A Lattice Model of Secure Information Flow

Dorothy E. Denning
Purdue University - 1976

Presented by Adam Fuchs

Time Machine...

1969 - Arpanet begins, First ATM
1971 - Micro Processor, Floppy Disk
1972 - Pocket Calculator
1973 - Micro Computer Commercially Available
1974 - First use of "Internet"
1975 - **This paper written**
1976 - CRAY-1, Apple I
1977 - PC MODEM, Apple II
1978 - TCP/IPv4

Secure Information Flow

"...no unauthorized flow of information is possible."

Good for securing:

- Classified computer systems
- Multi-user systems
- Inter-process communication (collaborating users)
- Inadvertent information leaks

Flow Model

$FM = \langle N, P, SC, ☺, \rightarrow \rangle$

N = {a,b,...}
Storage Objects (Inputs, Outputs, Program Variables)

P = {p,q,...}
Processes

SC = {A,B,...}
Security Classes (Unclassified, Secret, etc.)

☺ : Class Combining Operator
(SC x SC) → SC

→ : Flow Relation
e.g. Public → Private, but not Private → Public

Security Class Binding

Object a has security class binding <u>a</u>

Static Binding: <u>a</u> remains constant
- Particularly useful for input/output channels

Dynamic Binding: <u>a</u> changes depending on its contents
- Particularly useful for program variables

Class-Combining Operator ☺

Private ☺ Public = Public
Unclassified ☺ Secret = Secret

f(a,b,c) has security class <u>a</u>☺<u>b</u>☺<u>c</u>

SC is closed under ☺

Flow Relation → and Security Requirements

Public → Private is defined
Private → Public is not

"A flow model *FM* is *secure* if and only if execution of a sequence of operations cannot give rise to a flow that violates the relation '→'."

If we have a statement d = f(a,b,c), then $\underline{a}$☺$\underline{b}$☺$\underline{c}\rightarrow\underline{d}$ must hold.

Transitivity implies that local security implies overall security.

Lattice Structure

A lattice structure has:

- A partially ordered set of elements S
- A least upper bound defined on S x S
- A greatest lower bound defined on S x S
- A top element that is an upper bound for all elements in S
- A bottom element that is a lower bound for all elements in S

Fig. 1. Linear ordered lattice.

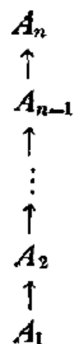
$SC = \{A_1, \ldots, A_n\}$

$A_i \rightarrow A_j$ iff $i \leq j$

$A_i \oplus A_j \equiv A_{\max(i,j)}$

$A_i \otimes A_j \equiv A_{\min(i,j)}$

$L = A_1$; $H = A_n$



Description | Representation

Fig. 2. Lattice of subsets of $X = \{x, y, z\}$.

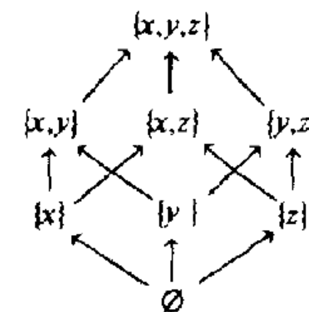
SC = powerset (X)

$A \rightarrow B$ iff $A \subseteq B$

$A \oplus B \equiv A \cup B$

$A \otimes B \equiv A \cap B$

$L = \emptyset$; $H = X$



Description | Representation

Lattice Applied to Information Flow

Four assumptions show that ‹*SC* , ☺ ,→› forms a lattice:

1. ‹*SC* ,→› is a partially ordered set.
2. *SC* is finite.
3. *SC* has a lower bound *L* such that $L \rightarrow A$ for all *A* in *SC.*
4. ☺ is the least upper bound operator on *SC* .

Properties of →:

1. Reflexive: $A \rightarrow A$
2. Transitive: $A \rightarrow B$ and $B \rightarrow C$ implies $A \rightarrow C$
3. Antisymmetric: $A \rightarrow B$ and $B \rightarrow A$ implies $A = B$

Explicit vs. Implicit Flow

Explicit: a := b, so <u>b</u> → <u>a</u>

.

Implicit: **if** a = 0 **then** b := c, so <u>a</u> → <u>b</u>

Abstract Program Representation

An abstract program (or statement) S is defined recursively by:

1. S is an elementary statement; e.g. assignment or I/O
2. There exist S_1 and S_2 such that $S = S_1 ; S_2$ (Sequence)
3. There exist $S_1, ..., S_m$ and an m-valued variable c such that $S = c : S_1, ..., S_m$ (Conditional)

while c **do** S and **if** c **then** S are both represented by $c : S$

Security Requirements for Abstract Program

1. Explicit flow caused by an elementary statement must be secure.
2. Each part of a sequence statement must be secure.
3. Each part of a conditional statement must be secure, and any implicit flow in a conditional statement must be secure.

