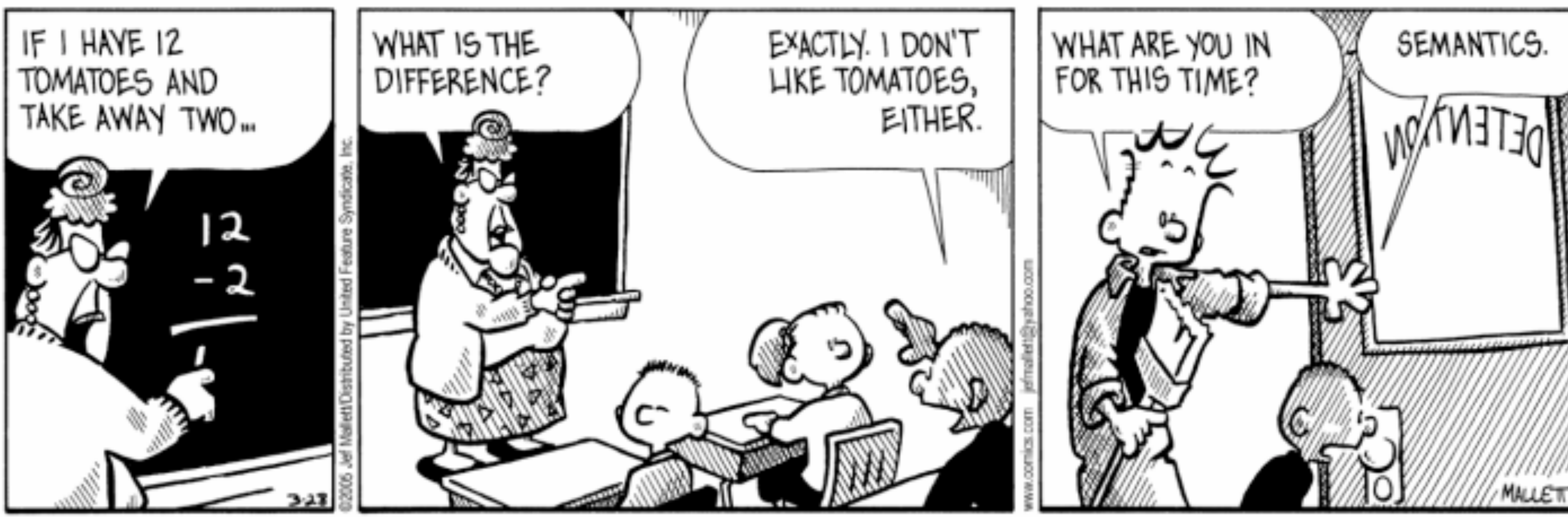
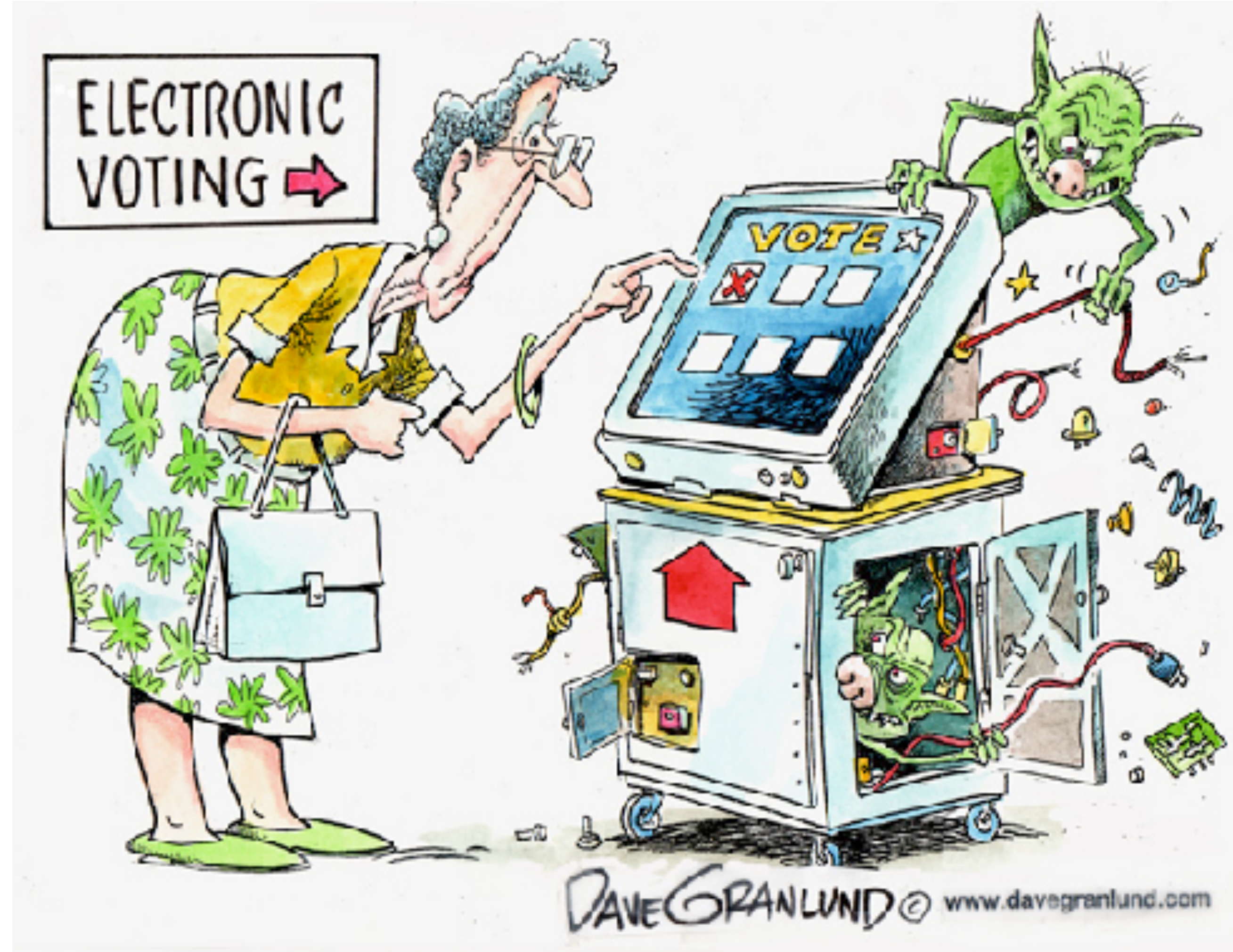
Semantics


IF I HAVE 12 TOMATOES AND TAKE AWAY TWO...
12
-2
3-28
©2005 Jef Mallett/Distributed by United Feature Syndicate, Inc.
WHAT IS THE DIFFERENCE?
EXACTLY. I DON'T LIKE TOMATOES, EITHER.
www.comics.com jefmallett@yahoo.com
WHAT ARE YOU IN FOR THIS TIME?
SEMANTICS.
DETENTION
MALLETT

Definition of Security Properties

Ana Matos
Miguel Correia Pedro Adão

Example: E-Voting



- Functionality
  - Cast vote, audit vote, determine winner, ...
- Assets
  - votes, voting service,
- Threat model
  - Goal: the adversary desires to influence the election.
  - Methods: Voting coercion, disabling or controlling voting booths, communications, servers...
- (...)

