# Lecture 20

- Compiler-based mechanisms
- Execution-based mechanisms
- The confinement problem
- Isolation: virtual machines, sandboxes
- Covert channels
  - Detection
  - Mitigation

# Procedure Calls

*tm*(*a*, *b*);

From previous slides, to be secure, $lub(\underline{x}, \underline{i}) \leq \underline{y}$ must hold

- In call, $x$ corresponds to $a$, $y$ to $b$
- Means that $lub(\underline{a}, \underline{i}) \leq \underline{b}$, or $\underline{a} \leq \underline{b}$

More generally:

```
proc pn(i_1, . . ., i_m: int; var o_1, . . ., o_n: 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 $\underline{shared(S)} \leq fglb(S)$
- begin $S_1$; . . . $S_n$ end
  - All $S_i$ must be secure
  - For all $i$, $\underline{shared(S_i)} \leq fglb(S_i)$

# Example

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

- Requirements:
  - $lub(\underline{y}, \underline{z}) \le \underline{x}$
  - $lub(\underline{b}, \underline{c}, \underline{x}) \le \underline{a}$
  - $\underline{sem} \le \underline{a}$
    - 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
  - $lub(\underline{shared(S_1)}, \ldots, \underline{shared(S_n)}\ \} \le glb(t_1, \ldots, t_m)$
    - 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 $S_1$, $lub(\underline{y}, \underline{z}) \le \underline{x}$
  - For $S_2$, $lub(\underline{b}, \underline{c}, \underline{y}) \le \underline{a}$
- Security requirement is both must hold
  - So this is secure if $lub(\underline{y}, \underline{z}) \le \underline{x} \land lub(\underline{b}, \underline{c}, \underline{y}) \le \underline{a}$

# 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 $x \neq 1$, $\underline{x}$ = 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` $\underline{x} \le \underline{PC}$ `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 $\leq \underline{x}$ |
| 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 $\underline{z} \leq \underline{y}$, fails, as $\underline{z} = \underline{x}$ from previous statement, and $\underline{y} \leq \underline{x}$

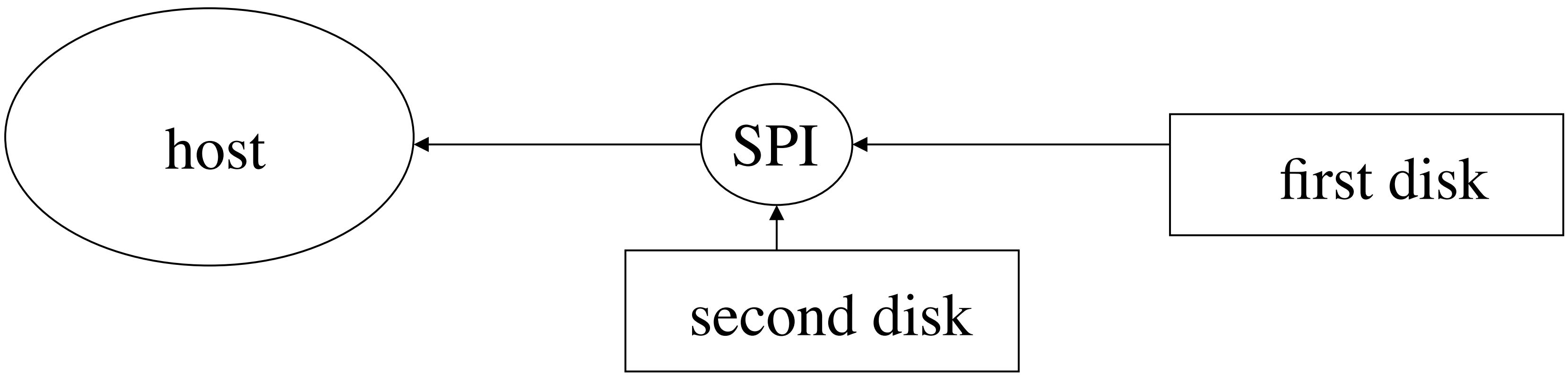
# Handling This (3)

- 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

# Examples

- 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

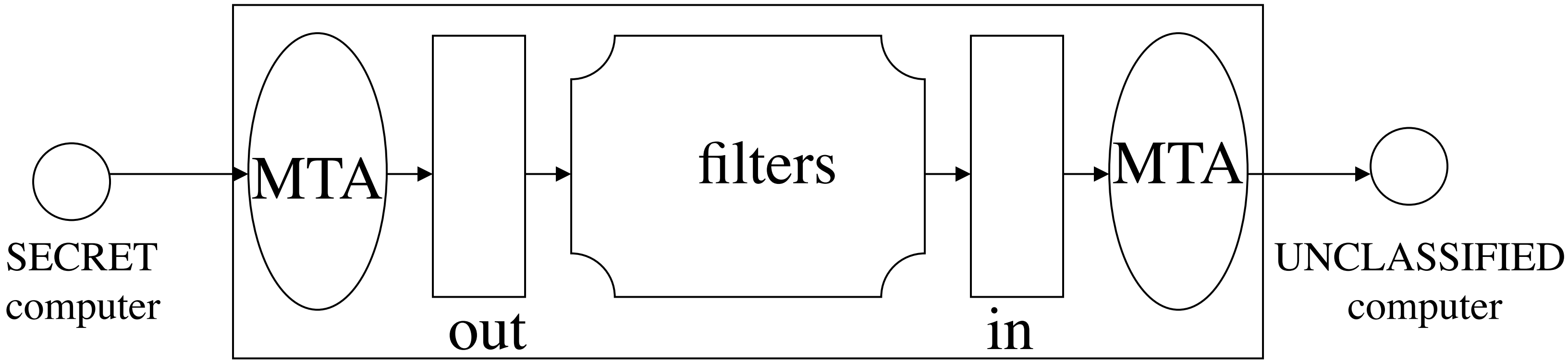



- SPI analyzes data going to, from host
  - No access to host main memory
  - Host has no control over SPI

# 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 discrepancy, 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

# Confinement

- What is the problem?
- Isolation: virtual machines, sandboxes
- Detecting covert channels

# Example Problem

- Server balances bank accounts for clients
- Server security issues:
  - Record correctly who used it
  - Send *only* balancing info to client
