ECS 235B Module 52
Execution Based
Information Flow Mechanisms

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`*: instruction not executed
- *`push(x, x)`*: push variable *x* and its security class <u>*x*</u> onto program stack
- *`pop(x, x)`* : pop top value and security class from program stack, assign them to variable *x* and its security class <u>*x*</u> 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 $\underline{x} \leq \underline{PC}$ **then** $PC := n$ **else** *skip*
    else
      if $\underline{PC} \leq \underline{x}$ **then** $x := x - 1$ **else** *skip*

More Instructions

- **`return`** (go to just after last *if*)
  - Same as:

    `pop``(`*`PC`*`, `*`PC`*`);`
- **`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)	$\underline{PC} \leq \underline{y}$
0	1	0	7	$\underline{z}$	(3, Low)	
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: integer class { x };
               var y: integer class { y })
var z: integer 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 $\underline{y} < \underline{x}$ (that is, $\underline{x}$ strictly dominates $\underline{y}$; they are not equal)

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`, $\underline{z}$ raised to $\underline{x}$ 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 $\underline{y} < \underline{x}$

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 $\underline{x} \leq \underline{z}$
  - This holds if and only if $\underline{x}$ = Low
    - Not possible as $\underline{y} < \underline{x}$ = Low by assumption and there is no class that Low strictly dominates

Quiz

Should a statement that causes an error be ignored, and execution continue?

1. Yes; if the program is aborted or a visible exception is taken, the user could deduce information about values in the program
2. Yes; such a statement cannot be certified and so it must be ignored
3. No; the user must be informed lest they draw an incorrect conclusion about values in the program
4. No; the user's clearance may allow them to see that an error occurred