- (...)
- Security properties
  - Every citizen can cast a vote. (availability)
  - The identity of who placed each vote is anonymous, and the choice of each voter is known only to the voter. (confidentiality)
  - Only the voter can determine its vote. (integrity)
  - The voter can verify that its vote was casted correctly. (auditability)...
- Enforcement mechanism
  - Authentication, cryptographic protocols ensure (up to cryptographic strength) anonymity, integrity and auditability, ...

How can we get strong guarantees?

We need to be precise.

- "Is the program secure?"
- "Is the program secure against this threat?"
- "Is the program secure against attacker model A?"
- "Does the program meet security property P in the execution context C?"
- "Can we prove that the program meets security property P in the execution context C?"

“Secure” programs

- We want to ensure that programs are “secure”.
- First, we must be clear about what we are protecting, what is the power of the attacker, and what being secure actually means.

Class Outline

- Noninterference, intuitively
- Formal semantics
  - Big-step operational semantics
- Formalization of Noninterference

Class Outline

- Noninterference, intuitively
- Formal semantics
  - Big-step operational semantics
- Formalization of Noninterference

“no illegal flows”

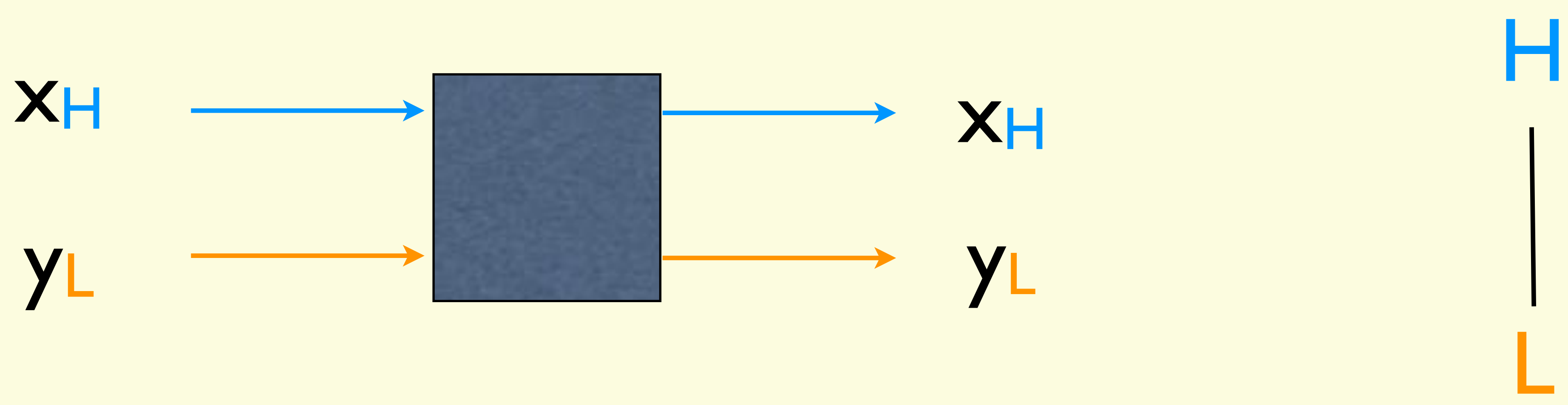
- We want to ensure that propagation of information by programs respects information flow policies, i.e. there are no illegal flows
- “no illegal flows” = an attacker cannot infer secret input or affect critical output by inserting inputs into the system and observing its outputs.
- How can we express the property of whether a program respects information flow policies?

Challenge for next class

- Secure program = the program does not encode illegal information flows
- Can you formulate a security property that defines when a program is secure?

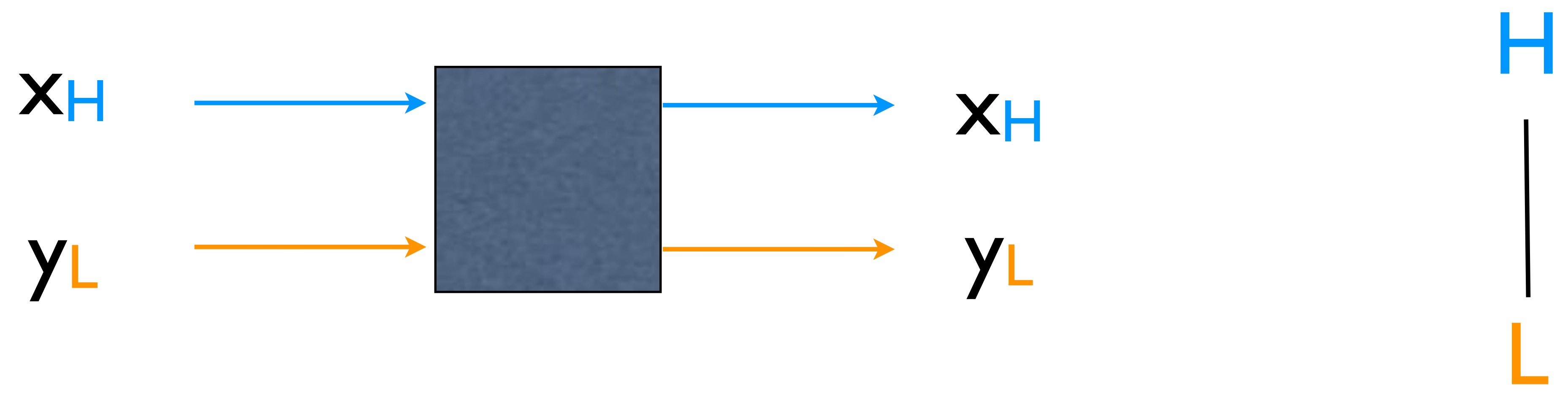
Tips

- Suppose you are the attacker. You are given a black box program, and you can control the "low" inputs of the program, and read its "low" outputs.
  - How many executions would you need to find out if the program leaks information?
  - What inputs would you give it in order to find out whether the program leaks information?



- Which of the following experiments allows you to conclude whether the program encodes leaks?
  - Input: y_L=0, Output: y_L=0
  - Input: y_L=0, Output: y_L=1
  - Input: y_L=0, Output: y_L=0 and Input: y_L=0, Output: y_L=0
  - Input: y_L=0, Output: y_L=0 and Input: y_L=1, Output: y_L=1
  - Input: y_L=0, Output: y_L=0 and Input: y_L=0, Output: y_L=1

Noninterference (for H-L)



- The program is secure if, for any two runs of the program that are given the same *low* (L) inputs, if the program terminates, it produces the same low (L) outputs.

