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 $\overline{x}$, $\overline{y}$ as well as values
  – If $\overline{x} = A$ and $\overline{y} = B$, 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:
  - $0 \leq y \leq 7$, 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
  $H(y_s | x_t) = -3(1/3) \lg (1/3) = \lg 3$
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

• $x$ 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 $y$”
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 $I = (SC_I, \leq_I, \text{join}_I)$:
  - $SC_I$ set of security classes
  - $\leq_I$ ordering relation on elements of $SC_I$
  - $\text{join}_I$ function to combine two elements of $SC_I$
- Example: Bell-LaPadula Model
  - $SC_I$ set of security compartments
  - $\leq_I$ ordering relation $dom$
  - $\text{join}_I$ 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 $x$ currently
- Note transitivity *not* required
- If information can flow from $a$ to $b$, then $b$ dominates $a$ under ordering of policy $I$:
  \[(\forall a, b \in O)\left[ a \rightarrow b \Rightarrow a_L \leq_I b_U \right] \]
Example 1

• \( SC_I = \{ U, C, S, TS \} \), with \( U \leq_I C \), \( C \leq_I S \), and \( S \leq_I TS \)

• \( a, b, c \in O \)
  – \( \text{confine}(a) = [ C, C ] \)
  – \( \text{confine}(b) = [ S, S ] \)
  – \( \text{confine}(c) = [ TS, TS ] \)

• 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
• \( x, y, z \in O \)
  – \( \text{confine}(x) = [C, C] \)
  – \( \text{confine}(y) = [S, S] \)
  – \( \text{confine}(z) = [C, TS] \)
• 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 …

- \( \forall x \in S_Q: \text{ let } f(x) = \{ y \mid y \in S_Q \land y \leq_Q x \} \)
  - Define \( S_{QP} = \{ f(x) \mid x \in S_Q \} \)
  - Define \( \leq_{QP} = \{ (x, y) \mid x, y \in S_Q \land x \subseteq y \} \)
    - \( S_{QP} \) 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
  - \( S_{QP}' = S_{QP} \cup \{ S_Q, \emptyset \} \)
  - Upper bound \( \text{ub}(x, y) = \{ z \mid z \in S_{QP} \land x \subseteq z \land y \subseteq z \} \)
  - Least upper bound \( \text{lub}(x, y) = \cap \text{ub}(x, y) \)
    - 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
  - \( \text{confine}(\text{PRO}) = \{ \text{public, analysis} \} \)
  - \( \text{confine}(A) = \{ \text{analysis, top-level} \} \)
  - \( \text{confine}(S) = \{ \text{covert, top-level} \} \)
Information Flow

• By confinement flow model:
  – $\text{PRO} \leq A$, $A \leq \text{PRO}$
  – $\text{PRO} \leq S$
  – $A \leq S$, $S \leq A$

• Data cannot flow to public relations officers; not transitive
  – $S \leq A$, $A \leq \text{PRO}$
  – $S \leq \text{PRO}$ 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 \( l_R, h_R: SC_R \rightarrow S_P \)
    - \( l_R(x) = \{ x \} \)
    - \( h_R(x) = \{ y \mid y \in SC_R \land y \leq_R x \} \)
  - \( S_P \) contains subsets of \( SC_R; \leq_P \) subset relation
  - Dual mapping function order preserving iff
    \[
    (\forall a, b \in SC_R)[ a \leq_R b \iff l_R(a) \leq_P h_R(b) ]
    \]
Theorem

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

Proof sketch: all notation as before

$(\Rightarrow)$ 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 $l_R(a) \leq_P h_R(b)$

$(\Leftarrow)$ 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 \( \text{confine}(x) = \{ x_L, x_U \} \), consider class \( y \)
  - Information can flow from \( x \) to element of \( y \) iff 
    \( x_L \leq_R y \), or \( l_R(x_L) \subseteq h_R(y) \)
  - Information can flow from element of \( y \) to \( x \) iff 
    \( y \leq_R x_U \), or \( l_R(y) \subseteq h_R(x_U) \)
Revisit Government Example

• Information flow policy is $R$
• Flow relationships among classes are:

  public $\leq_R$ public
  public $\leq_R$ analysis    analysis $\leq_R$ analysis
  public $\leq_R$ covert     covert $\leq_R$ covert
  public $\leq_R$ top-level  covert $\leq_R$ top-level
  analysis $\leq_R$ 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:
  - $\text{confine}(p) = \{ \{ \text{public} \}, \{ \text{public, analysis} \} \}$
  - $\text{confine}(a) = \{ \{ \text{analysis} \}, \{ \text{public, analysis, covert, top-level} \} \}$
  - $\text{confine}(s) = \{ \{ \text{covert} \}, \{ \text{public, analysis, covert, top-level} \} \}$
And the Flow Relations Are …

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

```plaintext
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 \( x \leq y \) and \( a \leq y \) and \( b \leq y \)
  – Note flows for \textit{both} branches must be true unless compiler can determine that one branch will \textit{never} be taken
```
Declarations

• Notation:

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

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

• Distinguished classes \( \text{Low, High} \)
  – Constants are always \( \text{Low} \)
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:
  \[ O_p: \textit{type class} \{ r_1, \ldots, r_n \} \]
  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:
  
  \[ ... := a[i] \]

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

- Information flowing in:

  \[ a[i] := ... \]

- 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

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

- 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 \( d := b \times c - x \); end

