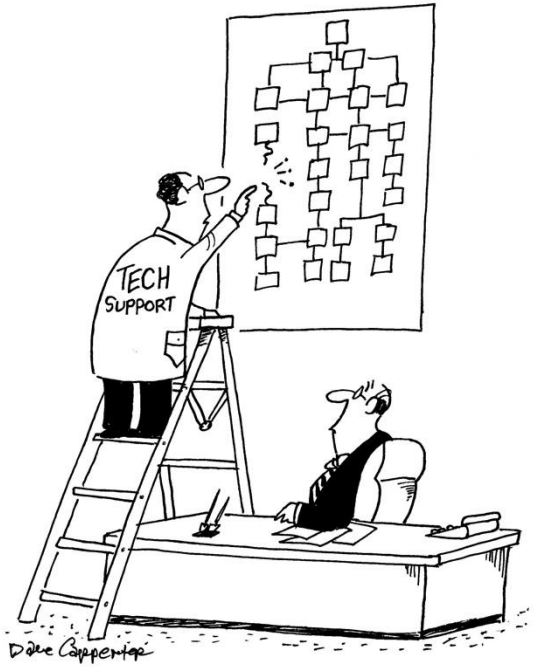
Lecture 19: Information Flow

CS 138

Spring 2026



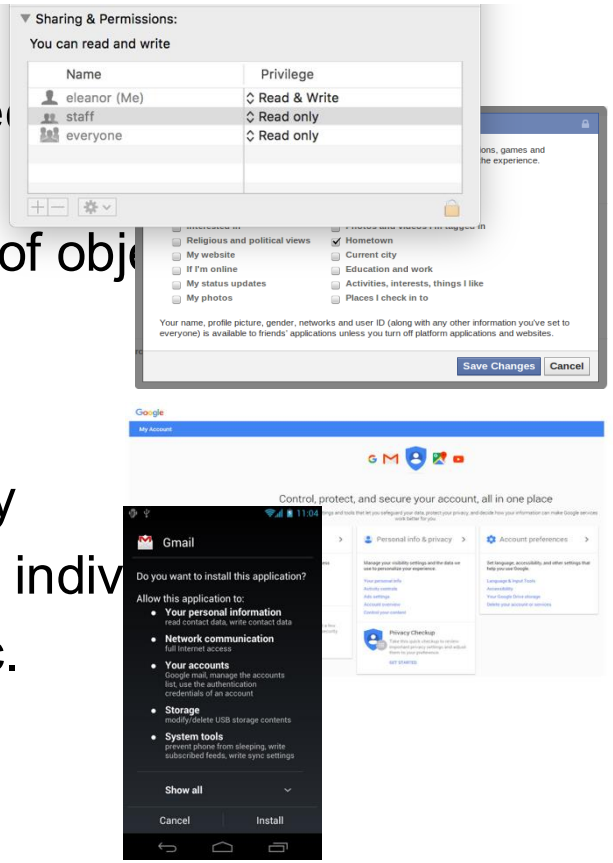
"Here's your problem."

Where we were…

- **Authentication:** mechanisms that bind principals to actions
- **Authorization:** mechanisms that govern whether actions are permitted
- **Audit:** mechanisms that record and review actions

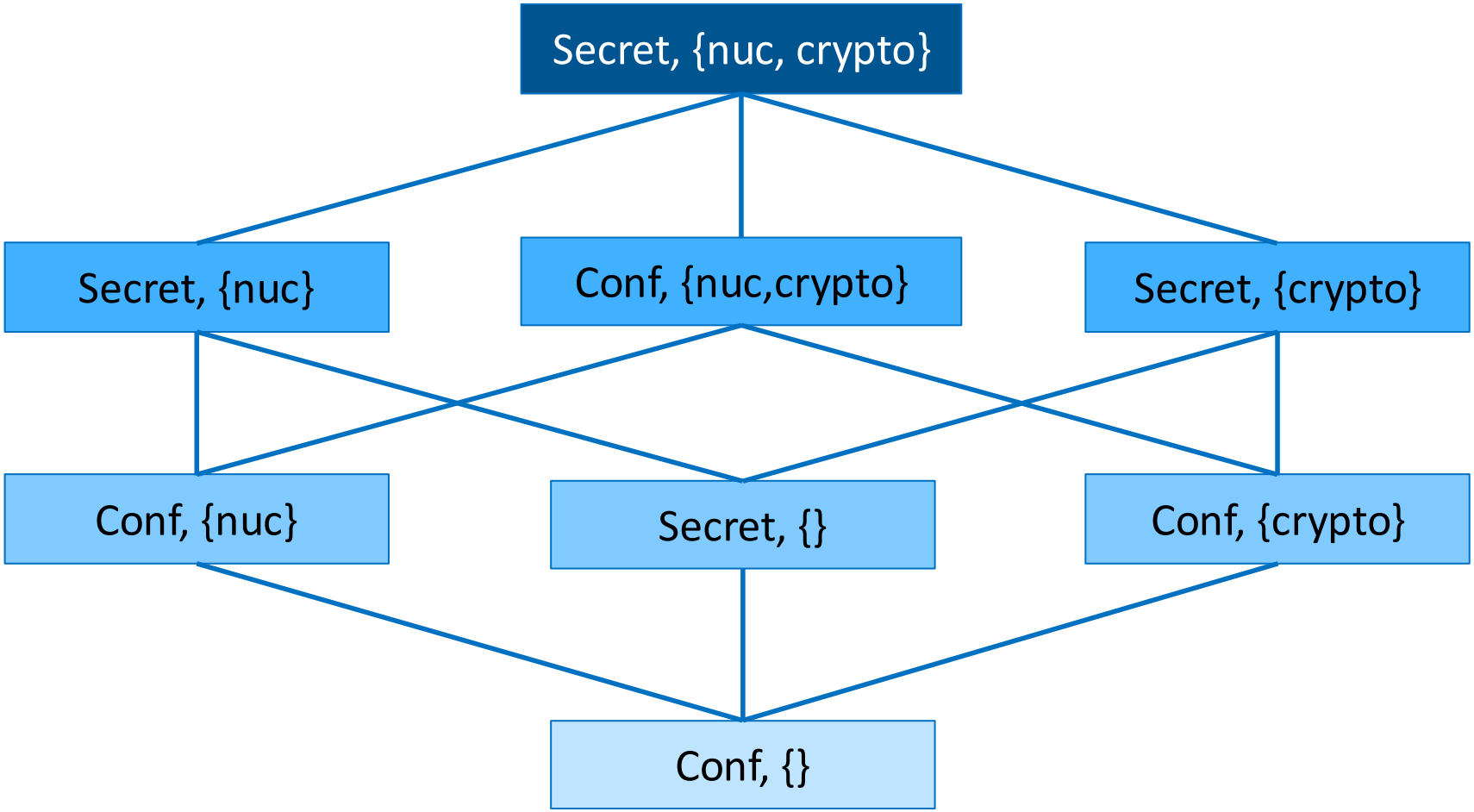
Who defines Policies?

