Automatic Enforcement of Security Properties – Exercises

Ana Matos
Software Security 2023/2024, IST

December 6, 2023

Goals

- To understand the **gap** between what can be expressed as **semantic properties**, and what can be enforced as an **automatic mechanism**.
- To gain intuitions on the **role of security labellings** and **policies** in the design and implementation of enforcement mechanisms.
- To understand how to set up a **basic infra-structure for a security analysis** based on labelling program components.
- To develop the basic **components of a tool** for **inspecting the Abstract Syntax Tree** of a program.
- To understand how possible program **traces** relate to the program's code.

1 Semantic property vs. Enforcement mechanism

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, each of the programs below, and the security property Deterministic Input-Output Noninterference.

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 $y = 0$ then $x := 42$ else $x := 4242$
6. if $x = 0$ then $x := 42$ else $x := 4242$

7. $x := y; \text{if } x = 0 \text{ then } x := 42 \text{ else } x := 4242$

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

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

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

1. The following *"Rules of thumb"* define a mechanism for accepting or rejecting programs of the WHILE language:

   - The "reading level" of an arithmetic expression or test is H if it contains a variable of level H. Otherwise it has level L.
   - You cannot assign an expression of "reading level" H to a variable of level L.

   (a) For each of the programs above, state whether rejecting programs according to the following *"Rules of thumb"* would produce false negatives (reject insecure programs), false positives (reject secure programs), with respect to (Deterministic Input-Output) Noninterference.

   (b) Is the above defined mechanism sound/complete/precise?

2. Repeat the previous question, but now consider the additional rules:

   - You cannot assign to a variable of level L, while in the branch of a conditional whose test has "reading level" H.
   - You cannot assign to a variable of level L, while in the body of a loop whose test has "reading level" H.

3. Write the analogous *"Rules of thumb"* for Integrity, and answer the corresponding questions above.

2 Information flow policies for mechanisms

Consider the following lattice-based information flow policies:

1. *High (H) - Low (L)* security policy for confidentiality and for integrity, where the variable x has level L and variable y has level H
2. Principal-based policy for confidentiality
3. Vulnerability policy for integrity, where classes are sets of vulnerability sources that might have tainted the information held by an object.

Define and implement an abstract data type Policy, with the following basic operations:

- set of security levels

- can-flow relation
- least upper bound operator
- greatest lower bound operator
- bottom element
- top element

Answer: In Python, policy 1:

```
class HLCPolicy():

    def get_sec_classes(self):
        return {"L", "H"}

    def can_flow(self, lab1, lab2):
        return lab1=="L" or lab2=="H"

    def glb(self, lab1, lab2):
        if lab1=="L" or lab2=="L" :
            return "L"
        else :
            return "H"

    def lub(self, lab1, lab2):
        if lab1=="H" or lab2=="H" :
            return "H"
        else :
            return "L"

    def bottom(self):
        return "L"

    def top(self):
        return "H"
```

3 Labelings

Consider the following off-the-shelf interpreter of a simple imperative language, which we worked with in the lab class about *Semantics and Properties*:

`http://jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python-part-1`.[1]

Modify the purpose of the program, so that instead of interpreting (executing) programs, it prints a tree with labellings of all the syntactic components. Produce two different versions:

[1]To run it using Python3 you can include the module 'functools'.

1. Assume that the variable `env` in `imp.py` represents a complete mapping from all variables to **confidentiality** security labels. A label should be given to each syntactic component, according to the following:

   - Expressions should be given an upper bound to the levels of the variables that appear in them.
   - Statements should be given a lower bound to the levels of the variables that are assigned in them.

2. Assume that the variable `env` in `imp.py` represents a partial mapping from all "input variables" (those whose name begins with the prefix `in_`) to **integrity** security labels. In this case the labels of the remaining variables should be inferred with the least restrictive label possible. Label should be given to each syntactic component, according to the following:

   - Expressions should be given an upper bound to the levels of the variables that appear in them.
   - Assignments should be given the (new updated) level of the variable that is being written, which should match that of the expression that is being read.