Consider the statement $S = c : S_1, \ldots, S_m$. Let $b_1, \ldots, b_n$ be the objects that receive explicit flow in S_1 through S_m. The implicit flow in S is $\underline{c} \rightarrow \underline{b_i}$, $1 \leq i \leq n$.

Static Binding - Case and MITRE Systems

- Each process p has a static binding $\underline{p}$.
- p can read from object a IFF $\underline{a} \rightarrow \underline{p}$.
- p can write to object b IFF $\underline{p} \rightarrow \underline{b}$.
- Transitivity guarantees that $\underline{a} \rightarrow \underline{b}$, so implicit flows are also secure.
- Enforcement happens at run-time.

This mechanism adds restrictions to the model:
Public → Public and Private → Private data transfers must use separate processes.

Static Binding - The Data Mark Machine

- Extension to the Case and MITRE systems.
- Data is marked with its security class.
- For program p , $\underline{p}$ changes according to the control flow.
- Upon entering statements $S = c : S_1 ,..., S_m$, $\underline{p}$ is pushed onto the stack and replaced with $\underline{p}$ ☺ $\underline{c}$.
- Explicit flow statements a := ... also require that $\underline{p} \rightarrow \underline{a}$.

This is less restrictive than Case and MITRE Systems.

Static Binding - Certification Mechanism

- Information flow security is checked at compile time.
- Each statement gets the lower bound of the security classes of objects receiving explicit flow.
  - For $S = (a := b)$, $\underline{S} = \underline{a}$, and $\underline{b} \rightarrow \underline{a}$ is checked.
  - For $S = S_1;S_2$, $\underline{S} = \underline{S_1}$ ☻ $\underline{S_2}$
  - For $S = c : S_1,\ldots,S_m$, $\underline{S} = \underline{S_1}$ ☻ ... ☻ $\underline{S_m}$, and $\underline{c} \rightarrow \underline{S}$ is checked.

This constitutes early static analysis, and is sound.

Dynamic Binding Mechanisms

- Dynamic binding allows security classes to change throughout the execution of a program.
- Requires some static bindings to relate to the real world.
- Vulnerable to leaking data through covert channels.

Dynamic Binding - Dynamic Data Mark Machine

Instead of checking that a dynamically bound object can receive flow, update the security class of that object:

- The check of $\underline{a}$ ☺ $\underline{b}$ ☺ $\underline{c} \rightarrow \underline{d}$ becomes $\underline{d}$:= $\underline{a}$ ☺ $\underline{b}$ ☺ $\underline{c}$

Statically bound objects are still checked (must be at runtime).

Flaw: Implicit flows are not checked in the absence of explicit flows.

Dynamic Data Mark Machine Example

$b := c := \textit{false};$
if $\sim a$ **then** $c := \textit{true};$
if $\sim c$ **then** $b := \textit{true}$

Assuming that the constants *true* and *false* are in the least class L, execution of this program by a process p proceeds as follows:

$b := c := \textit{false}; \quad \underline{b} := \underline{c} := L;$
$\underline{p} := \underline{a};$ **if** $\sim a$ **then** $\{c := \textit{true}; \quad \underline{c} := L \oplus \underline{p}\}; \quad \underline{p} := L;$
$\underline{p} := \underline{c};$ **if** $\sim c$ **then** $\{b := \textit{true}; \quad \underline{b} := L \oplus \underline{p}\}; \quad \underline{p} := L$

Since $\underline{p} = L \oplus \underline{p}$, this simplifies to:

$b := c := \textit{false}; \quad \underline{b} := \underline{c} := L;$
if $\sim a$ **then** $\{c := \textit{true}; \quad \underline{c} := \underline{a}\};$
if $\sim c$ **then** $\{b := \textit{true}; \quad \underline{b} := \underline{c}\}$

Dynamic Binding - Nondecreasing Class Mechanisms

Security classes change monotonically:

- $b := a$ results in $\underline{b} := \underline{b}$ ☺ $\underline{a}$, even though b is overwritten
- $\underline{p}$ also does not decrease.

Still suffers from implicit flow being unchecked in the absence of explicit flow in multi-process collaborations.

Other minor flaws may exist in some implementations...

Discussion: Flaws in the Model

Dissemination and declassification are not modeled

- Voting machine can release a sum of votes
- Actions based on classified data happen at observable security classes

Application of function to data can raise security class, so ☺ may be function specific