- The 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 \( f(x_1, \ldots, x_n) \) do \( S; \)
- Loop must terminate;
- \( S \) must be secure
- \( \text{lub}\{x_1, \ldots, x_n\} \leq \text{glb}\{y \mid y \text{ target of assignment in } S\} \)
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 $f(x_1, \ldots, x_n)$ do $S$;

- Loop must terminate;
- $S$ must be secure
- $\text{lub}\{x_1, \ldots, x_n\} \leq \text{glb}\{y \mid y \text{ target of assignment in } S\}$
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

```plaintext
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

\[ i > n \]
\[ i \leq n \]
\[ j > n \]
\[ j \leq n \]
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
  – $\text{lub}\{ x_{i1}, \ldots, x_{in} \} \leq \text{glb}\{ y | y \text{ target of assignment in } B_i \}$
Example of Requirements

- Within each basic block:
  \[ b_1: Low \leq i \quad b_3: Low \leq j \quad b_6: \text{lub}\{Low, i\} \leq i \]
  \[ b_5: \text{lub}\{x[i][j], i, j\} \leq y[j][i]\}; \text{lub}\{Low, j\} \leq j \]
  - 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, lub{ x, i } ≤ y must hold

• In call, x corresponds to a, y to b
• Means that lub{ a, i } ≤ b, or a ≤ 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 iₗ ≤ oₖ, then xₗ ≤ yₖ
• For all j and k, if oₗ ≤ oₖ, then yₗ ≤ yₖ
Exceptions

```plaintext
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 $\text{MAXINT}/y$
  - Info flows from $y$ to $x$, but $x \leq y$ 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 $sum \leq z$
  - This is false ($sum = \{ x \}$ dominates $z = \text{Low}$)
Infinite Loops

```plaintext
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 $i$, shared($S_i$) ≤ fglb($S_i$)
Example

\begin{verbatim}
begin
  \begin{align*}
    x &:= y + z; \quad (\ast \ S_1 \ \ast) \\
    \text{wait}(sem); \quad (\ast \ S_2 \ \ast) \\
    a &:= b \ast c - x; \quad (\ast \ S_3 \ \ast)
  \end{align*}
end
\end{verbatim}

• Requirements:
  \begin{itemize}
  \item lub\{ y, z \} \leq x
  \item lub\{ b, c, x \} \leq a
  \item sem \leq a
  \end{itemize}

• Because fglb(S_2) = 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
  – \( \text{lub}\{ \text{shared}(S_1), \ldots, \text{shared}(S_n) \} \leq \text{glb}(t_1, \ldots, t_m) \)
    • Where \( t_1, \ldots, t_m \) are variables assigned to in loop
Loop Example

while $i < n$ do begin

    $a[i] := item; \quad (*) \quad S_1 \quad (*)$
    wait(sem); \quad (*) \quad S_2 \quad (*)
    $i := i + 1; \quad (*) \quad S_3 \quad (*)$

end

• Conditions for this to be secure:
  – Loop terminates, so this condition met
  – $S_1$ secure if $\text{lub}\{ i, item \} \leq a[i]$
  – $S_2$ secure if $sem \leq i$ and $sem \leq a[i]$
  – $S_3$ trivially secure
**cobegin/coend**

```plaintext
cobegin
    x := y + z;  (* S₁ *)
    a := b * c - y;  (* S₂ *)
coend
```

- No information flow among statements
  - For $S₁$, $\text{lub}\{y, z\} \leq x$
  - For $S₂$, $\text{lub}\{b, c, y\} \leq a$

- 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, \( x \leq y \)
    
    \[
    \text{if } x = 1 \text{ then } y := a;
    \]
  - 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* means instruction not executed
- *push(x, x)* means push variable *x* and its security class *x* onto program stack
- *pop(x, x)* means 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:
    
    ```
    if $PC \leq x$ 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($PC, PC$); $PC := \text{lub}\{PC, x\}$; $PC := n$;
    end else if $PC \leq x$ 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 \( x \leq PC \) then \( PC := n \) else \( skip \)
  
  else

  if \( PC \leq x \) 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

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>$PC$</th>
<th>$PC$</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td>Low ≤ $x$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>$z$</td>
<td>(3, Low)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>$z$</td>
<td>(3, Low)</td>
<td>$PC ≤ y$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>
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 PC
  – On assignment of form $y := f(x_1, \ldots, x_n)$, $y$
    changed to lub{$x_1, \ldots, x_n$}
  – 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 };
beg
    y := 0;
    z := 0;
    if x = 0 then z := 1;
    if z = 0 then y := 1;
end;

• z changes when z assigned to

• Assume y < x
Analysis of Example

- $x = 0$
  - $z := 0$ sets $z$ to Low
  - if $x = 0$ then $z := 1$ sets $z$ to 1 and $z$ to $x$
  - So on exit, $y = 0$

- $x = 1$
  - $z := 0$ sets $z$ to Low
  - if $z = 0$ then $y := 1$ sets $y$ to 1 and checks that $\text{lub}\{\text{Low}, z\} \leq y$
  - So on exit, $y = 1$

- Information flowed from $x$ to $y$ even though $y < x$
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 \( z \leq y \), fails, as \( z = x \) from previous statement, and \( y \leq x \)
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 $z$ to Low then checks $x \leq z$
  – When $x = 1$, first “if” checks that $x \leq z$
  – This holds if and only if $x = \text{Low}$
    • Not possible as $y < x = \text{Low}$ 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

- host
- SPI
- first disk
- second disk
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
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