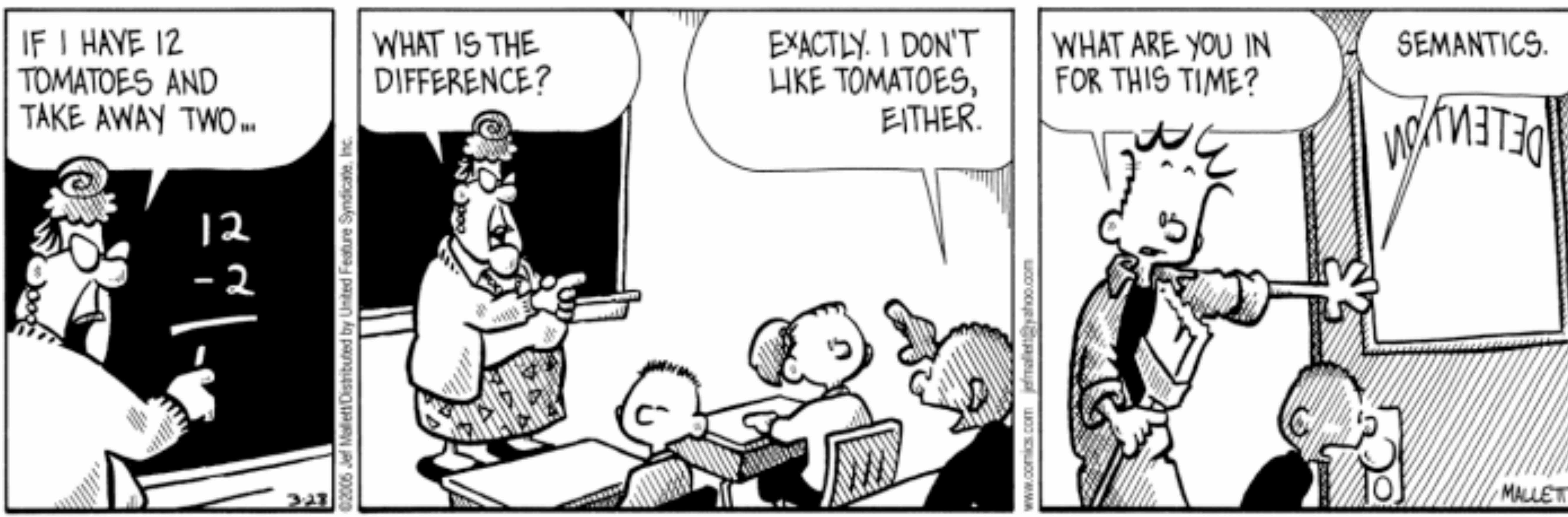
Semantics



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.
MALLETT

Definition of Security Properties

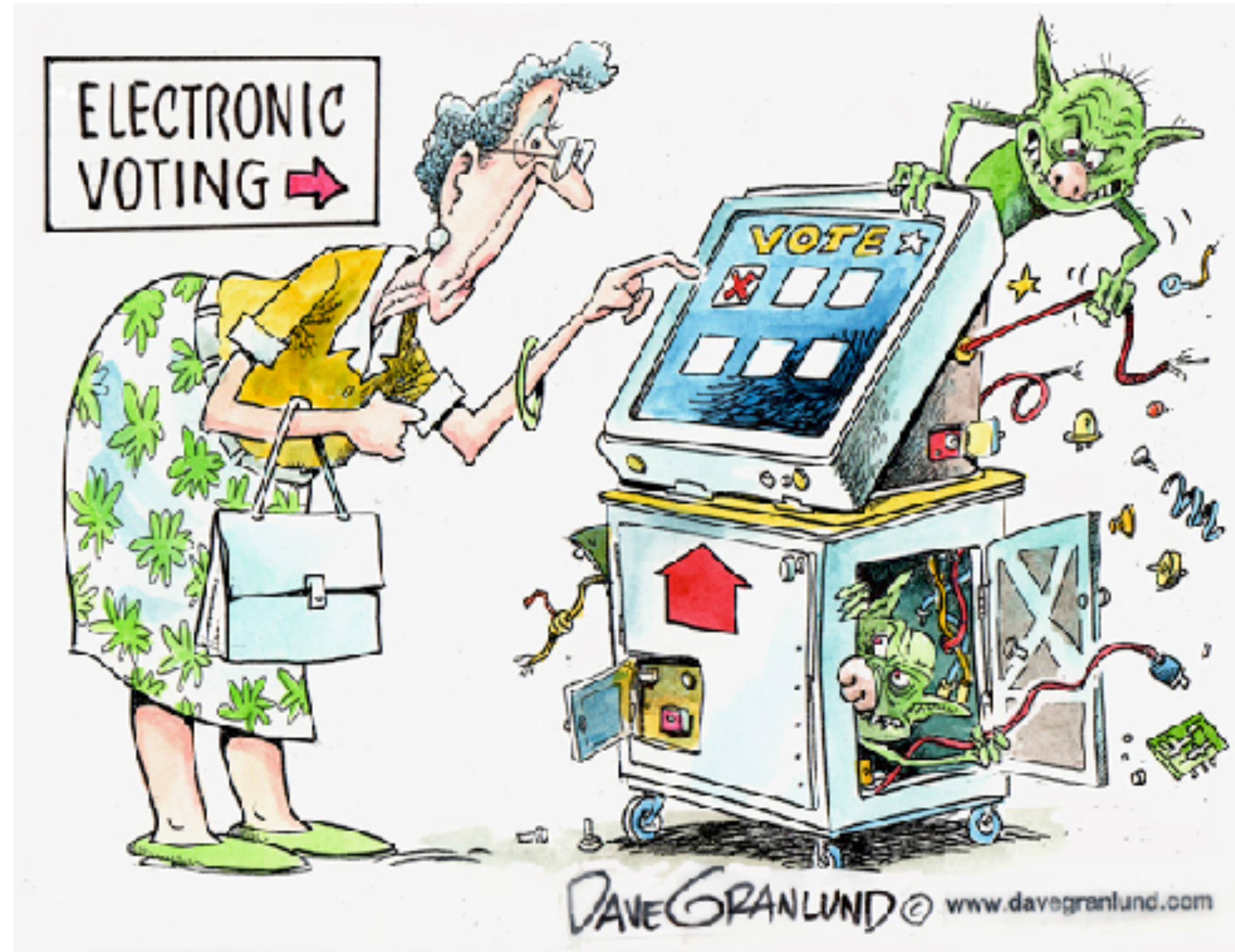
Ana Matos
Miguel Correia Pedro Adão

“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.

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?”

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

Class Outline

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

Noninteference, intuitively

