

Definition of Security Properties – Exercises

Ana Matos
Software Security 2025/2026, IST

November 18, 2025

Goals

- To **recognize and to justify**, both informally and formally, why programs satisfy or not a given security property.
- To understand the specific security property **Deterministic Input-Output Non-interference**, to be used as a running example in future classes.
- To become familiar with basic notations and concepts for **describing, implementing, and reasoning about** the semantics of programs and their properties.
- To gain insight on the **semantic nature of security properties**, how they can be expressed formally, and the power offered by formal techniques.

1 Noninterference, informally

Consider the *High (H)* - *Low (L)* policies for confidentiality and integrity, and different labelings (mapping from variables to security levels), including where the variable x has level L and variable y has level H , and the other way around. For each of the programs below, state whether they satisfy Deterministic Input-Output Noninterference, and justify your answer carefully. You can assume that the property is suitable for the execution context and attacker model.

1. $y := x; x := y + 1$
2. $x := 42; x := y$
3. $x := y; x := 42$
4. if $y = 0$ then $x := 42$ else $x := 42$
5. if $x = 0$ then $x := 42$ else $x := 4242$
6. $x := y; \text{if } x = 0 \text{ then } x := 42 \text{ else } x := 4242$

7. $x := 0$; while $y = 0$ do $x := 1$
8. $x := y$; while $x \neq 0$ do $x := x - 1$
9. $x := 0$; while $x \neq y$ do $x := x + 1$

2 Semantics - Formal definition and Implementation

Consider the Semantics of Arithmetic and Boolean Expressions, and the Big-Step Semantics for the WHILE language, as was presented in class.

1. These exercises are closely based on examples and exercises that appear in Nielson & Nielson 1992 for the “Natural” Semantics. Construct a derivation tree that allows to determine the result of evaluating the following statements, or explain why it does not exist:
 - (a) $y := 1$; while $x \neq 1$ do $(y := y \times x; x := x - 1)$
on a memory such that x has the value 3.
 - (b) $z := 0$; while $y \leq x$ do $(z := z + 1; x := x - y)$
on a state where x has the value 17 and y has the value 5.

Can you determine a state ρ such that the derivation tree obtained for the above statements does exist/not exist?

2. Consider the following programs.
 - (a) while $x \neq 1$ do $(y := y * x; x := x - 1)$
 - (b) while $1 \leq x$ do $(y := y * x; x := x - 1)$
 - (c) while $1 = 1$ do skip

For each statement determine whether or not it terminates for all choices of initial memory and whether or not it always loops. Try to argue for your answer using the axioms and rules of the semantics.

3. Consider the following off-the-shelf interpreter of a simple imperative language, which implements a Big-step semantics:

<http://jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python-part-1>.

- (a) Download it, and understand the code that performs evaluation. (To run it in more recent Python versions you will need to add to the file `imp_parser.py` the line `from functools import reduce`.) Tips:
 - The `main` method in file `imp.py` uses a parser to build a representation of the syntax of the program (abstract syntax tree), and then invokes its `eval` method to evaluate it.
 - The methods for performing the evaluation, for each form of statement and expression, can be found in `imp_ast.py`.

- (b) Modify the interpreter so that it prints a representation of the derivation tree when executing programs. Tips:
- Add a parameter to the `eval` method of statements and expressions in order to achieve indentation according to depth of nesting.
 - Use the `repr` method to print the statement or expression.

3 Semantic Properties

1. Consider the *High (H) - Low (L)* security policy for confidentiality, where the variable x has level L and variable y has level H and the following three properties:

Property A for all pairs of memories ρ_1 and ρ_2 such that $\rho_1(x) = \rho_2(x)$, we have that $\langle S, \rho_1 \rangle \rightarrow \rho'_1$ implies $\langle S, \rho_2 \rangle \rightarrow \rho'_2$ and $\rho'_1(x) = \rho'_2(x)$.

