Definition of Security Properties – Exercises

Ana Matos
Software Security 2023/2024, IST

December 6, 2023

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 Noninterference**, 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), includinng 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.

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

8. $x := y; \text{while } x \neq 0 \text{ do } x := x - 1$

9. $x := 0; \text{while } x \neq y \text{ do } x := x + 1$

Answer: Solutions for the *High (H) - Low (L)* policy for confidentiality where the variable x has level L and variable y has level H:

1. $y := x; x := y + 1$
   Answer: Satisfies Deterministic Input-Output Noninterference. Ater execution of the program, the final value of x is always the result of adding 1 to its original value. Therefore, two executions with the same initial value for x will always produce the same final value for x.

2. $x := 42; x := y$
   Answer: Does not satisfy Deterministic Input-Output Noninterference. Consider two initial memories that coincide in x but not in y.

3. $x := y; x := 42$
   Answer: Satisfies Deterministic Input-Output Noninterference. After execution of the program, the value of x is always 42.

4. $\text{if } y = 0 \text{ then } x := 42 \text{ else } x := 42$
   Answer: Satisfies Deterministic Input-Output Noninterference. (Same as 1(b)iii.)

5. $\text{if } x = 0 \text{ then } x := 42 \text{ else } x := 4242$
   Answer: Satisfies Deterministic Input-Output Noninterference. Two executions that start on initial memories where x has value 0, will terminate with x having value 42. Otherwise they will terminate with x having value 4242.

6. $x := y; \text{if } x = 0 \text{ then } x := 42 \text{ else } x := 4242$
   Answer: Does not satisfy Deterministic Input-Output Noninterference. Consider two initial memories that coincide in x but, but in one y has initial value 0, and in the other not.

7. $x := 0; \text{while } y = 0 \text{ do } x := 1$
   Answer: Satisfies Deterministic Input-Output Noninterference. When the execution terminates, the value of x is always 0.

8. $x := y; \text{while } x \neq 0 \text{ do } x := x - 1$
   Answer: Satisfies Deterministic Input-Output Noninterference. (Same as 1(b)vii.)

9. $x := 0; \text{while } x \neq y \text{ do } x := x + 1$
   Answer: Does not satisfy Deterministic Input-Output Noninterference. Consider two initial memories that coincide in x but, but in one y has initial value 0, and in the other greater than 0.

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) $y := 1; \text{while } x \neq 1 \text{ do } (y := y \times x; x := x - 1)$
   on a memory such that x has the value 3.
   Answer: See book "Semantics with applications" by Nielson and Nielson, pages 23-25.
   For ρ such that $\rho(x) < 1$ there is no derivation tree.

   (b) $z := 0; \text{while } y \leq x \text{ do } (z := z + 1; x := x - y)$
   on a state where x has the value 17 and y has the value 5.
   Answer: Similar to 1a. For ρ such that $\rho(y) \leq 0$ and $\rho(x) \geq \rho(y)$ there is no derivation tree.

   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) $\text{while } x \neq 1 \text{ do } (y := y * x; x := x - 1)$

   Answer: Terminates for $x \geq 1$.
   Justification: The semantics of the while construct requires the construction of a derivation tree for the sequential composition of the body of the loop, followed by the same loop, when the guard expression has value True. If the guard is initially True – in this case if $\rho(x) \neq 1$ – it can only become False for the "innter" loop by effect of the executio of the body. In this case, the body decreases the value of x. Therefore, if initially $\rho(x) < 1$, the derivation tree for the loop would be unfolded an infinite number of times into the same loop. On the other hand, when the guard of the loop is False, its derivation tree is determined by direct application of the corrresponding rule, thus "terminating" the construction of the derivation tree for the original program.

   (b) $\text{while } 1 \leq x \text{ do } (y := y * x; x := x - 1)$
   Answer: Always terminates. (Similar justification as for 2a.)

   (c) $\text{while } 1 = 1 \text{ do skip}$
   Answer: Always loops. (Similar justification as for 2a.)

   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. 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`.

   Answer: See explanations in the recorded class.

   (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.

   Answer:

In `imp.py`:

```
indent = 0     ### INDENT
ast.eval(env,indent)
```

In `imp_ast.py`, example for statement `CompoundStatement`:

```
def eval(self, env, indent):
    toprint = "    "*indent + "< " + repr(self) + ", " + str(env) + " > --> "
    self.first.eval(env, indent+1)     ### print components with more indentation
    self.second.eval(env, indent+1)
    print toprint + str(env)           ### print transition
```

In `imp_ast.py`, example for expression `VarAexp`

```
def eval(self, env, indent):
    toprint = "    "*indent + "[[ " + repr(self) + ", " + str(env) + " ]] = "
    if self.name in env:
        print toprint + str(env[self.name])  ### print evaluation of a variable
        return env[self.name]
    else:
        print 0
        return 0
```

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 $\langle S, \rho_1\rangle \to \rho'_1$ implies $\langle S, \rho_2\rangle \to \rho'_2$ 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 $\langle S, \rho_1\rangle \to \rho'_1$ and $\langle S, \rho_2\rangle \to \rho'_2$, 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 $\langle S, \rho_1\rangle \to \rho'_1$ and $\langle S, \rho_2\rangle \to \rho'_2$, implies $\rho'_1(x) = \rho'_2(x)$.