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

The property of being "secure"

- We want to ensure that propagation of information by programs respects information flow policies: 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?

Suppose YOU are the attacker

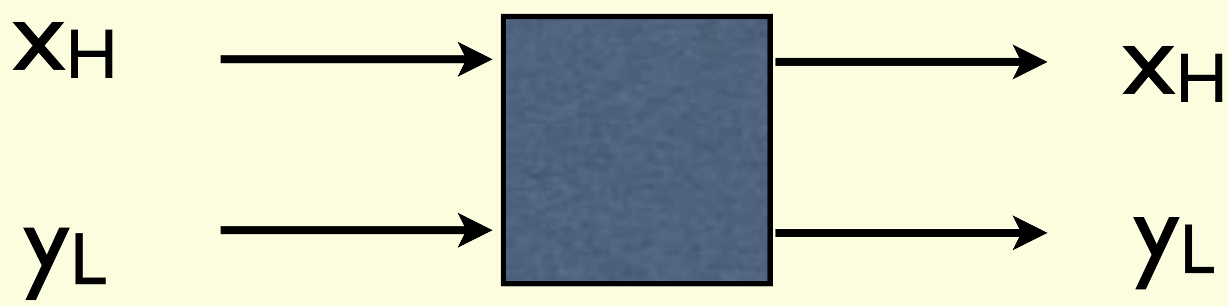
- You can only read and write to variables of "low level" -- i.e., your level or lower
- Can you use the program to uncover "high level" information?
- Can you use the program to affect "high level" information?