- **Discretionary access control** (DAC)
  - **Philosophy:** users have the *discretion* to spe themselves
  - Commonly, information belongs to the **owner** of obj
  - Access control lists, privilege lists, capabilities
- **Mandatory access control** (MAC)
  - **Philosophy:** central authority *mandates* policy
  - Information belongs to the authority, not to the indiv
  - MLS and BLP, Chinese wall, Clark-Wilson, etc.

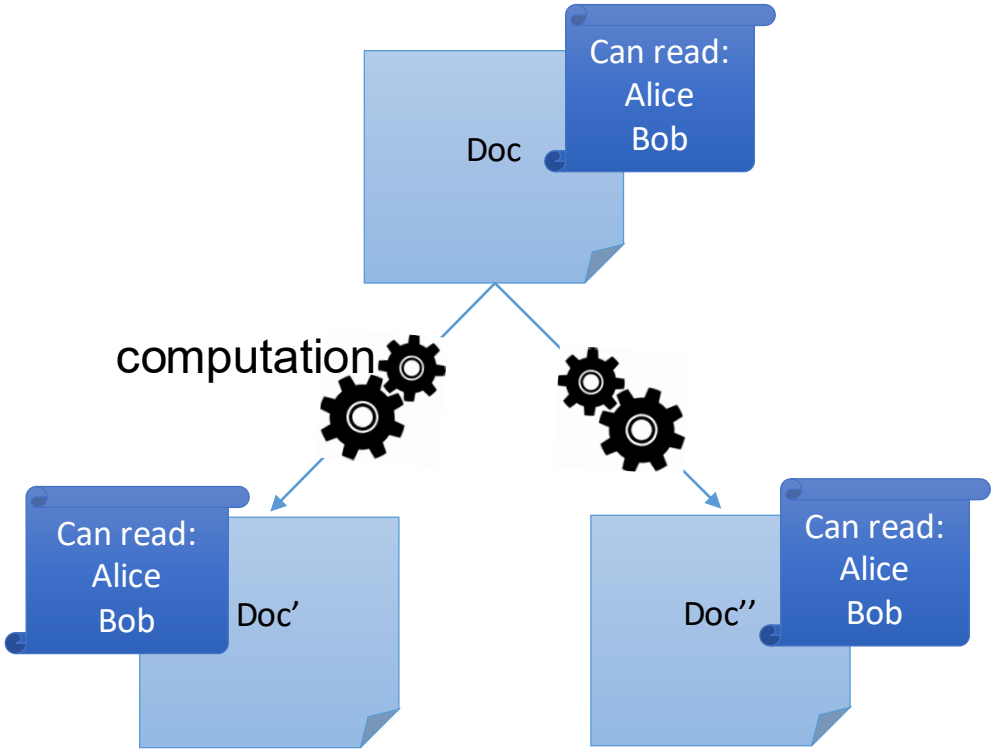


Review: Multi-Level Security

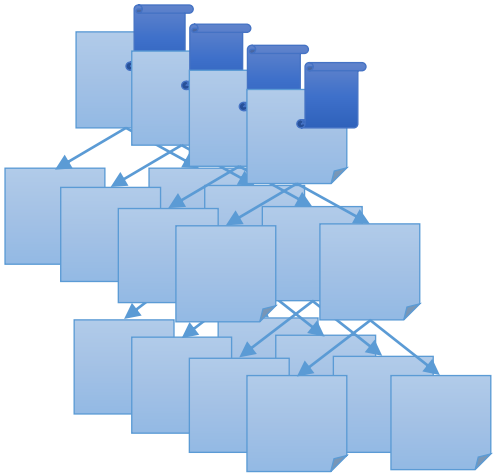
- Labels define policies



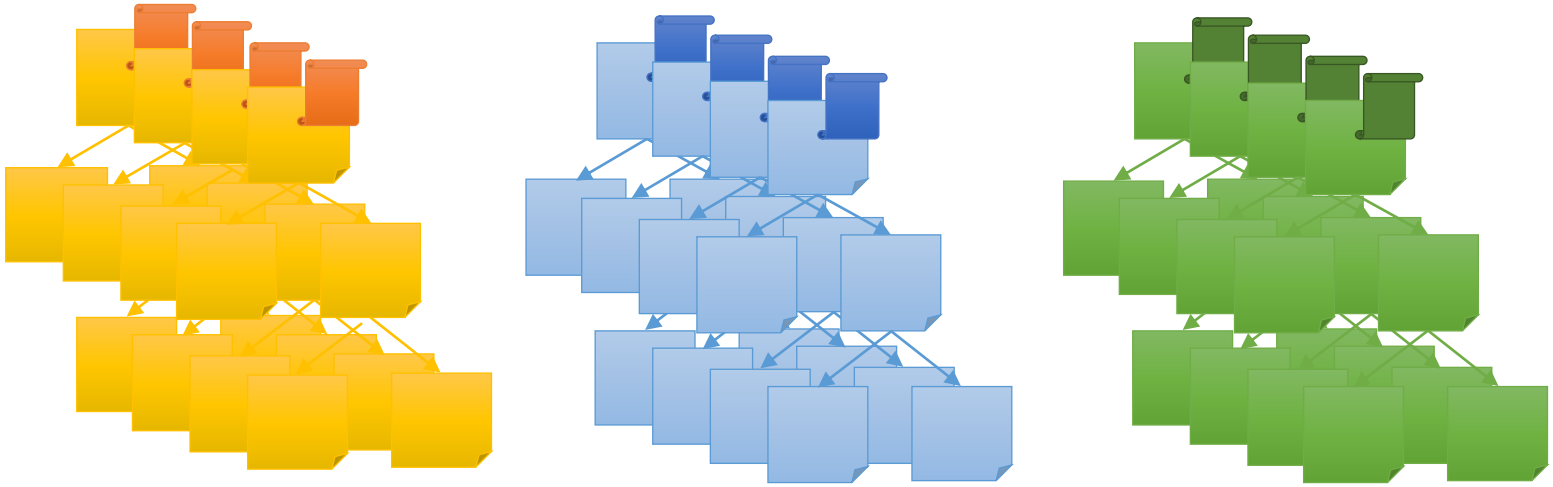
Access control for computed data



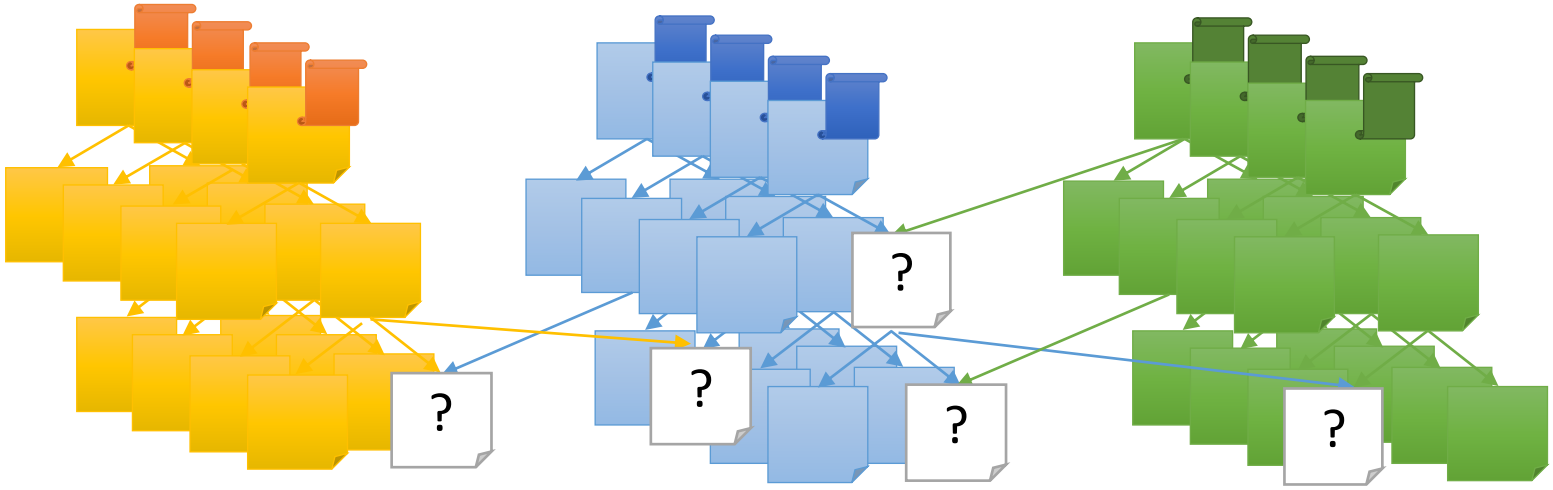
Scaling to many pieces of data…



Scaling to many users…

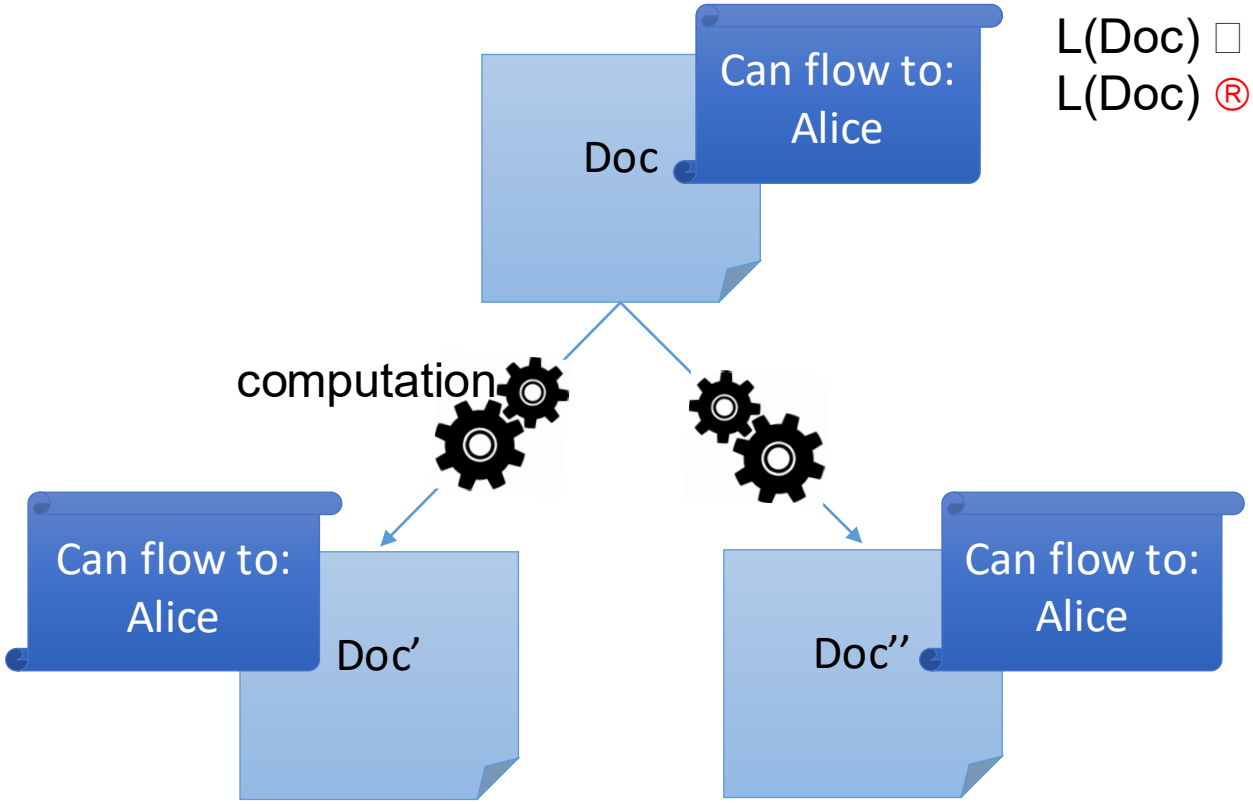


Scaling to many interactions…



Need to assign restrictions in an automatic way.