Tips:

- Inspired in the `eval` method, define a `classify` method so that instead of performing the evaluation of the program, it instead prints program components and their corresponding security labels.
- The methods for performing the evaluation, for each form of statement and expression, can be found in `imp_ast.py`.
- Add a parameter to the `eval` method of statements and expressions in order to achieve indentation according todepth of nesting.
- Use the `repr` method to print the statement or expression.

Answer: In `imp.py`:

```
labenv = {"n":"L", "p":"H"}   # Label environment.  Could also be read from the command line, etc.

policy = HLCPolicy()    # High-Low Confidentiality Policy
indent=0
ast.classify(labenv, policy, indent)
```

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

```
def classify(self, labenv, policy, ident):
    lab1=self.first.classify(labenv, policy, ident+1)
    lab2=self.second.classify(labenv, policy, ident+1)
    lab=policy.glb(lab1,lab2)
    print("    "*ident + repr(self) + ": " + lab)
    return lab
```

In `imp_ast.py`, example for expression `RelopBexp`:

```
def classify(self, labenv, policy, ident):
    lab1 = self.left.classify(labenv, policy, ident+1)
    lab2 = self.right.classify(labenv, policy, ident+1)
    lab=policy.lub(lab1,lab2)
    print("    "*ident + repr(self) + ": " + lab)
    return lab
```

4 Tool development (continuation) - AST traversal

Analysing information flows in a program requires knowledge about the program's possible behaviours. These behaviors are fully encoded in the program, and can be inspected even without executing the program, as is done in statical analysis. This is usually done over the programn's *Abstract Syntax Tree (AST)*, as it captures all the relevant information that appears in the code, in a structured way.

When performing static analysis, the actual path that the program will take is not always knwon. Therefore, when tracking information flows that are encoded in a program, it might be necessary to consider multiple paths that split and converge according to the constructs encoded in the program.

As an example to see an AST for a real world language, you can use Python's `ast` module, to obtain a tree of objects whose classes all inherit from `ast.AST` (`https://docs.python.org/3/library/ast.html`). The following exercises are proposed for the Python language, but you can also look into ASTs for JavaScript (`https://esprima.org/`) or PHP (`https://gitlab.rnl.tecnico.ulisboa.pt/ssof2223/project/project/-/blob/master/slice-parser.php`) for example.

1. Write a program that takes a program written in Python, extracts its AST, and outputs the result in the JSON format. In Python you can import the `ast` module, and use `astexport.export.export_dict` from `astexport`, which takes a Python AST and returns its representation in JSON. You can visualize the result as a tree using this online tool: `http://jsonviewer.stack.hu/`.

2. Write a function that traverses an AST, and prints, for each node, its node type (field "`ast_type`") and the line number of where it starts. Make sure you have a recurive function.

   Answer:

   ```
   with open(filename,"r") as fp:
     py_str=fp.read()
     ast_py=ast.parse(py_str)
     ast_dict=astexport.export.export_dict(ast_py)

   print(json.dumps(ast_dict))
   ```

3. Write a function that traverses an AST, and prints its sets of complete traces, i.e., a representation of all possible execution paths. To tackle the cases where the

number of possible paths is infinite (think of the while loop), you can assume a constant maximum number of repetitions that the loop can do. For time reasons, choose only a subset of the possible node types, such as the ones that capture the WHILE language constructs. Proceed by case analysis on the node type.

4. Extend the MultiLabelling class developed last week in order to tackle the analysis of possible execution paths the split and converge. To this end, add the following basic operations:

   (a) Operation for creating and returning a (deep) copy of a multilabelling.

   (b) Combinor for returning a new multilabelling where multilabels associated to names capture what might have happened if *either* of the multilabellings hold.