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?

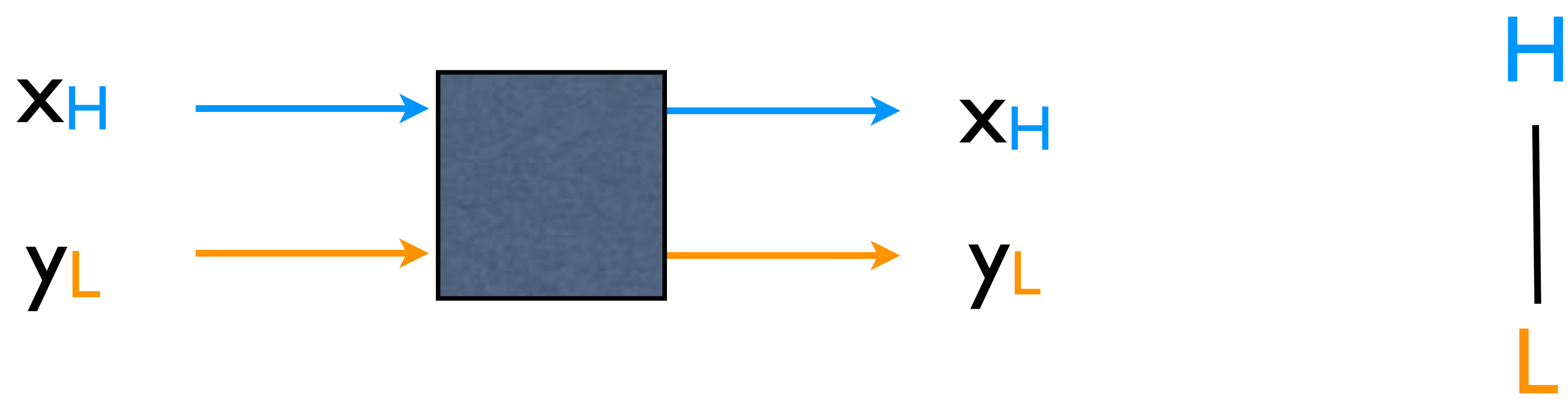


- Which of the following experiments allows you to conclude whether the program encodes leaks?
  - Input: y_L=0, Output: y_L=0
  - Input: y_L=0, Output: y_L=1
  - Input: y_L=0, Output: y_L=0 and Input: y_L=0, Output: y_L=0
  - Input: y_L=0, Output: y_L=0 and Input: y_L=0, Output: y_L=1

What is the power of the attacker?