Preserves confidentiality?

- $y_{secret} := x_{you}$ ✓
- $x_{you} := y_{secret}$ ✗

Explicit leak

- if y_{secret} then $x_{you} := 0$ else $x_{you} := 0$ ✓
- if y_{secret} then $x_{you} := 0$ else $x_{you} := 1$ ✗
- if x_{you} then $y_{secret} := 0$ else $y_{secret} := 1$ ✓

Implicit leak

- $x_{you} := 0$; while $x_{you} < y_{secret}$ do $x_{you} := x_{you} + 1$ ✗
- $x_{you} := 1$; while y_{secret} do $x_{you} := 0$ ✓
- while x_{you} do skip ; $y_{secret} := 0$ ✓

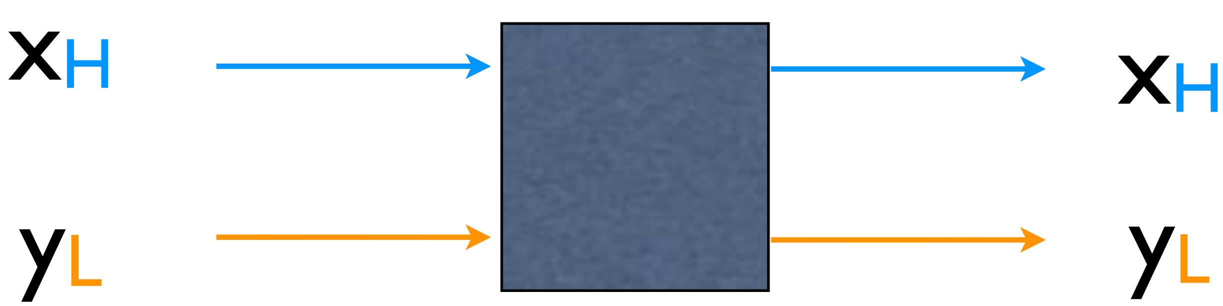
Preserves integrity?

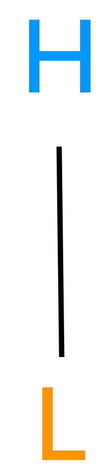
- $y_{untainted} := x_{you}$
- $x_{you} := y_{untainted}$
- if $y_{untainted}$ then $x_{you} := 0$ else $x_{you} := 0$
- if $y_{untainted}$ then $x_{you} := 0$ else $x_{you} := 1$
- if x_{you} then $y_{untainted} := 0$ else $y_{untainted} := 1$
- $y_{untainted} := 0$; while $x_{you} < y_{untainted}$ do $y_{untainted} := y_{untainted}+1$
- $x_{you} := 1$; while y_{secret} do $x_{you} := 0$
- while x_{you} do skip ; $y_{untainted} := 0$

