Lecture 24
November 16, 2022
Basics of Information Flow

• Bell-LaPadula Model embodies information flow policy
  • Given compartments $A$, $B$, info can flow from $A$ to $B$ iff $B \text{ dom } A$

• So does Biba Model
  • Given compartments $A$, $B$, info can flow from $A$ to $B$ iff $A \text{ dom } B$

• Variables $x$, $y$ assigned compartments $x$, $y$ as well as values
  • Confidentiality (Bel-LaPadula): if $x = A$, $y = B$, and $B \text{ dom } A$, then $y := x$ allowed but not $x := y$
  • Integrity (Biba): if $x = A$, $y = B$, and $A \text{ dom } B$, then $x := y$ allowed but not $y := x$

• For now, focus on confidentiality (Bell-LaPadula)
  • We’ll get to integrity later
Entropy and Information Flow

• Idea: information 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
  • $H(x_s | y_t) < H(x_s | y_s)$
  • If no $y$ at time $s$, then $H(x_s | y_t) < H(x_s)$
Example 1

• Command is $x := y + z$; where:
  • $x$ does not exist initially (that is, has no value)
  • $0 \leq y \leq 7$, equal probability
  • $z = 1$ with probability $1/2$, $z = 2$ or $3$ with probability $1/4$ each

• $s$ state before command executed; $t$, after; so
  • $H(y_s) = H(y_t) = -8(1/8) \lg (1/8) = 3$

• You can show that $H(y_s | x_t) = (3/32) \lg 3 + 9/8 \approx 1.274 < 3 = H(y_s)$
  • Thus, information flows from $y$ to $x$
Example 2

• Command is

\[
\text{if } x = 1 \text{ then } y := 0 \text{ else } y := 1;
\]

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:
  \[
  \text{if } x = 1 \text{ then } y := 0 \text{ else } y := 1;
  \]

• So must look for implicit flows of information to analyze program
Notation

• $x$ means class of $x$
  • In Bell-LaPadula based system, same as “label of security compartment to which $x$ belongs”

• $x \leq y$ 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 $x$ can flow into class $y$”
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 \( y := b; \)

• Information flows from \( x \) and \( a \) to \( y \), or from \( x \) and \( b \) to \( y \)

• Certified only if \( x \leq y \) and \( a \leq y \) and \( b \leq y \)
  • Note flows for both branches must be true unless compiler can determine that one branch will never be taken
Declarations

• Notation:

\[ x: \text{int class } \{ A, B \} \]

means \( x \) is an integer variable with security class at least \( \text{lub}\{ A, B \} \), so \( \text{lub}\{ A, B \} \leq x \)

• 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

\[ i_p: \text{type class} \{ i_p \} \]
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:

  \[ o_p: \text{type class} \{ r_1, \ldots, r_n \} \]

where \( r_i \) is class of \( i \)th input or input/output argument
Example

```plaintext
proc sum(x: int class { A };
    var out: int class { A, B });
begin
    out := out + x;
end;
• Require $x \leq out$ and $out \leq out$
```
Array Elements

• Information flowing out:

\[ \ldots := a[i] \]

Value of \( i \), \( a[i] \) both affect result, so class is \( \text{lub\{ } a[i], i \} \)

• Information flowing in:

\[ a[i] := \ldots \]

• Only value of \( a[i] \) affected, so class is \( a[i] \)
Assignment Statements

\( x := y + z; \)

- Information flows from \( y, z \) to \( x \), so this requires \( \text{lub}\{ y, z \} \leq x \)

More generally:

\( y := f(x_1, \ldots, x_n) \)

- the relation \( \text{lub}\{ x_1, \ldots, x_n \} \leq y \) must hold
Compound Statements

\[ x := y + z; \ a := b * c - x; \]

• First statement: \( \text{lub}\{ y, z \} \leq x \)
• Second statement: \( \text{lub}\{ b, c, x \} \leq a \)
• 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 $d := b \cdot c - x$; end

• Statement executed reveals information about $x$, $y$, $z$, so $\text{lub}\{x, y, z\} \leq \text{glb}\{a, d\}$

More generally:

if $f(x_1, \ldots, x_n)$ then $S_1$ else $S_2$; end

• $S_1, S_2$ must be secure
• $\text{lub}\{x_1, \ldots, x_n\} \leq \text{glb}\{y \mid y \text{ target of assignment in } S_1, S_2\}$
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 \( f(x_1, ..., x_n) \) do \( S; \)