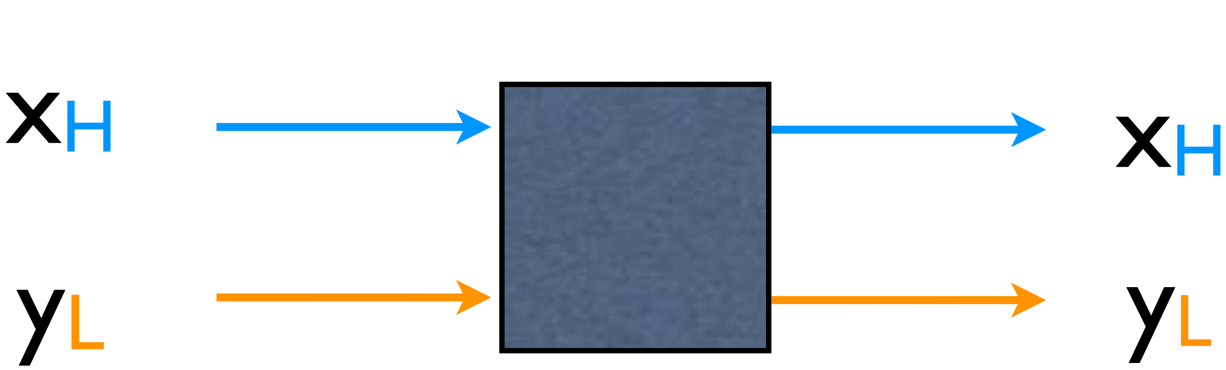
- Execution context: sequential (deterministic)
- Intuition: an attacker is a program that is sequentially composed with S, and has access to "low" outputs.
- Deterministic Input-Output attacker:
  Can only observe final outputs of <u>terminating</u> computations.

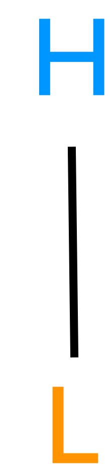
Noninterference (almost)