(a) Which of the properties imply Deterministic Input-Output Noninterference and Possibilistic Input-Output Noninterference?
Answer: Property A is Possibilistic Input-Output Noninterference. Property C is Deterministic Input-Output Noninterference. Property A implies Property C. Property B is the negation of Property C. If Property B holds then Property A will not.

(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.
Answer:

i. $y := x; x := y + 1$
Answer: Property A and C hold (but not Property B). For all ρ, execution of the program always terminates and produces a memory ρ' such that $\rho'(x) = \rho(x) + 1$. Therefore, two executions starting with ρ_1 and ρ_2 such that $\rho_1(x) = \rho_2(x)$ imply that $\langle S, \rho_1\rangle \to \rho'_1$ and $\langle S, \rho_2\rangle \to \rho'_2$ with $\rho'_1(x) = \rho'_2(x)$.

ii. $x := 42; x := y$
Answer: Property B holds. Consider ρ_1, ρ_2 such that $\rho_1(y) \neq \rho_2(y)$.

iii. $x := y; x := 42$
Answer: Property A and C hold (but not Property B). After execution of the program, the value of x is always 42.

iv. if $y = 0$ then $x := 42$ else $x := 42$
Answer: Proposition 1 holds. (Same as 1(b)iii.)

v. if $x = 0$ then $x := 42$ else $x := 4242$
Answer: Property A and C hold (but not Property B). If $\rho_1(x) = \rho_2(x) = 0$ then $\rho'_1(x) = \rho'_2(x) = 42$. Otherwise, if $\rho_1(x) \neq \rho_2(x) = 0$ then $\rho'_1(x) = \rho'_2(x) = 4242$.

vi. $x := y;$ if $x = 0$ then $x := 42$ else $x := 4242$
Answer: Property B holds. Consider ρ_1, ρ_2 such that $\rho_1(y) = 0$ and $\rho_2(y) \neq 0$.

vii. $x := 0;$ while $y = 0$ do $x := 1$
Answer: Property C holds (but not Property B). When the execution terminates, the value of x is always 0.
Property A does not hold. For ρ_1, ρ_2 such that $\rho_1(y) = 0$ and $\rho_2(y) \neq 0$, while the program can terminate with $\rho_1(y) \neq 0$ and $\rho'_1(y) = 0$, it cannot with $\rho_1(y) = 0$.

viii. $x := y;$ while $x \neq 0$ do $x := x - 1$
Answer: Property C holds (but not Property B), and Property A does not hold. (Same as 1(b)vii.)

ix. $x := 0; \text{while } x \neq y \text{ do } x := x+1$
Answer: Property B holds. Consider ρ_1, ρ_2 such that $\rho_1(y) = 0$ and $\rho_2(y) > 0$.

2. Two statements S_1 and S_2 are said to be *semantically equivalent* when for all states ρ and ρ' we have that

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

Show whether the following pairs of statements are semantically equivalent.

(a) $S_1; S_2$ and $S_2; S_1$
Answer: The two statements are not semantically equivalent in general. Consider $x := 1; x := 0$ and $x := 0; x := 1$.

(b) $S_1; (S_2; S_3)$ and $(S_1; S_2); S_3$
Answer: Suppose that $\langle S_1; (S_2; S_3), \rho \rangle \to \rho'$. Then there is a derivation tree that shows it, and that includes transitions of the form (1) $\langle S_1, \rho \rangle \to \rho_1$, (2) $\langle S_2, \rho_1 \rangle \to \rho_2$ and (3) $\langle S_3, \rho \rangle \to \rho'$. Using these, we can construct a derivation tree for the transition $\langle (S_1; S_2); S_3, \rho \rangle \to \rho'$.
The implication in the opposite direction can be proved similarly.

(c) (advanced) while t do S and if t then $(S; \text{while } t \text{ do } S)$ else skip
Answer: See book "Semantics with applications" by Nielson and Nielson, pages 26-27.

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 terminatio, 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 patterns[1] that it should look for.

Furthermorem the tool will need to be able to label resources – say, variables –in the program with labels[2] that summarize the security related history of the data that the

[1] See section 4, question 1, of the first exercise sheet and the Pattern class.
[2] See section 4, questions 2 and 3, of the first exercise sheet and the Label and MultiLabel classes.

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 are being considered, those 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.
    (b) Operation that given 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 the fact them in a format that enables to report vulnerabilities at the end of the analysis.