

# Automata-Based Confidentiality Monitoring<sup>\*</sup>

Gurvan Le Guernic<sup>1,2</sup>, Anindya Banerjee<sup>2</sup>,  
Thomas Jensen<sup>1</sup>, and David A. Schmidt<sup>2</sup>

<sup>1</sup> IRISA - Campus universitaire de Beaulieu, 35042 Rennes - France  
{Gurvan.Le\_Guernic, jensen}@irisa.fr

<sup>2</sup> Kansas State University - Manhattan, KS 66506 - USA  
{ab,schmidt}@cis.ksu.edu

**Abstract.** Non-interference is typically used as a baseline security policy to formalize confidentiality of secret information manipulated by a program. In contrast to static checking of non-interference, this paper considers dynamic, automaton-based, monitoring of information flow for a single execution of a sequential program. The monitoring mechanism is based on a combination of dynamic and static analyses. During program execution, abstractions of program events are sent to the automaton, which uses the abstractions to track information flows and to control the execution by forbidding or editing dangerous actions. The mechanism proposed is proved to be sound, to preserve executions of well-typed programs (in the security type system of Volpano, Smith and Irvine), and to preserve some *safe* executions of ill-typed programs.

## 1 Introduction

With the intensification of communication in information systems, interest in security has increased. This paper deals with the problem of confidentiality, more precisely with *non-interference* in sequential programs. This notion has first been introduced by Goguen and Meseguer [1] as the absence of *strong dependency* [2].

A sequential program,  $P$ , is said to be *non-interfering* if the values of its public (or low) outputs do not depend on the values of its secret (or high) inputs. Formally, non-interference of  $P$  is expressed as follows: given any two initial input states  $\sigma_1$  and  $\sigma_2$  that are indistinguishable with respect to low inputs, the executions of  $P$  started in states  $\sigma_1$  and  $\sigma_2$  are *low-indistinguishable*; i.e. there is no observable difference in the public outputs. In the simplest form of the *low-indistinguishable* definition, public outputs include only the final values of low variables. In a more general setting, the definition may additionally involve intentional aspects such as power consumption, computation times, etc.

Static analyses for non-interference have been studied extensively and are well surveyed by Sabelfeld and Myers [3]. The *novelty* of the approach developed in this paper lies in:

---

<sup>\*</sup> Banerjee and Le Guernic were partially supported by NSF grants CNS-0627748, CNS-0209205, CCF-0296182 and ITR-0326577. Schmidt was partially supported by NSF grants ITR-0326577 and ITR-0086154. Le Guernic was also partially supported by the *PoTestAT* project (*ACI Sécurité*).

1. its ability to give a judgment for a *single execution alone* and not only for *all the executions of a program as a whole*,
2. the monitoring mechanism used to ensure the confidentiality of secret data.

The bulk of previous research [4, 5, 6, 7, 8, 9, 10, 11] associates the notion of non-interference to a program and develops static analyses that accept a *program* only if *all its executions* ensure the confidentiality of secrets. In contrast, this paper presents a *dynamic analysis that uses the results of a static analysis*: the dynamic analysis accepts or rejects a *single execution* of a program without necessarily doing the same for all other executions. The monitoring mechanism introduced guarantees confidentiality of secret data: either the monitor deduces that the current execution is non-interfering or it alters the behavior of the execution to obtain a non-interfering execution. The feasibility of this approach is shown by Hamlen et al. [12] who prove that any policy that can be statically asserted is enforceable using monitors which have access to the program's source.

There are three main benefits to using a monitoring mechanism rather than a static analysis. First, the security levels of inputs and outputs may be different from one execution to another; and a monitoring mechanism lets the security levels vary before each execution while still enforcing non-interference. For example, consider the effect of monitoring the Unix command `more` that takes as input a file divided into blocks and displays the blocks sequentially while waiting for the user to press a key between each block. A monitor for `more` would display a block only if the security level of the block is lower than the security level of the user; otherwise a default security message will be displayed. The monitor behavior depends on the particular file given as input, *not on all possible inputs*. This feature makes the monitoring mechanism a “lazy” polyvariant static analysis. A second benefit of monitors is their ability to safely use a program which has not been proved to respect a given property — maybe because one of its executions does not respect it; the monitor can run the *safe executions* — i.e. non-interferent executions — of an *unsafe program*. Finally, a monitoring mechanism follows the precise control flow of a program and thus calculation of control dependences (as might be performed in static analyses) can be more accurate. Section 5 contains an example showing how the work presented in this paper benefits from this improved accuracy.

A distinguishing feature of this dynamic analysis, compared to other program monitors, lies in the property overseen. Monitoring information flow is more complicated than, *e.g.*, monitoring divisions by zero, since it must take into account not only the current state of the program but also the execution paths *not taken* during execution. For example, executions of the following programs (a) **if  $h$  then  $x := 1$  else skip** and (b) **if  $h$  then skip else skip** in an initial state where  $h$  is **false** are equivalent concerning executed commands. In contrast, (b) is obviously a non-interfering program, while (a) is not. The execution of (a), with a low-equivalent initial state where  $h$  is **true** and  $x$  is 0, does not give the same final value for the low output  $x$ .