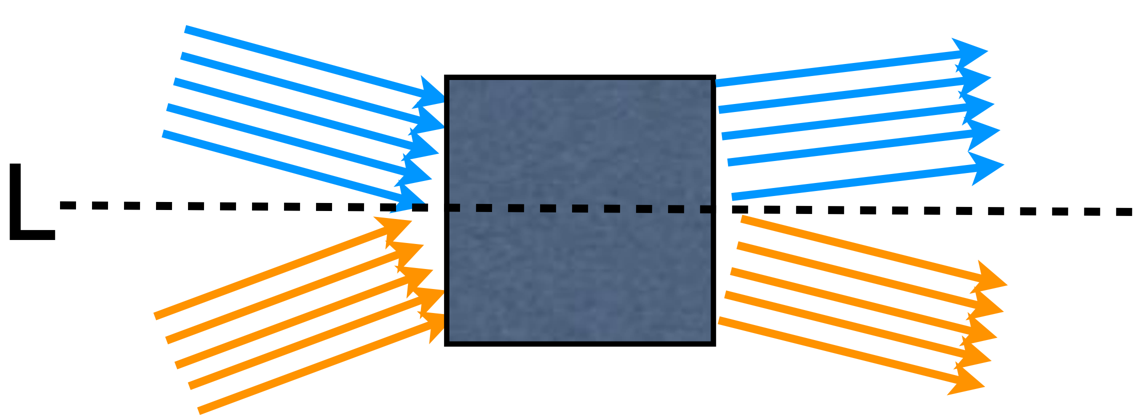


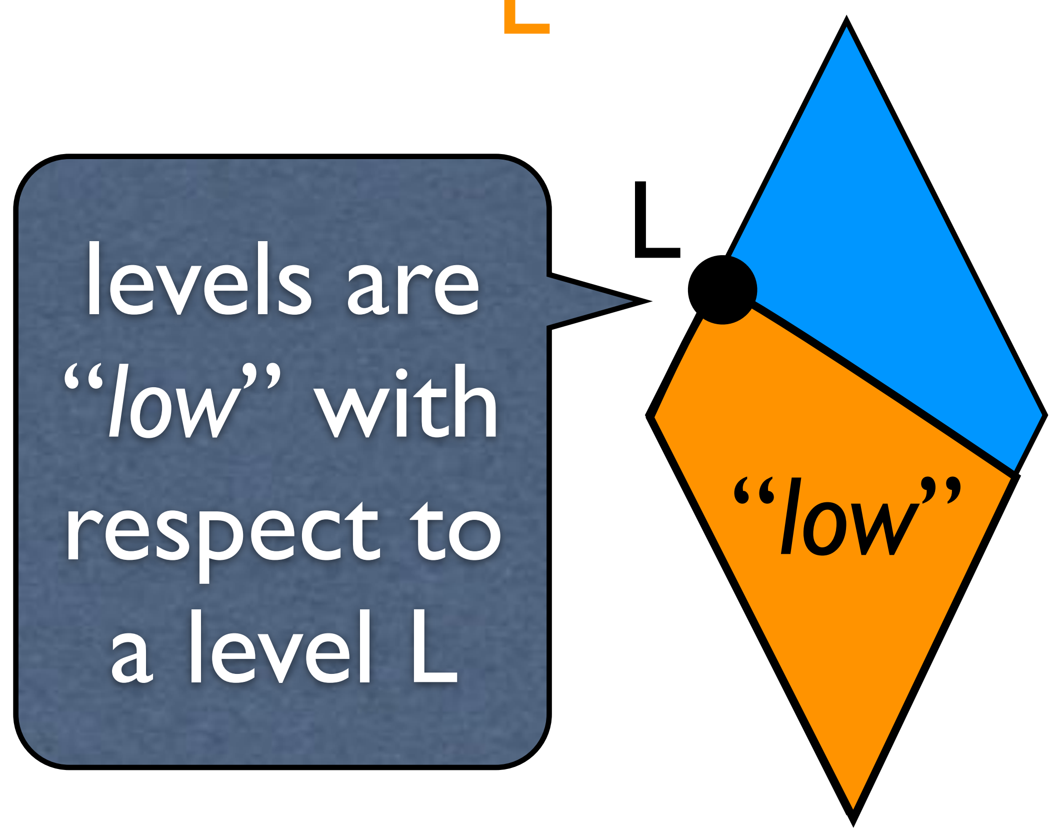
- The program is secure if, for any two runs of the program that are given the same *low* (L) inputs, if the program terminates, it produces the same low (L) outputs.