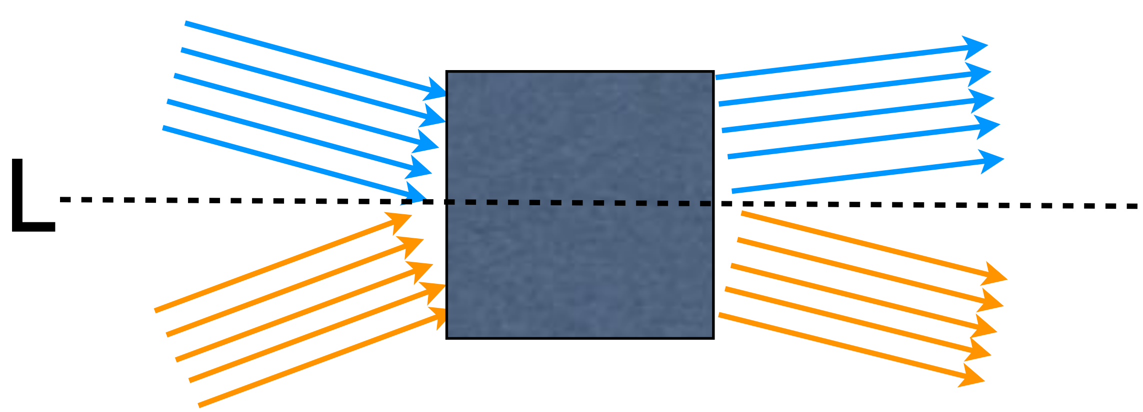
Explicit leak

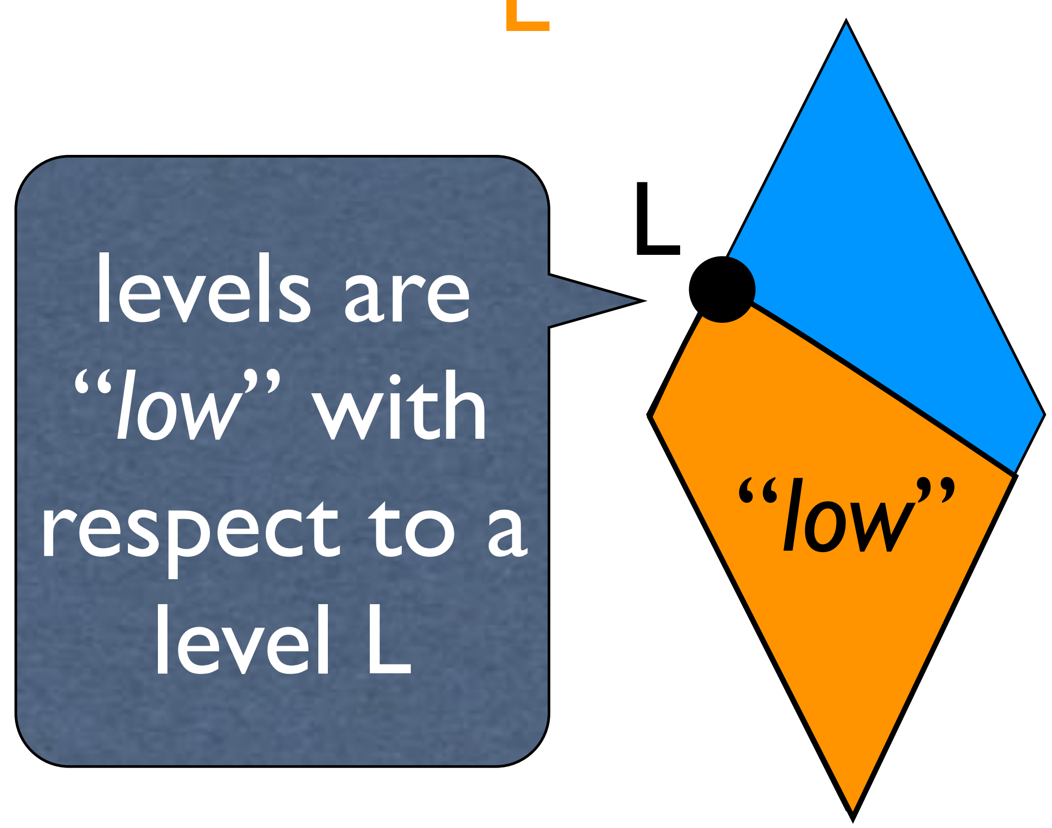
Implicit leak

Noninterference, for general policies

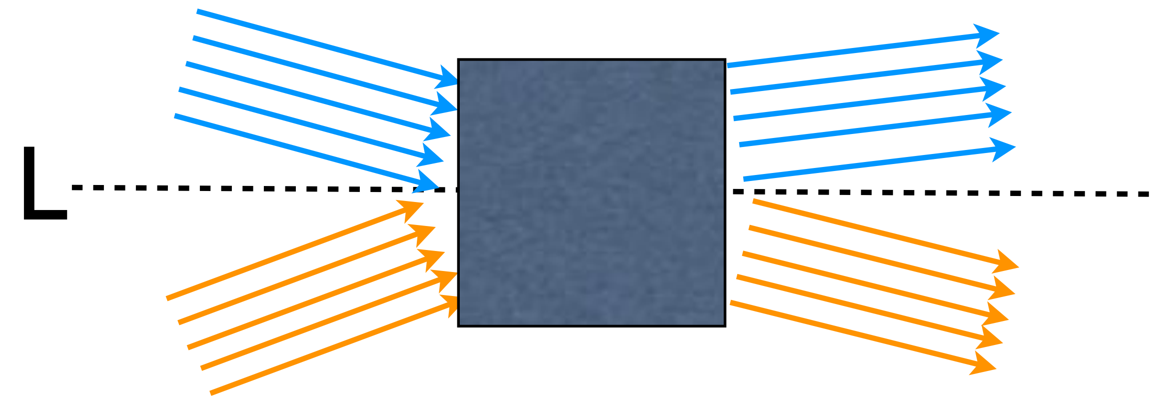


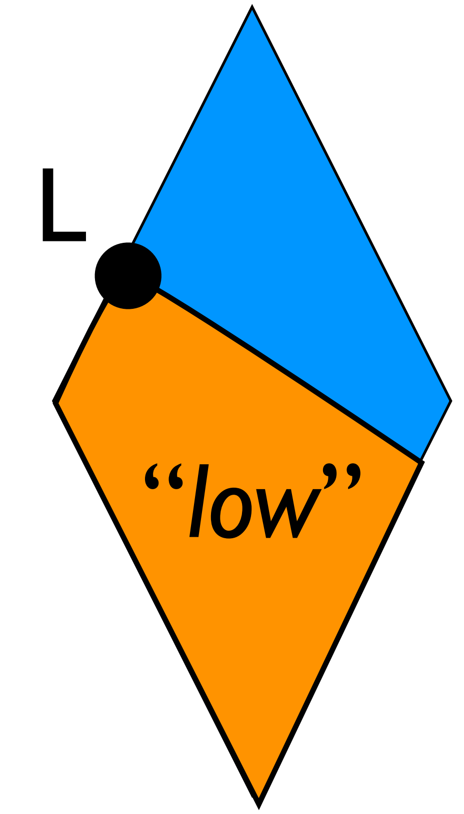






Noninterference (general policies)





- A program is secure if, <u>for every observational level L</u>, for any two runs of the program that are given the same *low* inputs, if the program terminates, it produces the same *low* outputs.

What is the power of the attacker?

- The attacker we are assuming cannot observe
  - non-terminating computations
  - intermediate steps of computations
  - time that it takes to produce the output
  - likelihood of each possible output
- This can be inadequate for different
  - execution contexts
  - language expressivity

Deterministic Input-Output attacker

- Execution context: sequential (deterministic)
- Intuition: an attacker is a program that is executed sequentially after the observed program, and has access to "low" outputs.
- Deterministic Input-Output Noninterference: Is only sensitive to outputs of <u>terminating</u> computations.

Concurrent execution context

- Is our notion of Noninterference adequate in a concurrent context?
  - $x_L{:=}1 \mid\mid (x_L{:=}2; x_L{:=}x_L{+}2)$
- The above program is considered insecure because of its non-deterministic behavior
  - $< x_L{:=}1 \mid\mid (x_L{:=}2; x_L{:=}x_L{+}2), \rho> \rightarrow \rho[x_L{:=}4]$
  - $< x_L{:=}1 \mid\mid (x_L{:=}2; x_L{:=}x_L{+}2), \rho> \rightarrow \rho[x_L{:=}1]$
  - $< x_L{:=}1 \mid\mid (x_L{:=}2; x_L{:=}x_L{+}2), \rho> \rightarrow \rho[x_L{:=}3]$

Concurrent execution context

- The following programs are considered secure with respect to our notion of Noninterference
  - if PIN_H then y_H:=false else z_H:=false
  - while y_H do skip ; x_L:=1 ; z_H:=false
  - while z_H do skip ; x_L:=0 ; y_H:=false
- But when composed concurrently, the program is insecure!

Concurrent attacker

- Execution context: concurrent
- Intuition: an attacker program that is concurrently composed with the observed program does not depend on its termination. It has access to "low" outputs, and possibly non-termination (or even intermediate steps).
- Possibilistic Input-Output Noninterference:
  Is sensitive to whether the program is <u>capable</u> of terminating and producing certain final outputs.

Secure? (w.r.t. ...)

- $y_H := x_L$ ✓
- $x_L := y_H$ ✗
- if y_H then $x_L := 0$ else $x_L := 1$ ✗
- while y_H do skip ; $x_L := 0$ ✗

Termination leak

Counter-example:
$\rho_1(x_L) = \rho_2(x_L) = 1$ and $\rho_1(y_H)$=false and $\rho_2(y_H)$=true
(the program cannot terminate on ρ_2)

Intermediate-step attacker

$x_L := y_H ; x_L := 1$

Possible low outcomes do not depend on y_H. However, the intermediate steps differ.

- Intermediate-step-sensitive Noninterference: Is sensitive to intermediate steps of computations.

  (To define an intermediate-step-sensitive property we would use a small-step semantics.)

Time-sensitive attacker

x_L:=0 ; if y_H then skip else skip;skip;skip;skip ; x_L:=1

Possible outcomes and intermediate steps do not depend on y_H. However, the time it takes to change the value of x_L is different.

- Temporal Noninterference:
  Is sensitive to the time it takes to produce outputs.

Probabilistic attacker

$x_L := y_H \;\;||\;\; x_L := \text{random}(100)$

Possible outcomes do not depend on y_H. However, the probability of the value of x_L revealing that of y_H is higher.

- Probabilistic Noninterference:
  Is sensitive to the likelihood of outputs.