• Loop must terminate;
• \( S \) must be secure
• \( \text{lub}\{ x_1, ..., x_n \} \leq \text{glb}\{ y \mid y \text{ target of assignment in } S \} \)
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

\[
\text{proc } tm(x: \text{array}[1..10][1..10] \text{ of integer class } \{x\}; \\
\text{var } y: \text{array}[1..10][1..10] \text{ of integer class } \{y\}); \\
\text{var } i, j: \text{integer class } \{i\}; \\
\text{begin} \\
\text{\quad } b_1 \quad i := 1; \\
\text{\quad } b_2 \text{ L2: if } i > 10 \text{ goto L7; } \\
\text{\quad } b_3 \quad j := 1; \\
\text{\quad } b_4 \text{ L4: if } j > 10 \text{ then goto L6; } \\
\text{\quad } b_5 \quad y[j][i] := x[i][j]; \quad j := j + 1; \text{ goto L4; } \\
\text{\quad } b_6 \text{ L6: } i := i + 1; \text{ goto L2; } \\
\text{\quad } b_7 \text{ L7: } \\
\text{end; }
\]
Flow of Control

\[ b_1 \rightarrow b_2 \xrightarrow{i > n} b_7 \]

\[ b_6 \rightarrow b_2 \xrightarrow{i \leq n} b_3 \]

\[ b_4 \xrightarrow{j > n} b_6 \]

\[ b_5 \xrightarrow{j \leq n} b_4 \]
Immediate Forward Dominators

• 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($b_2$) = $b_7$ $b_2$→$b_7$ or $b_2$→$b_3$→$b_6$→$b_2$→$b_7$
  • IFD($b_3$) = $b_4$ one path
  • IFD($b_4$) = $b_6$ $b_4$→$b_6$ or $b_4$→$b_5$→$b_6$
  • 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 $\text{IFD}(b_i)$
  • 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
  • $\text{lub}\{ x_{i1}, \ldots, x_{in} \} \leq \text{glb}\{ y \mid y \text{ target of assignment in } B_i \}$
Example of Requirements

\[ \text{lub}\{\text{Low, } i\} \leq i \]

\[ \text{Lub}\{ x[i][j], i, j \} \leq y[j][i] \}; \text{lub}\{\text{Low, } j\} \leq j \]

\( b_1 : i := 1; \)
\( b_2 \text{ L2: if } i > 10 \text{ goto L7;} \)
\( b_3 : j := 1; \)
\( b_4 \text{ L4: if } j > 10 \text{ then goto L6;} \)
\( b_5 : y[j][i] := x[i][j]; \)
\( j := j + 1; \text{ goto L4;} \)
\( b_6 \text{ L6: } i := i + 1; \text{ goto L2;} \)
\( b_7 \text{ L7: } \)
Example of Requirements

• Within each basic block:
  \[ b_1: \text{Low} \leq i \quad b_3: \text{Low} \leq j \quad b_6: \text{lub}\{\text{Low}, j\} \leq i \]
  \[ b_5: \text{lub}\{x[i][j], i, j\} \leq y[j][i]\}; \text{lub}\{\text{Low}, i\} \leq i \]
  • Combining, \( \text{lub}\{x[i][j], i, j\} \leq y[j][i]\}
  • From declarations, true when \( \text{lub}\{x, i\} \leq y \)

• \( B_2 = \{b_3, b_4, b_5, b_6\} \)
  • Assignments to \( i, j, y[j][i] \); conditional is \( i \leq 10 \)
  • Requires \( i \leq \text{glb}\{i, j, y[j][i]\} \)
  • From declarations, true when \( i \leq y \)
Example (continued)

• $B_4 = \{ b_5 \}$
  • Assignments to $j, y[j][i]$; conditional is $j \leq 10$
  • Requires $j \leq \text{glb}\{ j, y[j][i] \}$
  • From declarations, means $i \leq y$

• Result:
  • Combine $\text{lub}\{ x, i \} \leq y; i \leq y; i \leq y$
  • Requirement is $\text{lub}\{ x, i \} \leq y$
Procedure Calls

\( tm(a, b); \)

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

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

More generally:

\[
\text{proc } pn(i_1, \ldots, i_m : \text{int}; \text{ var } o_1, \ldots, o_n : \text{int}); \text{ begin } S \text{ end;}
\]