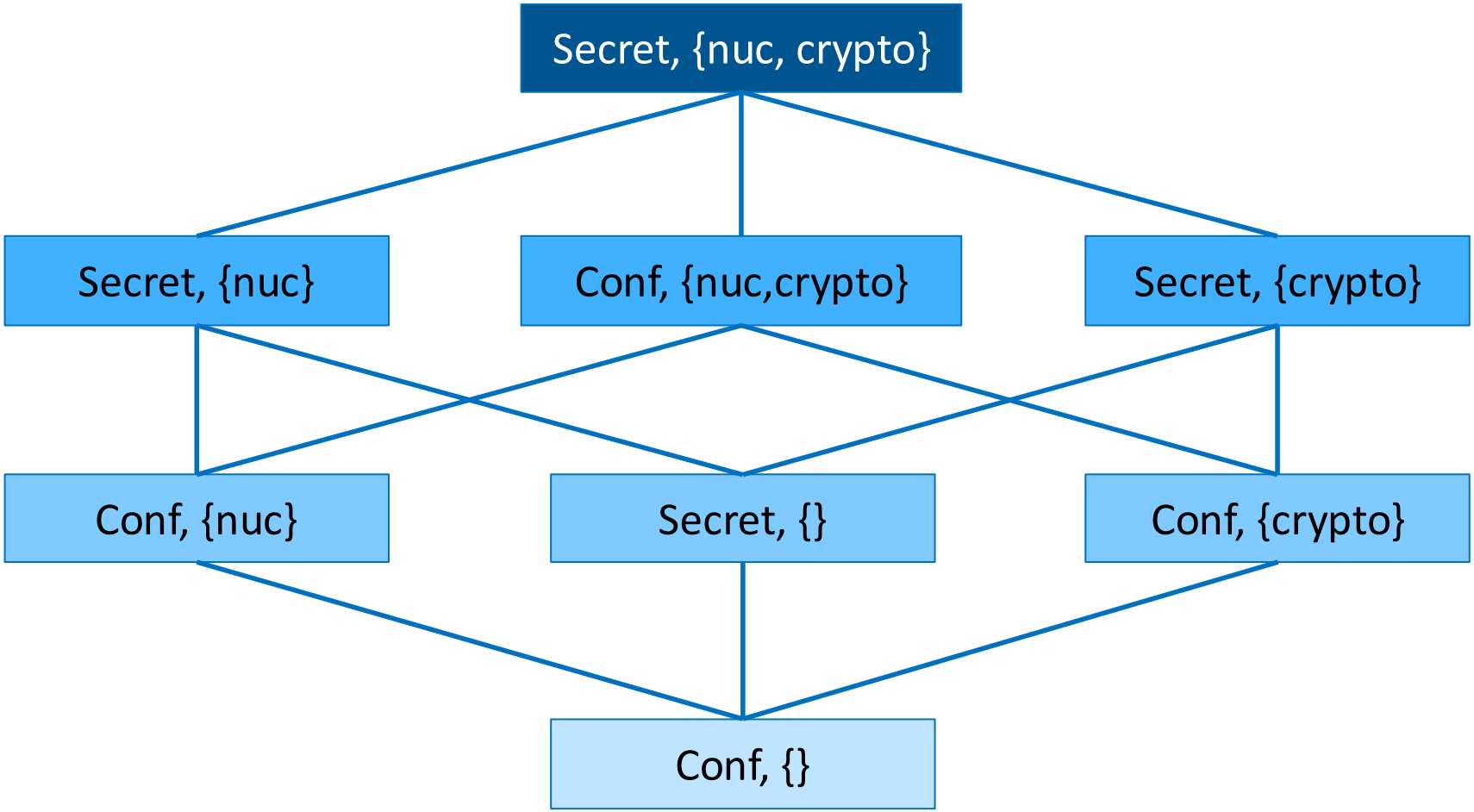
Information flow policies



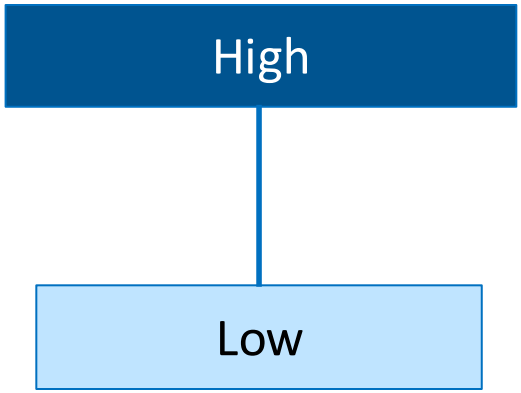
L(Doc) □ L(Alice)
L(Doc) ®L(Bob)

Automatic deduction of policies!

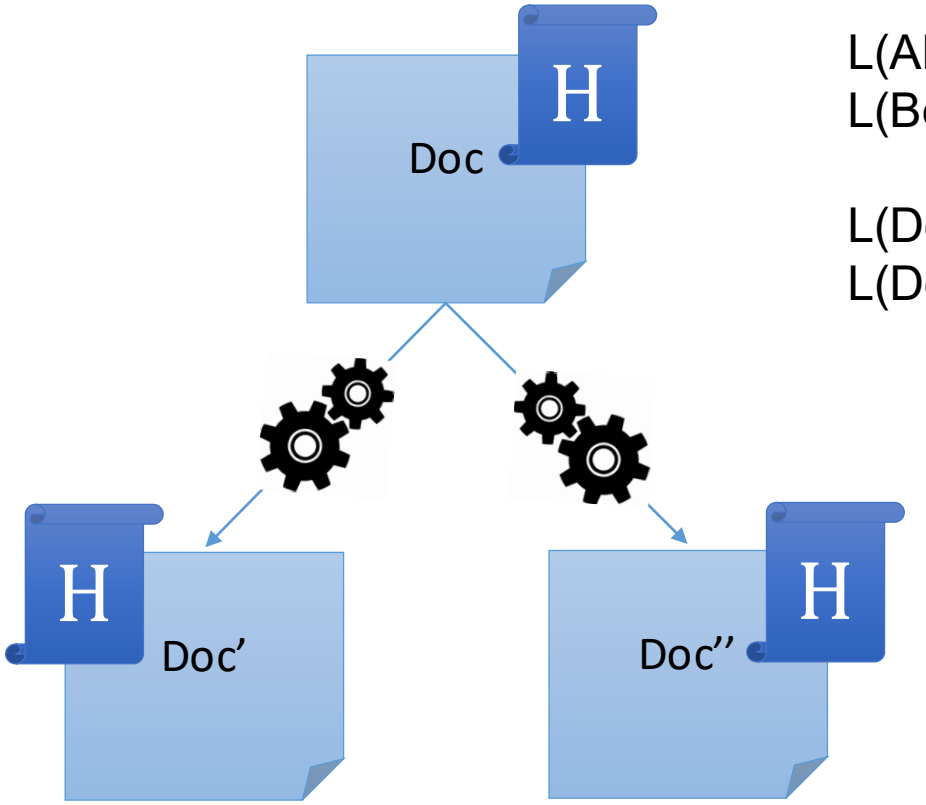
Labels represent policies



Labels represent policies



Labels represent policies



L(Alice) = H
L(Bob) = L

L(Doc) □ L(Alice)
L(Doc) ®L(Bob)

Information Flow (IF) Policies

- Focus on **information** not objects
- An IF policy specifies **restrictions** on some data, and on all its derived data.
- IF policy for confidentiality:
  - Value v *and all its derived values* are allowed to be read only by Alice
  - (Different from an access control policy, which would say something like Value v is allowed to be read only by Alice)
- The enforcement mechanism **automatically** deduces the restrictions for derived data.