Noninterference, for general policies

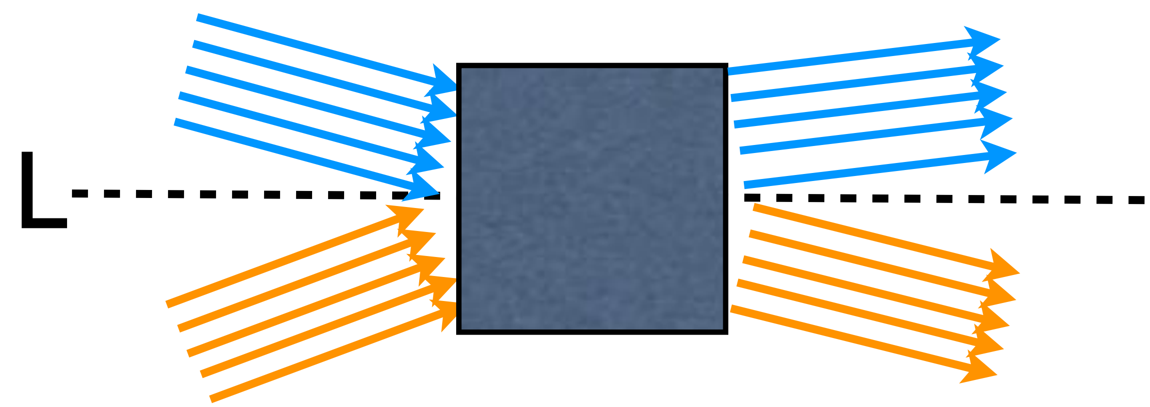


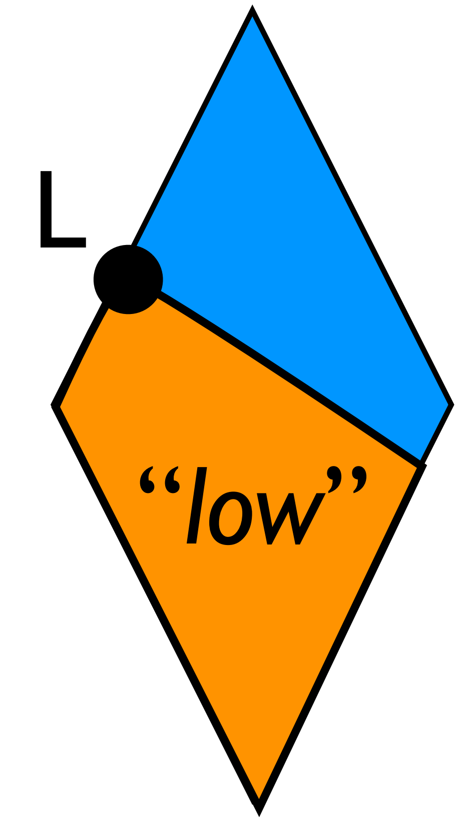






Noninterference (informally)





- A program is secure if, <u>for any 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.

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

- $y_H := x_L$
- $x_L := y_H$

Explicit leak

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

Implicit leak

Covert information flow

- Consider two files:
  - “secret.txt” - owned by Alice, can only be read and written by Alice
  - “public.txt” - can be read and written by everyone
- Can you conceive a program that discloses the secret information but respects Deterministic Input-Output Noninterference?
  - Can Discretionary Access Control prevent it?
  - Can Mandatory Access Control prevent it?

Limitations of the attacker

- A deterministic input-output attacker model assumes that the attacker 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

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!

What is the power of the attacker?

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

Possibilistic Input-Output Noninterference

- Program S is secure if, for any two memories that agree on “low” variables, if running S on one of them terminates and produces a certain final memory, then running S on the other can also terminate and produce a final memory that agrees on the “low” variables.

Termination-sensitive!

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:
  Can observe the likelihood of outputs.

Limits of Noninterference

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

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 appealing because it provides strong security guarantees. However it is too restrictive!
- if ($password_H$ == $attempt_L$)
  then $print_L$ “Right!“
  else $print_L$ “Wrong!”
- $filename=<STDIN>;
  open(FOO,"> $filename");

Downgrading

- Sometimes we need to leak information in a controlled way.
- Example for confidentiality (called declassification):

  declassify password:L in
  if ($password_H$ == $attempt_L$)
  then $print_L$ “Right!“
  else $print_L$ “Wrong!”

- And for integrity (called endorsement)?

  Think of Perl’s taint mode.

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.

