Software Security - Trace Properties, Self-Composition and Symbolic Execution – Exercises

José Fragoso Santos Ana Matos
Software Security 2023/2024, IST

December 19, 2023

Aim To become familiar with basic notations and concepts for defining trace-based properties. To understand the concepts of safety and liveness properties.

To become acquainted with symbolic execution mechanism for verifying security properties, and dually, for finding security bugs.

To know how to apply the self-composition technique for verifying Noninterference.

1 Trace Properties

1. Recall the approach for defining trace properties introduced in the lecture. For each of the following properties, give the formal definition of the following trace properties:
   (a) *AllZero*, describing the traces that terminate with a state mapping all of its variables to zero;
   (b) *MonX*, describing all traces such that the value of the program variable x never gets decremented;
   (c) *AllMon*, describing all traces such that the values of all program variables never get decremented;
   (d) *XGreaterThanY*, describing all traces such that, whenever both variables x and y are defined, x is greater than y;
   (e) *XGreaterThanAll*, describing all traces such that, whenever x is defined, it is greater than all other program variables;
   (f) *YBookKeepX*, describing all traces such that the variable x is only allowed to swap sign once, and, if it does, the variable y will eventually be set to the old value of x and will keep that value until the execution finishes.
2. Which of the properties above are liveness properties and which are safety properties? Justify your answer.

3. For each of the above properties, write a WHILE language statement that satisfies the property and a WHILE language statement that does not. For the statement that does not, give a program trace that exhibits the bug.

2 Self-Composition and Symbolic Execution

1. Consider the following WHILE statements:

   (a) if h then $l := l + z$ else skip

   (b) if h then $x := x + z$ else skip; if $!h$ then $y := y + z$ else skip;
   $l := x + y; x := 0; y := 0$

2. Use self-composition and symbolic execution to check which of the above programs satisfy non-interference. Assuming a High-Low confidentiality policy, and a security labeling Γ such that $\Gamma(h) = H$ and $\Gamma(l) = \Gamma(x) = \Gamma(y) = \Gamma(z) = L$.

3. Which of the programs above would be considered secure by a standard type system for information flow control? Which would be considered insecure? What can be concluded about the precision of self-composition + symbolic execution when compared to type systems for information flow control?

3 Tool development (conclusion) - implicit flows

In order to develop a *sound* tool for detecting illegal information flows, it is necessary to tackle *implicit flows*, e.g. for the proposed Python subset, those that result from dependencies between assignments and the information that is tested in conditions. To track them, it will be necessary to determine the security class that captures the information that is tested in all ifs and whiles that dominate the assignments, or in other words, that captures the information that is associated to the knowledge of what point of the program that is being considered (think of the program counter).

1. Extend the class Policy with an operation that, given a name and a multilabel that describes the information that is flowing from the program counter, determines the corresponding illegal flows, i.e., which part of the multilabel *is concerned with implict flows* and 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. Extend your function that walks ASTs corresponding to expressions, so that it also receives a security class corresponding to the program counter, and takes it into consideration when detecting illegal flows that arrive to potential sinks.

3. Extend your function that walks ASTs corresponding to statements, so that it also receives a security class corresponding to the program counter, and takes it into consideration when analysing instructions that could establish a flow to a sink, and updates it for recursive calls for analysing code whose execution depends on additional conditions.

4. Define a function that given the illegal flows that have been detected dyring the traversal of the AST, and which are collected by the Vulnerabilities class, outputs all the vulnerabilities that were detected, specifying its sources, sinks and possible sanitizers.