- Client security issues:
  - Log use correctly
  - Do not save or retransmit data client sends

# Generalization

- Client sends request, data to server
- Server performs some function on data
- Server returns result to client
- Access controls:
  - Server must ensure the resources it accesses on behalf of client include *only* resources client is authorized to access
  - Server must ensure it does not reveal client's data to any entity not authorized to see the client's data

# Confinement Problem

- Problem of preventing a server from leaking information that the user of the service considers confidential

# Total Isolation

- Process cannot communicate with any other process
- Process cannot be observed

Impossible for this process to leak information

- Not practical as process uses observable resources such as CPU, secondary storage, networks, etc.

# Example

- Processes *p*, *q* not allowed to communicate
  - But they share a file system!
- Communications protocol:
  - *p* sends a bit by creating a file called *0* or *1*, then a second file called *send*
    - *p* waits until *send* is deleted before repeating to send another bit
  - *q* waits until file *send* exists, then looks for file *0* or *1*; whichever exists is the bit
    - *q* then deletes *0*, *1*, and *send* and waits until *send* is recreated before repeating to read another bit

# Covert Channel

- A path of communication not designed to be used for communication
- In example, file system is a (storage) covert channel

# Rule of Transitive Confinement

- If $p$ is confined to prevent leaking, and it invokes $q$, then $q$ must be similarly confined to prevent leaking
- Rule: if a confined process invokes a second process, the second process must be as confined as the first

# Lipner's Notes

- All processes can obtain rough idea of time
  - Read system clock or wall clock time
  - Determine number of instructions executed
- All processes can manipulate time
  - Wait some interval of wall clock time
  - Execute a set number of instructions, then block

# Kocher's Attack

- This computes $x = a^z \bmod n$, where $z = z_0 \ldots z_{k-1}$

```
x := 1; atmp := a;
for i := 0 to k–1 do begin
  if z_i = 1 then
      x := (x * atmp) mod n;
  atmp := (atmp * atmp) mod n;
end
result := x;
```

- Length of run time related to number of 1 bits in $z$

# Isolation

- Present process with environment that appears to be a computer running only those processes being isolated
  - Process cannot access underlying computer system, any process(es) or resource(s) not part of that environment
  - A *virtual machine*
- Run process in environment that analyzes actions to determine if they leak information
  - Alters the interface between process(es) and computer