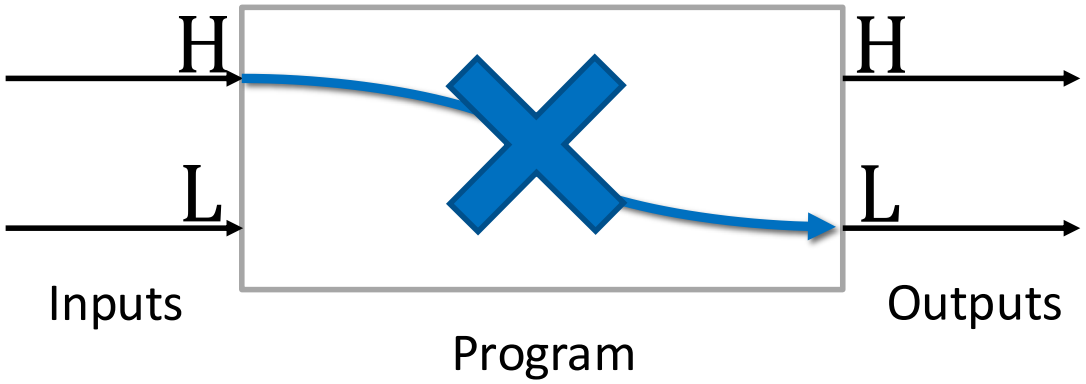
Policy Granularity

- Objects can be system principles (files, programs, sockets…)
- Objects can be program variables

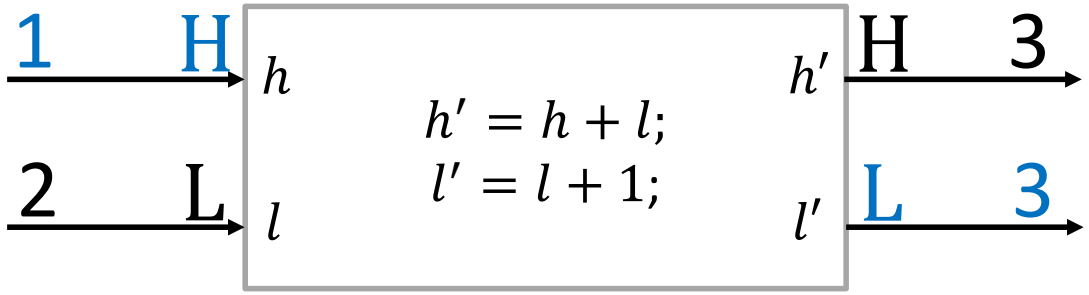
Noninterference [Goguen and Meseguer 1982]

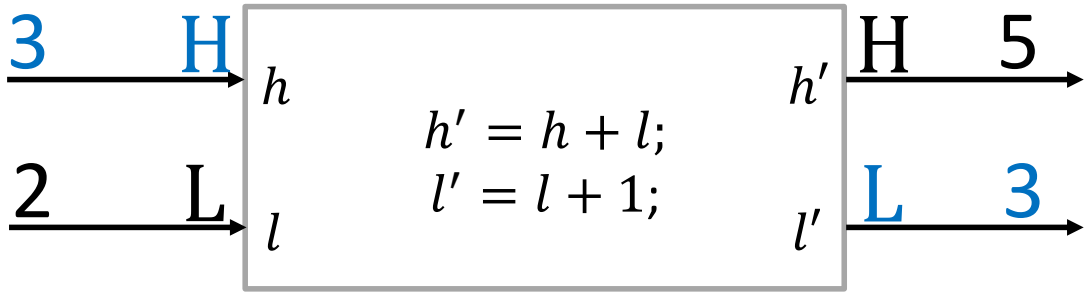
An interpretation of noninterference for a program:

- Changes on H inputs should not cause changes on L outputs.



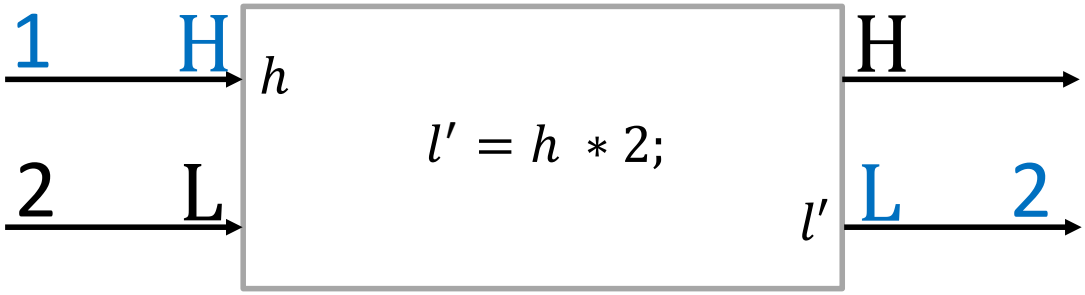
Noninterference: Example

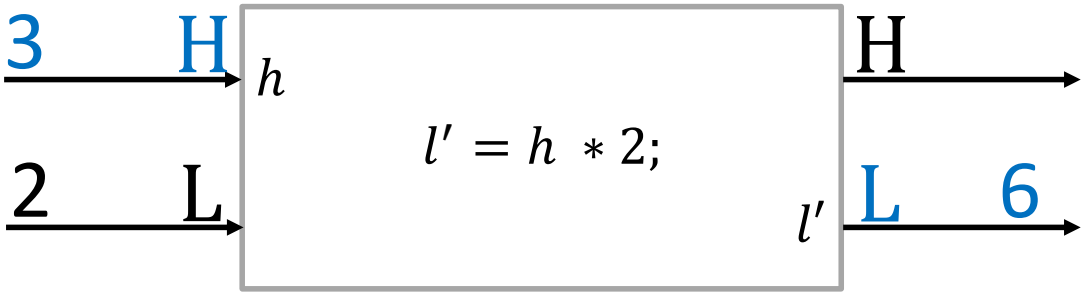




The program satisfies noninterference!