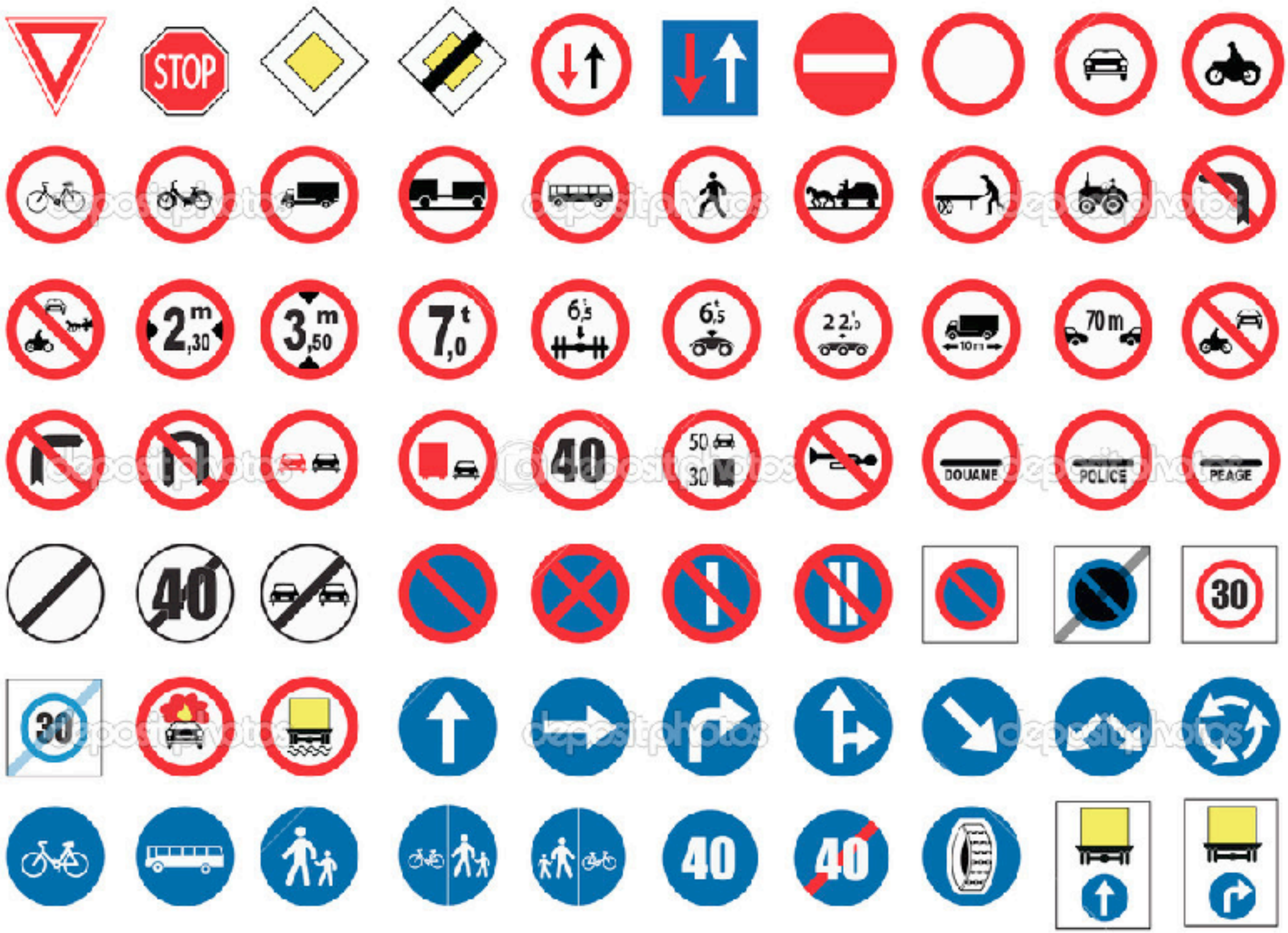
To start with, we need

- 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.

Class Outline

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

Semantics?


STOP
2 m ,30
3 m ,50
7 t ,0
70 m
40
50
30
DOUANE
POLICE
PEAGE
40
30
30
40
40

Programming language semantics

specification of the meaning of programs of 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 - defines what is the result of a computation, as a mathematical object.
- Operational semantics - describes how the effect of a computation is produced when executed on a machine.

A WHILE language

- 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:
  $c \in Z$ - constants (integers)
  $x \in Var$ - variables
  $a \in Aexp$ - arithmetic expressions
  $t \in Bexp$ - tests
  $S \in Stm$ - statements

- Grammar (BNF notation):

  $op ::= + \mid - \mid \times \mid /$ $cmp ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$

  $a ::= c \mid x \mid a_1 \text{ op } a_2$ $t ::= a_1 \text{ cmp } a_2$

  $S ::= x{:=}a \mid \text{skip} \mid S_1 \,;\, S_2 \mid \text{if } t \text{ then } S \text{ else } S \mid \text{while } t \text{ do } S$

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).

State function ρ

What is a state?

- ρ - memory or state function that maps variables to integers

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

Semantics of statements: S

- S - partial function that maps pairs (statement,state) to state.
- We will define it using a "big-step" operational semantics, by means of big-step transitions: speaks 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 have used a "small-step" operational semantics, by means of small-step transitions: speaks about how the <u>individual steps</u> of the computations take place.

$$<S, \rho> \Rightarrow <S', \rho'>$$

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.