- \( S \) must be secure
- For all \( j \) and \( k \), if \( i_j \leq o_k \), then \( x_j \leq y_k \)
- For all \( j \) and \( k \), if \( o_j \leq o_k \), then \( y_j \leq y_k \)
Exceptions

```plaintext
proc copy(x: integer class { x });
    var y: integer class Low;
var sum: integer 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 sum is MAXINT/y
  • Information flows from y to sum, but sum ≤ y never checked

• Need to handle exceptions explicitly
  • Idea: on integer overflow, terminate loop
    ```
    on integer_overflow_exception sum do z := 1;
    ```
  • Now information flows from sum to z, meaning sum ≤ z
  • This is false (sum = { x } dominates z = Low)
Infinite Loops

\textbf{proc} \textit{copy}(x: integer 0..1 class \{ x \});
\[
\text{var } y : \text{integer } 0..1 \text{ class Low};
\]
\textbf{begin}
\[
y := 0;
\]
\quad \textbf{while} \ x = 0 \ 	extbf{do}
\quad \text{(* nothing *)};
\quad y := 1;
\]
\textbf{end}

\begin{itemize}
  \item If \( x = 0 \) initially, infinite loop
  \item If \( x = 1 \) initially, terminates with \( y \) set to 1
  \item No explicit flows, but implicit flow from \( x \) to \( y \)
\end{itemize}
Semaphores

Use these constructs:

\[
\text{wait}(x): \quad \text{if } x = 0 \text{ then block until } x > 0; \quad x := x - 1;
\]

\[
\text{signal}(x): \quad x := x + 1;
\]

- \(x\) is semaphore, a shared variable
- Both executed atomically

Consider statement

\[
\text{wait}(sem); \quad x := x + 1;
\]

- 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*
  • $\text{shared}(S)$: set of shared variables read
    • Idea: information flows out of variables in $\text{shared}(S)$
  • $\text{fglb}(S)$: glb of assignment targets *following* $S$
  • So, requirement is $\text{shared}(S) \leq \text{fglb}(S)$

• begin $S_1$; ... $S_n$ end
  • All $S_i$ must be secure
  • For all $i$, $\text{shared}(S_i) \leq \text{fglb}(S_i)$
Example

begin

\[ x := y + z; \quad (* S_1 * ) \]
\[ \text{wait}(sem); \quad (* S_2 * ) \]
\[ a := b * c - x; \quad (* S_3 * ) \]

end

• Requirements:
  • \( \text{lub}\{ y, z \} \leq x \)
  • \( \text{lub}\{ b, c, x \} \leq a \)
  • \( \text{sem} \leq a \)
    • Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = \text{sem} \)
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 $S_1, ..., S_n$ in loop secure
  - $\text{lub}\{ \text{shared}(S_1), ..., \text{shared}(S_n) \} \leq \text{glb}(t_1, ..., t_m)$
    - Where $t_1, ..., t_m$ 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\{ $i$, item \} $\leq a[i]$
  • $S_2$ secure if sem $\leq i$ and sem $\leq a[i]$
  • $S_3$ trivially secure
cobegin/coend

cobegin

\[
x := y + z; \quad (* S_1 \*)
\]

\[
a := b \times c - y; \quad (* S_2 \*)
\]

coend

• No information flow among statements
  • For \( S_1 \), \( \text{lub}\{y, z\} \leq x \)
  • For \( S_2 \), \( \text{lub}\{b, c, y\} \leq a \)

• Security requirement is both must hold
  • So this is secure if \( \text{lub}\{y, z\} \leq x \land \text{lub}\{b, c, y\} \leq a \)
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, $x \leq y$
    
    ```
    if x = 1 then y := a;
    ```
  • When $x \neq 1$, $x = \text{High}$, $y = \text{Low}$, $a = \text{Low}$, 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 $x$ onto program stack
- $pop(x, x)$: pop top value and security class from program stack, assign them to variable $x$ and its security class $x$ respectively
Instructions

• $x := x + 1$ (increment)
  • Same as:
    \[
    \text{if } PC \leq x \text{ then } x := x + 1 \text{ else skip}
    \]

• \textbf{if} $x = 0$ \textbf{then goto} $n$ \textbf{else} $x := x - 1$ (branch and save PC on stack)
  • Same as:
    \[
    \text{if } x = 0 \text{ then begin}
    \text{push}(PC, PC); \text{ PC } := \text{lub}\{PC, x\}; \text{ PC } := n;
    \text{end else if } PC \leq x \text{ then}
    x := x - 1
    \text{else}
    \text{skip;}
    \]
More Instructions

• \( \text{if' } x = 0 \text{ then goto } n \text{ else } x := x - 1 \) (branch without saving PC on stack)
  • Same as:
    \[
    \text{if } x = 0 \text{ then }
    \begin{align*}
    \text{if } x \leq PC \text{ then } PC := n \text{ else skip} \\
    \text{else}
    \begin{align*}
    \text{if } PC \leq x \text{ then } x := x - 1 \text{ else skip}
    \end{align*}
    \end{align*}
    \]
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
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