Limits of Noninterference

- As sensitive as the attacker model.
    - Covert channels?
- As flexible as the security property permits.
    - Too restrictive?

What is the power of the attacker?

- Can the attacker observe:
  - That a program does not terminate?
  - Intermediate steps of the computation?
  - The possibility of producing certain states?
  - The likelihood of producing certain states?

What is the power of the attacker?

- Execution context: sequential (deterministic)
- Intuition: an attacker is a program that is sequentially composed with S, and has access to "low" outputs.
- Deterministic Input-Output attacker:
  Can only observe final outputs of <u>terminating</u> computations.

Covert channels

- Information can be transferred via other side-channels that are not intended for that purpose:
  - computation time
  - memory allocation
  - power consumption
  - ...

More flexibility

- Noninterference is simple and provides strong security guarantees. But sometimes we need to **leak information in a controlled way**.

- Need to reveal a bit of a secret (for confidentiality)
  if (password$_H$ == attempt$_L$)
     then print$_L$ "Right!"
     else print$_L$ "Wrong!"

- Need to trust certain user input (for integrity)
  \$filename$_L$=<STDIN>;
  open$_H$(FOO,"> \$filename$_L$");

Downgrading

- **Declassification** (for confidentiality)
  Example: flow declarations locally enable more flows

```
declassify password:L in
   if (password_H == attempt_L) {
     then print_L "Right!";
     else print_L "Wrong!";
   }
```

- **Endorsement** (for integrity)
  Example: pattern matching in Perl's taint mode

```
if ($filename_L =~ /^([-\@\w.]+)$/) {
  $filename_H = $1_H;
  open_H(FOO,"> $filename_H");
}
```

Conclusion

- The framework for the analysis should reflect the assumptions we want to study:
  - Language expressivity?
  - Execution context?
  - Attacker power?
  - Strength/flexibility of the security requirements?

In this course

- We will focus primarily on Input-Output Deterministic Noninterference.
- We will study different enforcement mechanisms for this security property.
- We want to have strong guarantees of that our mechanisms work.
- We will therefore use formal methods.

To start with, we need

- A precise way to talk about:
  - what are the possible programs and inputs
  - what is the (final) result (output) of executing a (deterministic) program with a certain input.
- In other words, we need:
  - a formal language (syntax and semantics)
  - notation to reason about terminating computations.

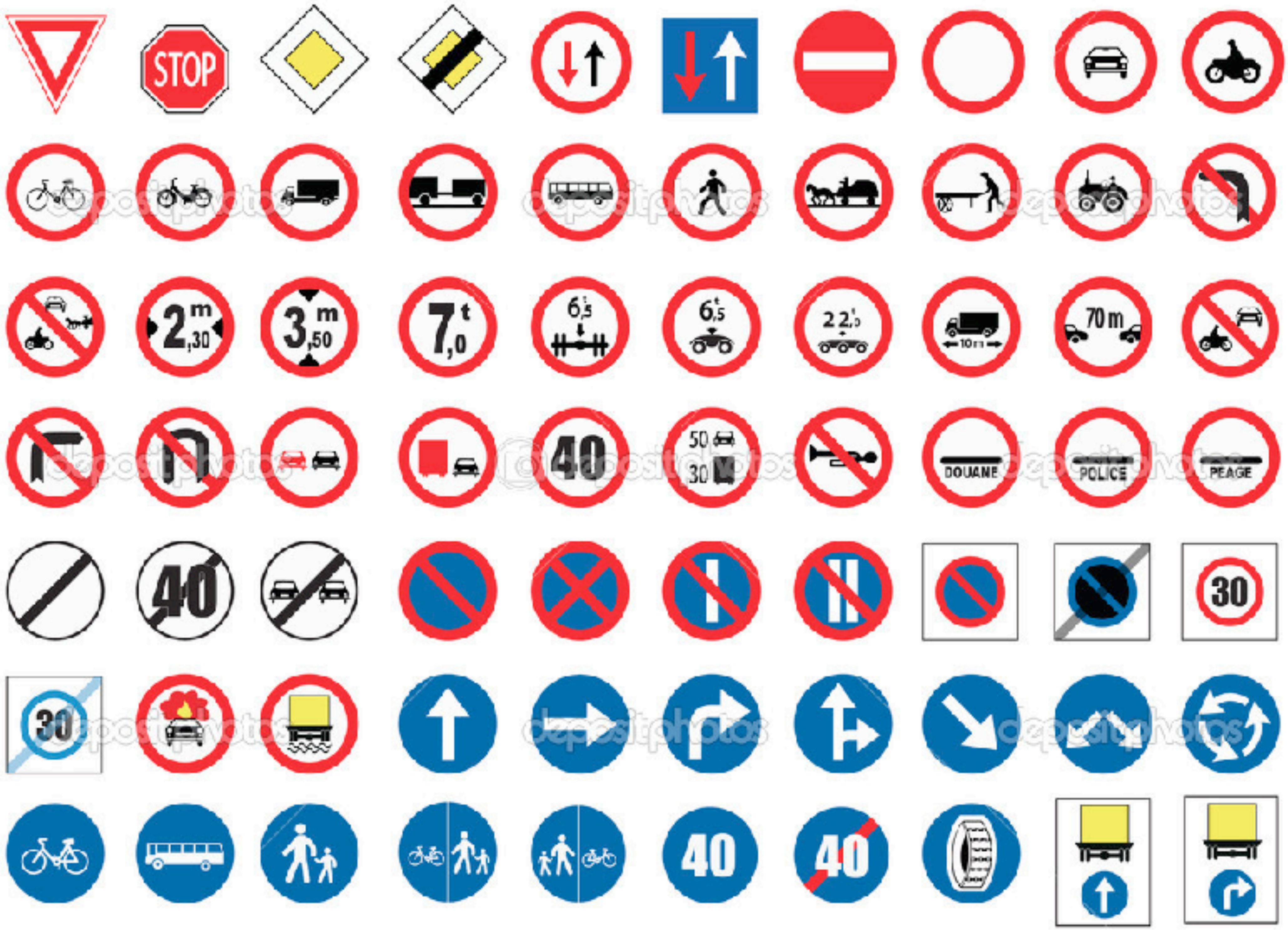
Formal security property

- Why use a **formal** security property?
  - For clarity (preventing ambiguities in the specification)
  - For conciseness
  - For correctness (the basis for implementation, analysis and verification)
- What do we need?
  - Techniques for reasoning about the syntax and the semantics of programs.

Class Outline

- Noninterference, intuitively
- Formal semantics
  - Big-step operational semantics
- Formalization of Noninterference

Semantics?



Programming language semantics

specification of the meaning of programs in that programming language

☞ "Semantics with Applications: A Formal Introduction", H. Nielson, F. Nielson, 1992 (Chapter 1 -- Section 2.1.).

