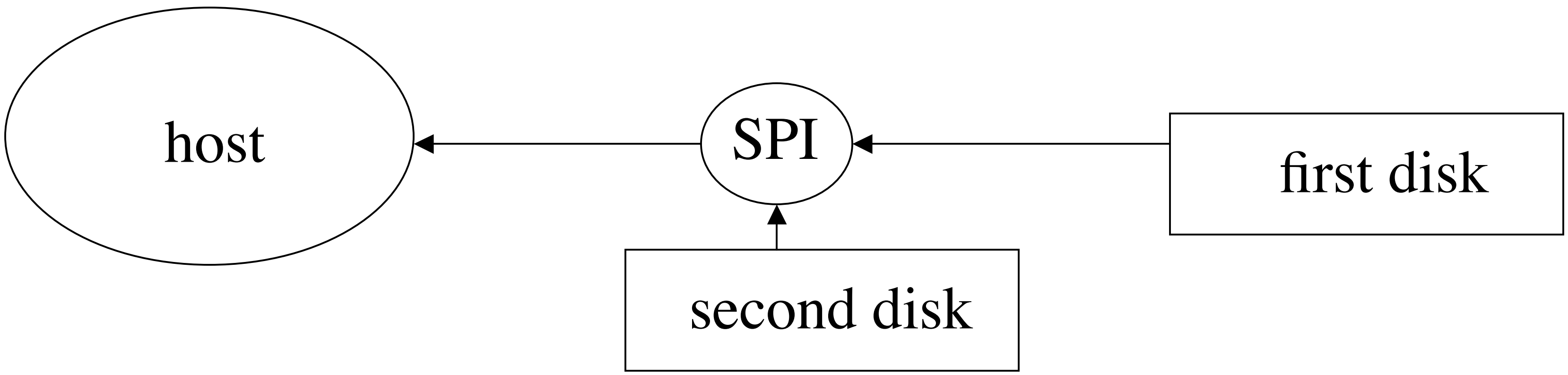
Handling This (3)

- 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 $\underline{x} \le \underline{z}$
  - When x = 1, first "if" checks that $\underline{x} \le \underline{z}$
  - 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

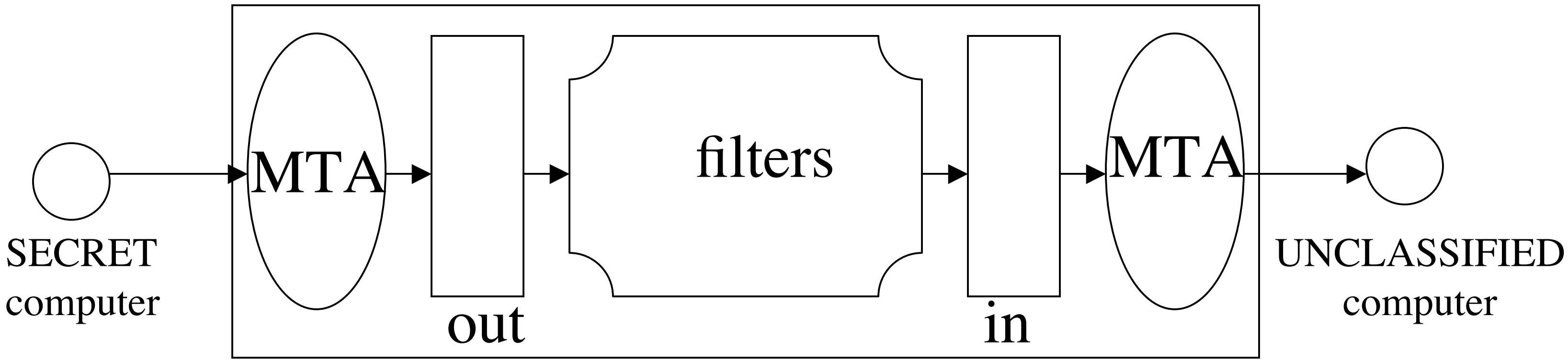


- SPI analyzes data going to, from host
  - No access to host main memory
  - Host has no control over SPI

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
- Compiler-based checks flows at compile time
- Execution-based checks flows at run time