Operational semantics (big-step transitions)

- 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:

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

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:

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

… 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'.

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

- While loop:

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

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

When t evaluates to 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''.

When t evaluates to false the cycle does nothing.

(All) Big-step axioms & rules

Assignment: $\langle x{:=}a, \rho\rangle \to \rho[x \mapsto \mathcal{A}[\![a]\!]_\rho]$

Skip: $\langle \text{skip}, \rho\rangle \to \rho$

Sequential composition: $$\frac{\langle S_1, \rho\rangle \to \rho' \quad \langle S_2, \rho'\rangle \to \rho''}{\langle S_1;S_2, \rho\rangle \to \rho''}$$

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

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

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

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

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

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

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

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.

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

no meaning is given to non-terminating computations

Summarizing: Big-step transition system

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

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 (in future class, for a byte-code language.)
- This allows us to **reason rigorously about the behavio**r of programs in the language.
- We will 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 -

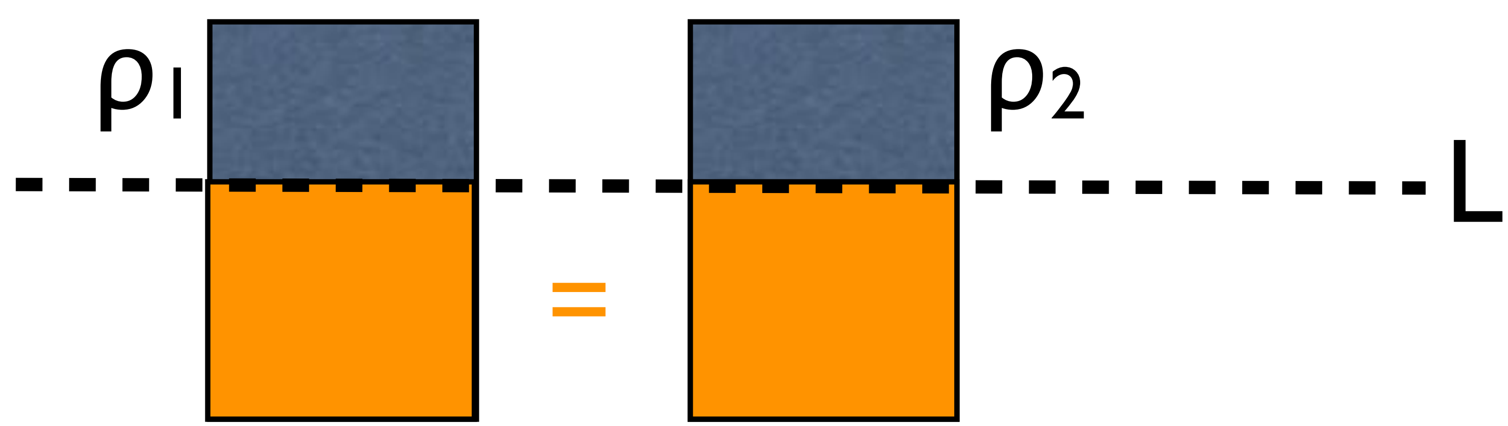
A program is secure if, for any observational level L, for any two runs of the program that are given the same *low* input, if the program terminates, it produces the same *low* outputs.

Execution of the program according to specified semantics.

Security labeling and indistinguishability relation between states.

Indistinguishability

Two memories ρ_1 and ρ_2 are indistinguishable with respect to a security labeling Γ and a level L, if ρ_1 and ρ_2 agree on the values of variables that are lower or equal to L. I.e.: For all variables x such that $\Gamma(x) \leq L$ we have that $\rho_1(x) = \rho_2(x)$.



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

(Omitting the parameter Γ for simplicity.)

Noninterference (formally)

Deterministic Input-Output Noninterference -
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.