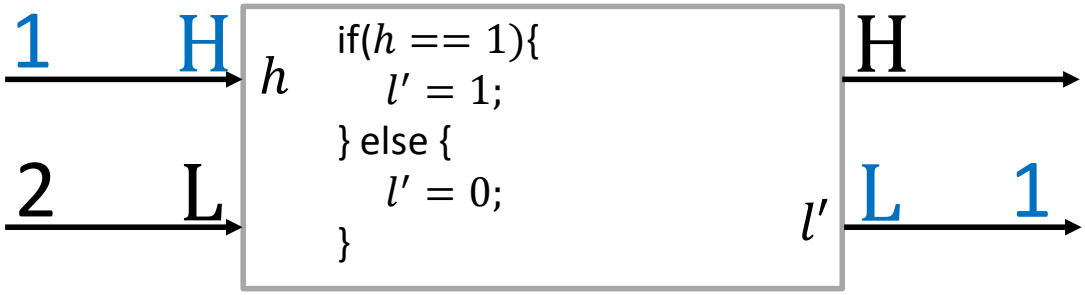
Noninterference: Example

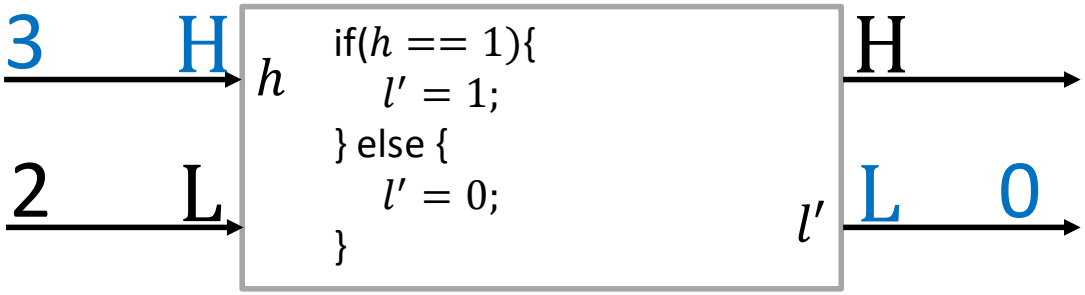




The program does not satisfy noninterference!

Noninterference: Example





The program does not satisfy noninterference!

Noninterference

- Consider a program P.
- Consider two memories M_1 and M_2, such that they agree on values of variables tagged with L: $M_1 =_{\mathrm{L}} M_2$.

  (M_1 and M_2 might not agree on values of variables tagged with H)
- $P(M_i)$ are the observations produced by executing C to termination on initial memory M_i:
  - final outputs, or
  - intermediate and final outputs.
- Then, observations tagged with L should be the same:
  - $P(M_1) =_{\mathrm{L}} P(M_2)$.

Noninterference: $\forall M_1, M_2$: if $M_1 =_{\mathrm{L}} M_2$, then $P(M_1) =_{\mathrm{L}} P(M_2)$.

Exercise 1: Noninterference

Assume P_1, P_2 each take two inputs: h_I (label H) and l_i (label L)

1. P_1 outputs (h_O, l_O) where $h_O = h_I || l_I$ and $l_O = l_I$
   - || denotes string concatenation.

2. P_2 outputs l_O where $l_o = \begin{cases} l_I & \text{if } h_I \text{ is even} \\ l_I || l_I & \text{if } h_I \text{ is odd} \end{cases}$

Enforcement Mechanisms

- Static Information Flow Control:
  - type checking
- Dynamic Information Flow Control:
  - taint-tracking
  - runtime monitoring

A simple programming language

```
e ::= n | x | e1+e2 | e1 < e2 | ...

p ::= x = e;
    | p1 p2
    | if(e) then { p1 } else { p2 }
    | while(e){ p }
    | nop;
```

Exercise: A programming language

- Using our simple programming language, write a program that takes one input x_I and ends with an output x_O that is equal to the sum of the odd numbers between 0 and x_I (inclusive)

```
e ::= n | x | e1+e2 | e1 < e2 | …

p ::= x = e; | p1 p2
    | if(e) then { p1 } else { p2}
    | while(e){ p } | nop;
```

Type Systems

- A program is well-typed if all operands are the right type for the operator and all variables are the right type for the expression

```
int x;
string y;

x = 4 + 5;
y = "hello" + "world";
```

```
x = "hello" * "world";
x = "hello" + 5;
x = "hello" + "world";
```

- determining that a program is well-typed requires proving that all expressions and all assignments are the right type

Logical Inference

- Syntax for logical Inference: $\frac{premise(s)}{conclusion}$

- Examples:

$$\frac{x=4,\ y=5}{x+y=9} \qquad \frac{x=True,\ y=False}{x\ or\ y = True} \qquad \frac{}{security\ is\ fun!}$$

Type Inference (Expressions)

```
int x;
bool y;
```

$\Gamma(x) = \mathbf{int;}$

$\Gamma(y) = \mathbf{bool;}$

- Type environment Γ maps variables to type
- Goal: Judgement (aka proof that) $\Gamma \vdash \mathbf{e} : t$
  According to mapping Γ, expression **e** has type t
- Constants: $\frac{}{\Gamma \vdash n::int}$ $\frac{}{\Gamma \vdash True::bool}$ $\frac{}{\Gamma \vdash False::bool}$
- Variables: $\frac{\Gamma(x)=t}{\Gamma \vdash x::t}$
- Expressions: $\frac{\Gamma \vdash e1::int, \Gamma \vdash e2::int}{\Gamma \vdash e1+e2::int}$ $\frac{\Gamma \vdash e1::int, \Gamma \vdash e2::int}{\Gamma \vdash e1<e2::bool}$ ...