Defining the semantics of a programming language

We will use two techniques:

- **Denotational** semantics for **expressions**: defines mathematically **what** is the result of a computation.

- **Operational** semantics for **instructions**: describes **how** the effect of a computation is produced when executed on a machine.

A WHILE language

- Note:
  - The next slides present a WHILE language as defined in the tutorial G. Barthe et al. (2011).
  - The language is a (simpler) variation of the one in Nielson & Nielson (1999).(*)

(*) Differences are:

- We refer to the natural operational semantics as "big-step semantics", and to the structural operational semantics as "small-step semantics".
- Constants are simply integers (and not numerals).
- Arithmetic expressions use operations that are used directly in the semantics.
- Boolean expressions are reduced to comparisons between arithmetic expressions (tests).

Syntax of WHILE

- Syntactic categories:
  c - constants (integers)
  x - variables
  a - arithmetic expressions
  t - tests
  S - statements

- Grammar (BNF notation):

operations

comparisons

expressions

tests

statements

op ::= + | - | × | / cmp ::= < | ≤ | = | ≠ | ≥ | >

a ::= c | x | a_1 op a_2 t ::= a_1 cmp a_2

S ::= x:=a | skip | S_1 ; S_2 | if t then S else S | while t do S

State (a.k.a. memory): ρ

What is a state/memory?

- ρ - a function that maps variables to integers

  Example: $\rho(x) = 1, \quad \rho(y) = 42 \ldots$

Functions that give meaning

Let's make use of the following semantic functions:

- $\mathcal{A}$ - function that maps pairs of arithmetic expression and state to integers (assume defined) ✓
- $\mathcal{B}$ - function that maps pairs of test and state, to booleans (assume defined) ✓
- $\mathcal{S}$ - partial function that maps pairs of statement and state to state (to define next).

Meaning of statements: $\mathcal{S}$

- $\mathcal{S}$ - partial function that maps pairs (statement,state) to state.
- We will define it using big-step transitions that speak about how the overall result of executions is obtained.

$$\langle S, \rho \rangle \rightarrow \rho'$$

When executing program S on memory ρ we obtain the new memory ρ'

- We could also use small-step transitions that speak about how the individual steps of the computations take place.

$$\langle S, \rho \rangle \Rightarrow \langle S', \rho' \rangle$$

Performing one step of program S on memory ρ leaves the continuation S' and produces new memory ρ'

- (We will use small step semantics later in the course.)

Big-step Operational Semantics

- Skip: $< \text{skip}, \rho > \rightarrow \rho$

Does nothing!

- Assignment: $< x{:=}a, \rho > \rightarrow \rho[x \mapsto \mathcal{A}[\![a]\!]_\rho]$

Updates the value of x in ρ with the result of evaluating a

the update of state ρ is defined as:

$$\begin{cases} (\rho[y \mapsto c])(x) = c, & \text{if } x=y \\ \rho(x), & \text{otherwise} \end{cases}$$

Axioms - do not depend on any hypothesis in order to give the final result of the entire computation

When the first program starting on ρ produces ρ'...

... and the second program starting on ρ' produces ρ''...

- Sequential composition:

$$\frac{\langle S_1, \rho\rangle \rightarrow \rho' \qquad \langle S_2, \rho'\rangle \rightarrow \rho''}{\langle S_1;S_2, \rho\rangle \rightarrow \rho''}$$

... then the entire sequential composition starting on ρ produces ρ''.

Rules - the final result of the entire computation below the line, depends on the hypothesis above the line

When t evaluates to true…

… and the first branch starting on ρ produces ρ'…

- Conditional test:

$$\frac{\langle S_1, \rho \rangle \to \rho'}{\langle \text{if } t \text{ then } S_1 \text{ else } S_2, \rho \rangle \to \rho'} \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{true}$$

then the conditional starting on ρ produces ρ'.

$$\frac{\langle S_2, \rho \rangle \to \rho'}{\langle \text{if } t \text{ then } S_1 \text{ else } S_2, \rho \rangle \to \rho'} \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{false}$$

When t evaluates to true...

- While loop:

$$\frac{\langle S,\rho\rangle \to \rho' \quad \langle \text{while } t \text{ do } S, \rho'\rangle \to \rho''}{\langle \text{while } t \text{ do } S, \rho\rangle \to \rho''} \qquad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{true}$$

... the body starting on ρ produces ρ'...

... and the continuation of the cycle on ρ' produces ρ''...

... then the cycle on ρ produces ρ''.

$$\langle \text{while } t \text{ do } S, \rho\rangle \to \rho \qquad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{false}$$

When t evaluates to false the cycle does nothing.

(All) Big-step axioms & rules

Assignment: $<x{:=}a,\ \rho> \rightarrow \rho[x \mapsto \mathcal{A}[\![a]\!]_\rho]$

Skip: $<\text{skip},\ \rho> \rightarrow \rho$

Sequential composition:
$$\frac{<S_1,\ \rho> \rightarrow \rho' \quad <S_2,\ \rho'> \rightarrow \rho''}{<S_1;S_2,\ \rho> \rightarrow \rho''}$$

Conditional test:
$$\frac{<S_1,\ \rho> \rightarrow \rho'}{<\text{if } t \text{ then } S_1 \text{ else } S_2,\ \rho> \rightarrow \rho'} \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{true}$$

$$\frac{<S_2,\ \rho> \rightarrow \rho'}{<\text{if } t \text{ then } S_1 \text{ else } S_2,\ \rho> \rightarrow \rho'} \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{false}$$

While loop:
$$\frac{<S,\rho> \rightarrow \rho' \quad <\text{while } t \text{ do } S,\ \rho'> \rightarrow \rho''}{<\text{while } t \text{ do } S,\ \rho> \rightarrow \rho''} \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{true}$$

$$<\text{while } t \text{ do } S,\ \rho> \rightarrow \rho \quad \text{if } \mathcal{B}[\![t]\!]_\rho = \text{false}$$

Example - Evaluation

- Evaluate $(z:=x ; x:=y) ; y:=z$, starting from a state ρ_0 that maps all variables except x and y to 0, and has $\rho_0(x) = 5$ and $\rho_0(y) = 7$.

derivation tree

$$\dfrac{\dfrac{\langle z:=x, \rho_0\rangle \to \rho_1 \qquad \langle x:=y, \rho_1\rangle \to \rho_2}{\langle (z:=x ; x:=y), \rho_0\rangle \to \rho_2} \qquad \langle y:=z, \rho_2\rangle \to \rho_3}{\langle (z:=x ; x:=y) ; y:=z, \rho_0\rangle \to \rho_3}$$

$$\rho_1 = \rho_0[z \mapsto 5] \quad \rho_2 = \rho_1[x \mapsto 7] \quad \rho_3 = \rho_2[y \mapsto 5]$$

Evaluation - derivation tree

To evaluate a statement S, starting from a state ρ, a derivation tree must be constructed:

1. Construct the tree from root upwards.
2. Try to find an axiom or rule whose left side matches <S, ρ> and whose side conditions are satisfied.