Property B there exist two memories ρ_1 and ρ_2 such that $\rho_1(x) = \rho_2(x)$, and $\langle S, \rho_1 \rangle \rightarrow \rho'_1$ and $\langle S, \rho_2 \rangle \rightarrow \rho'_2$, but $\rho'_1(x) \neq \rho'_2(x)$.

Property C for all pairs of memories ρ_1 and ρ_2 such that $\rho_1(x) = \rho_2(x)$, we have that $\langle S, \rho_1 \rangle \rightarrow \rho'_1$ and $\langle S, \rho_2 \rangle \rightarrow \rho'_2$, implies $\rho'_1(x) = \rho'_2(x)$.

- (a) Which of the properties imply Deterministic Input-Output Noninterference and Termination Sensitive Input-Output Noninterference?
 - (b) For each of the programs S in Section 1 above, say whether properties **A**, **B**, and **C**, hold. Justify using the rules of the semantics.
2. Two statements S_1 and S_2 are said to be *semantically equivalent* when for all states ρ and ρ' we have that

$$\langle S_1, \rho \rangle \rightarrow \rho' \text{ if and only if } \langle S_2, \rho \rangle \rightarrow \rho'$$

Show whether the following pairs of statements are semantically equivalent.

- (a) $S_1; S_2$ and $S_2; S_1$
 - (b) $S_1; (S_2; S_3)$ and $(S_1; S_2); S_3$
 - (c) (advanced) $\text{while } t \text{ do } S$ and $\text{if } t \text{ then } (S; \text{while } t \text{ do } S) \text{ else skip}$
3. Formalize the property of that a language is *deterministic* (i.e., that the same inputs to a program in the language always produce the same results). Write two versions:
 - (a) One that only speaks of terminating computations, where “results” consist of final memory outputs after termination.
 - (b) One that, besides considering final memory outputs after termination, also considers non-termination as a possible “result”.

4 Tool development (continuation) - Policy and Labelling

In order to develop a tool for detecting vulnerabilities by tracking illegal information flows, it is necessary to define the information flow policy that captures what are illegal flows. This *policy* will need to be configured according to the vulnerabilities that should be detected, and how they should be recognized, i.e., what are the vulnerability patterns¹ that it should look for.

Furthermore the tool will need to be able to label resources – say, variables – in the program with labels² that summarize the security related history of the data that the variable holds. Since this labelling may change throughout the program according to what is in the variables at each point, the tool will need to maintain a *labelling* map from variable names to labels.

1. Develop a class *Policy*, representing an information flow policy, that uses a pattern data base for recognizing illegal information flows. It should include at least the following basic operations:
 - (a) Constructor of a *Policy* object, receiving as input the patterns to be considered.
 - (b) Selectors for returning the vulnerability names that have a given name as a source, those that have a given name as a sanitizer, and those that have a given name as a sink.
 - (c) Operation that, given a name and a multilabel that describes the information that is flowing to a certain name, determines the corresponding illegal flows, i.e., which part of the multilabel has the given name as a sink. It should return a multilabel that is like the one received as parameter, but that only assigns labels to patterns for which an illegal flow is taking place.
2. Develop a class *MultiLabelling*, that represents a mapping from variable names to multilabels. It should include at least the following basic operations:
 - (a) Constructor of a *MultiLabelling* object, that enables to map variable names to multilabels.
 - (b) Selectors for returning the multilabel that is assigned to a given name.
 - (c) Mutator for updating the multilabel that is assigned to a name.
3. Develop a class *Vulnerabilities*, that is used to collect all the illegal flows that are discovered during the analysis of the program slice. It should include at least the following basic operations:
 - (a) Constructor of a *Vulnerabilities* object, that enables to collect all the relevant info on the illegal flows that are found, organized according to vulnerability names.

¹See section 4, question 1, of the first exercise sheet and the *Pattern* class.

²See section 4, questions 2 and 3, of the first exercise sheet and the *Label* and *MultiLabel* classes.

- (b) Operation that given a multilabel and a name, which represents detected illegal flows – the multilabel contains the sources and the sanitizers for the patterns for which the name is a sink and the flows are illegal) – saves them in a format that enables to report vulnerabilities at the end of the analysis.