Example: Type Inferences

- Let $\Gamma(\mathbf{x}) = \text{int}$ and $\Gamma(\mathbf{y}) = \text{int}$.
- What is the type of **x+y+1**?
- *Proof tree:*

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathbf{x}) = \mathit{int}}{\Gamma \vdash \mathbf{x} : \mathit{int}} \quad \dfrac{\Gamma(\mathbf{y}) = \mathit{int}}{\Gamma \vdash \mathbf{y} : \mathit{int}}}{\Gamma \vdash \mathbf{x} + \mathbf{y} : \mathit{int}} \quad \dfrac{}{\Gamma \vdash \mathbf{1} : \mathit{int}}}{\Gamma \vdash (\mathbf{x} + \mathbf{y}) + \mathbf{1} : \mathit{int}}$$

Exercise: Type Inference

- Let $\Gamma(\mathbf{x}) = \text{int}$ and $\Gamma(\mathbf{y}) = \text{int}$.
- What is the type of **y>x+5**?
- *Proof tree:*

$$\dfrac{\dfrac{\Gamma(\mathbf{y}) = \mathit{int}}{\Gamma \vdash \mathtt{y} : \mathit{int}} \qquad \dfrac{\dfrac{\Gamma(\mathbf{x}) = \mathit{int}}{\Gamma \vdash \mathtt{x} : \mathit{int}} \qquad \dfrac{}{\Gamma \vdash \mathtt{5} : \mathit{int}}}{\Gamma \vdash \mathtt{x + 5} : \mathit{int}}}{\Gamma \vdash \mathtt{y > x + 5} : \mathit{bool}}$$

Enforcing Information Flow

- Goal: Design a type system such that

$$\Gamma \vdash \mathtt{p} \quad \Rightarrow \quad \mathtt{p} \text{ satisfies NonInterference}$$

Type Checking (Programs)

- When is a one-line program `x = e;` well-typed?

Static type system

Assignment-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : t \qquad t \sqsubseteq \Gamma(\mathtt{x})}{\Gamma \vdash \mathtt{x=e;}}$$

Sequence-Rule:
$$\frac{\Gamma \vdash \mathtt{p1} \qquad \Gamma \vdash \mathtt{p2}}{\Gamma \vdash \mathtt{p1\ p2}}$$

If-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \mathrm{bool} \quad \Gamma \vdash \mathtt{p1} \quad \Gamma \vdash \mathtt{p2}}{\Gamma \vdash \mathtt{if(e)\ then\{\ p1\ \}\ else\{\ p2\ \}}}$$

While-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \mathrm{bool} \qquad \Gamma \vdash \mathtt{p}}{\Gamma \vdash \mathtt{while(e)\{\ p\ \}}}$$

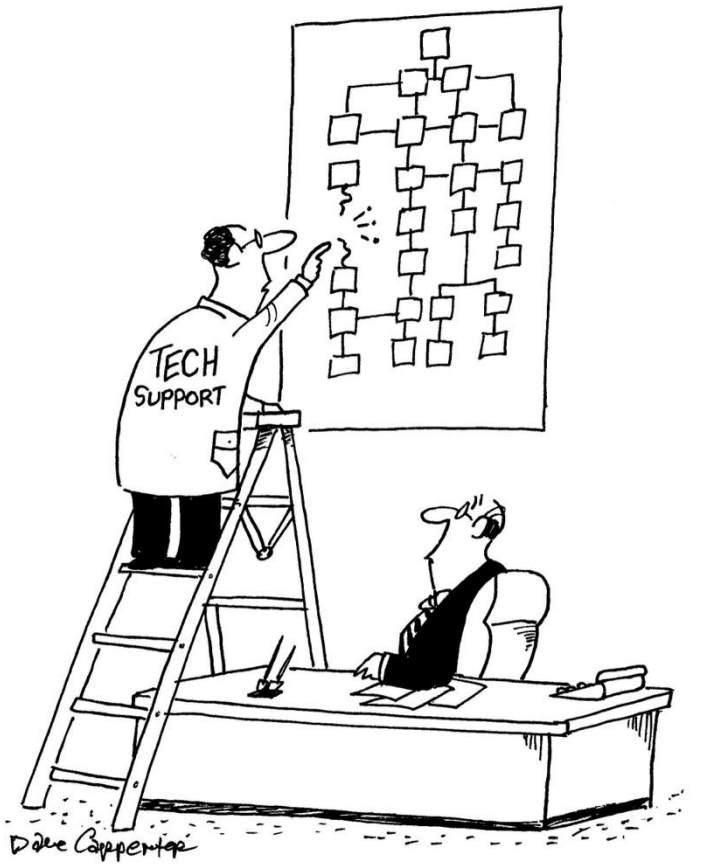
Skip-Rule:
$$\frac{}{\Gamma \vdash \mathtt{nop;}}$$

Enforcing Information Flow

- Goal: Design a type system such that

$$\Gamma \vdash \mathtt{p} \quad \Rightarrow \quad \mathtt{p} \text{ satisfies NonInterference}$$

Information Flow



"Here's your problem."