- If it is an axiom - determine the final state and terminate.
- If it is a rule - try to construct the derivation tree for the premisses of the rule in order to determine the final state.

The semantic function $\mathcal{S}$

- The meaning of statements is given as a (partial) function from the set of states to the set of states.

$$\mathcal{S}[\![\mathrm{S}]\!]_{\rho} = \begin{cases} \rho' & \text{if } \langle \mathrm{S}, \rho\rangle \rightarrow \rho' \\ \text{undefined}, & \text{otherwise} \end{cases}$$

no meaning is given to non-terminating computations

Summarizing: Big-step transition system

- Configurations:
  - intermediate <Statement S, state ρ>
  - terminal ρ
- Transitions: $\langle S, \rho \rangle \rightarrow \rho'$
- Rules: $\dfrac{\langle S_1, \rho_1 \rangle \rightarrow \rho_1' \ \ldots \ \langle S_n, \rho_n \rangle \rightarrow \rho_n'}{\langle S, \rho \rangle \rightarrow \rho'}$ if ....

Conclusions

- We have defined formally the operational semantics for a simple language WHILE.
- The same can be done **for any language in a similar** manner.
- This allows us to **reason rigorously about the behavior** of programs in the language.
- We will now use it to **formalize and prove security properties**, which are often semantic.

Class Outline

- Noninterference, intuitively
- Formal semantics
  - Big-step operational semantics
- Formalization of Noninterference

Formalization of Noninterference

☞ "Language-Based Information-Flow Security", A. Sabelfeld and A. Myers , 2002.

Noninterference (informally)

Noninterference -
A program is secure if, for any observational level L, for any two runs of the program that are given the same *low* inputs, if the program terminates, it produces the same *low* outputs.

Noninterference (informally)

Program from a given programming language.

Given an information flow policy, all security levels that are lower or equal to L.

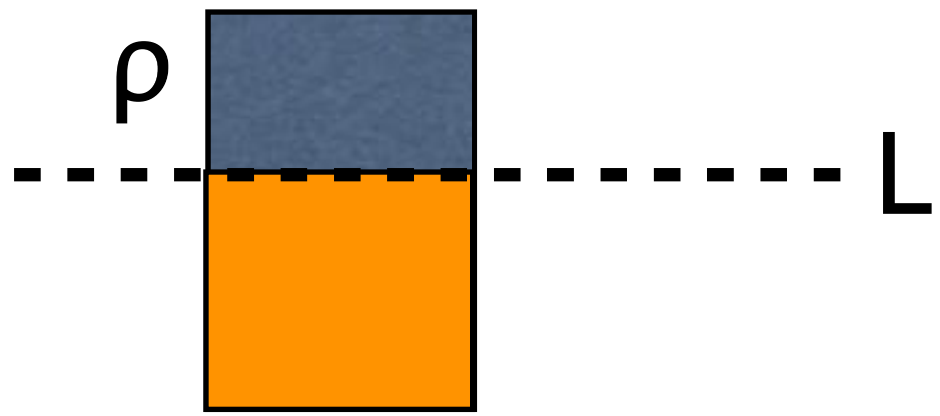
Noninterference -
A program is secure if, for any observational level L, for any two runs of the program that are given the same *low* inputs, if the program terminates, it produces the same *low* outputs.

Execution of the program according to specified semantics.

Security labeling and indistinguishability relation between states.

Observable at level L

The part of a memory ρ that is observable at level L corresponds to the set of variables that are lower or equal to L.

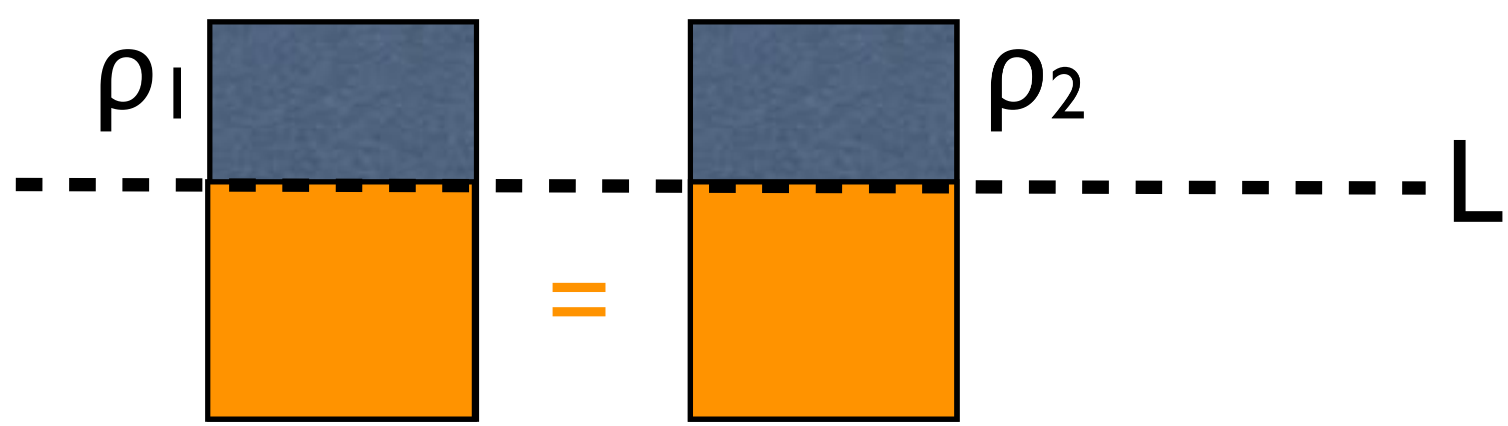


(Omitting the parameter Γ for simplicity.)

Indistinguishable at level L

Two memories ρ_1 and ρ_2 are indistinguishable with respect to a security level L, if ρ_1 and ρ_2 agree on the values of variables that are observable at level L.

I.e.: For all x such that $\Gamma(x) \leq L$, then $\rho_1(x) = \rho_2(x)$.



We then write $\rho_1 \sim_L \rho_2$.

(Omitting the parameter Γ for simplicity.)

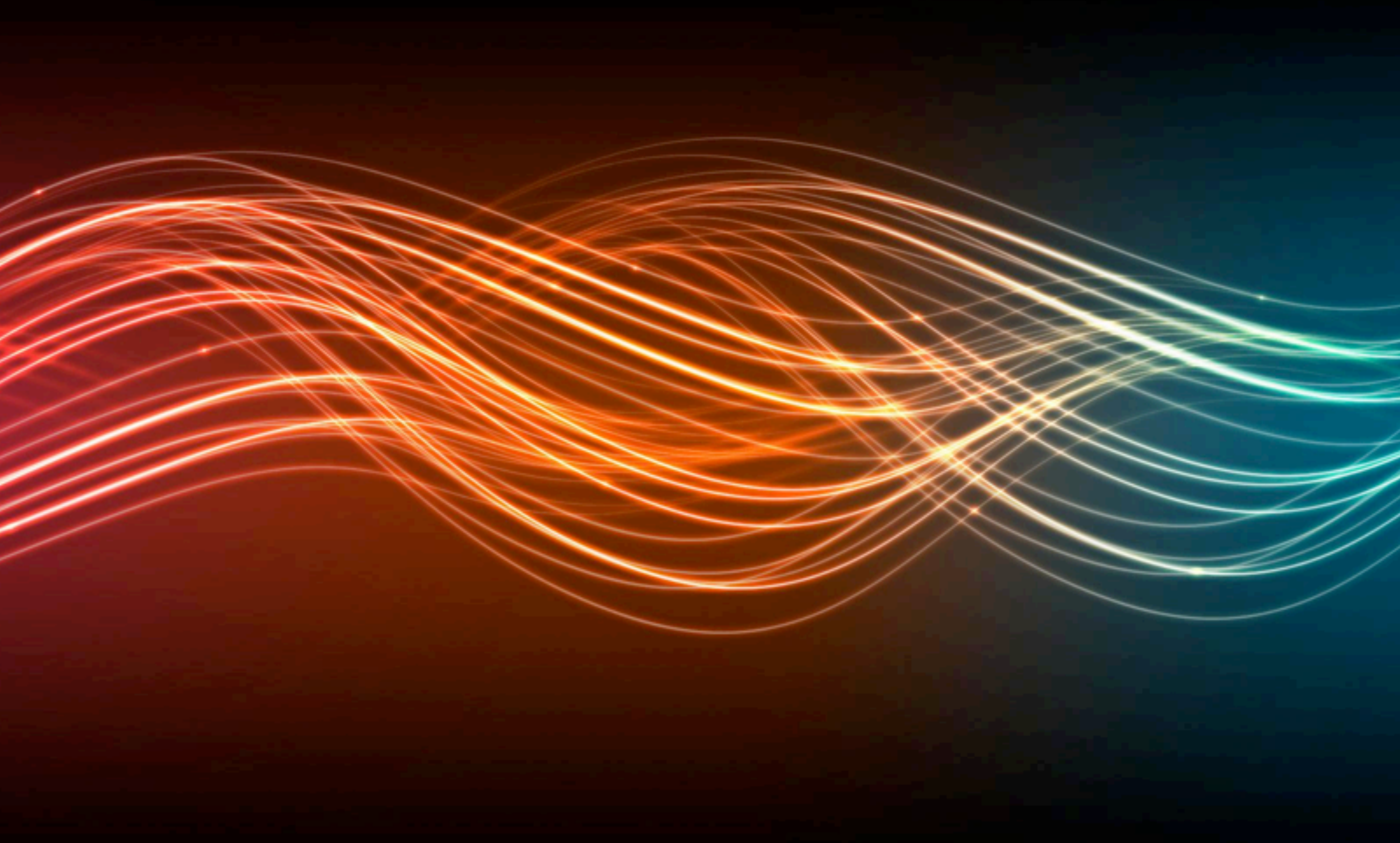
Noninterference (formally)

Deterministic Input-Output Noninterference -
A program S is secure if for every security level L and for all pairs of memories ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$, we have that

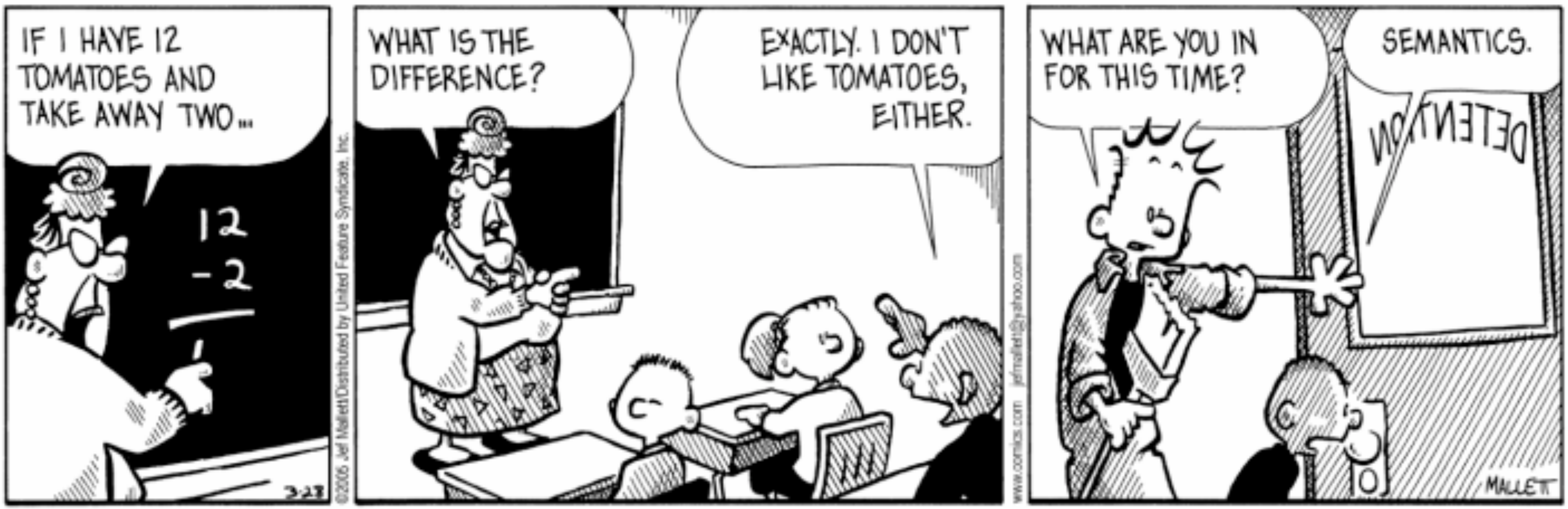
$\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ implies $\rho_1' \sim_L \rho_2'$.

Conclusions

- In order to have strong guarantees about security, we need to **be precise** about the security property we want to enforce.
- Defining a security property requires defining what is the **threat model**, including the execution context and the power of the attacker.
- To be precise about a (semantic) security property requires a **formal semantics** for expressing it.



Semantics



IF I HAVE 12 TOMATOES AND TAKE AWAY TWO...
12
-2
WHAT IS THE DIFFERENCE?
EXACTLY. I DON'T LIKE TOMATOES, EITHER.
WHAT ARE YOU IN FOR THIS TIME?
SEMANTICS.
DETENTION
©2005 Jef Mallett/Distributed by United Feature Syndicate, Inc.
www.comics.com jefmallett@yahoo.com
3-28
MALLETT