Automata-based Confidentiality Monitoring

Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, David Schmidt

Automata-based Confidentiality Monitoring *

Gurvan Le Guernic[1,2], Anindya Banerjee[2],
Thomas Jensen[1], and David A. Schmidt[2]

[1] IRISA - Campus universitaire de Beaulieu, 35042 Rennes - France
{Gurvan.Le_Guernic, jensen}@irisa.fr
[2] 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 σ_1 and σ_2 that are indistinguishable with respect to low inputs, the executions of `P` started in states σ_1 and σ_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:

* 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 [4–11] 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.

This paper presents the semantics of a confidentiality monitoring mechanism which is proved to be sound. This mechanism is useful, for example, for the exe-

cution of programs downloaded from untrusted sources. Based on the semantics presented here, the monitoring mechanism can be implemented as a program transformation or as a virtual machine. To the best of our knowledge, this work is the first that presents a non-interference monitoring mechanism supported by formal proofs of soundness. Because of its dynamic nature, the mechanism is more expressive than the non-interference type system of Volpano et al. [11]. The next section defines terminology and introduces the scope of the work. Section 3 defines the monitoring semantics, which is automata-based. The properties of monitored executions and a comparison with type systems are contained in Sect. 4. Section 5 discusses related work and concludes. Proofs of theorems in this paper can be found in the companion technical report [13].

2 Language and Non-interference Monitoring Principles

The work presented aims at monitoring executions in order to enforce *non-interference*. The novelty and difficulty of this objective, compared to standard works on monitoring, lies in the monitoring mechanism. Monitors which see executions as a sequence of executed actions, like standard execution monitors [14], are not sufficient to enforce non-interference. This particular point is supported by works of McLean and Schneider. McLean [15] proved that information flow policies equivalent to non-interference are not *properties*; and Schneider [14] concluded that execution monitors are limited to the enforcement of *properties*. An information theoretic viewpoint why this is the case is given by Ashby:

> "the information carried by a particular message depends on the set it comes from. The information conveyed is not an intrinsic property of the individual message." [16, § 7/5 page 124].

In order to enforce strong constraints on the information flow (like non-interference), the monitoring mechanism must be aware of the commands that are *not* evaluated by a given execution [17, Sect. 4.2.2]. This is part of the approach taken in the work presented in this paper. Conceptually, the monitor has a partial access to the target program's source code.

The notion of non-interference is intrinsically linked to the notion of information flow. This paper distinguishes three types of information flows: *direct flows* from the right part to the left part (or target) of assignments, *explicit indirect flows* from the *context* of execution to targets of *executed* assignments and *implicit indirect flows* from the *context* of execution to targets of *un-executed* assignments. The *context* of execution of a command is the set of entities that influence whether this command is executed or not.

The language: syntax and semantics. We study a simple imperative sequential language with loops and outputs. The work proposed in this paper can be extended to more complex languages. The main requirement is to have a precise

knowledge of the control flow. The grammar of the studied language follows:

$$
\begin{aligned}
A &::= x := e \mid \textbf{skip} \mid \textbf{output}\ e \\
B &::= \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end} \mid \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \\
S &::= S\ ;\ S \mid B \mid A
\end{aligned}
$$

The only constraints put on expressions (e) are that their evaluation must be deterministic and without side effects. Statements (S) are either sequences ($S\ ;\ S$), conditionals and while loops (B), or atomic actions (A). The output statement, "**output** e", is a generic statement used to represent any kind of public (or low) output, e.g., the action of printing the value of expression e on the terminal, producing a sound, or laying out a new window on the desktop. Only public outputs (i.e. outputs that are visible by standard users) are coded with the output statement; secret outputs are simply ignored. For example, sending a message m on a public network is represented by "**output** m," but sending an encrypted message n on a public network is abstracted by "**output** c", where c is a constant which emphasizes the fact that the content of an encrypted message cannot be revealed. Finally, sending a message on a private network, to which standard users do not have access, does not appear in the code of the programs studied.

Non-terminating executions leaks information through timing channels; such channels are not in the scope of this work. Consequently, the paper focuses on terminating executions, which allows the use of big-step operational semantics (also called *natural semantics* [18]). The standard semantics of the language (Fig. 2) is described using evaluation rules, written $\sigma \vdash S \overset{o}{\Rightarrow} \sigma'$. This reads as follows: statement S executed in state σ yields state σ' and *output sequence* o. Let $\mathbb{D}$ be the semantic domain of values, and $\mathbb{X}$ the domain of variables. The definition of program states ($\mathbb{X} \to \mathbb{D}$) is extended to expressions, so that $\sigma(e)$ is the value of the expression e in state σ. An output sequence is a word in $\mathbb{D}^\star$.

$$
\sigma \vdash x := e \overset{\epsilon}{\Rightarrow} \sigma[x \mapsto \sigma(e)] \qquad \sigma \vdash \textbf{output}\ e \overset{\sigma(e)}{\Longrightarrow} \sigma \qquad \sigma \vdash \textbf{skip} \overset{\epsilon}{\Rightarrow} \sigma
$$

$$
\frac{\sigma \vdash S_1 \overset{o_1}{\Longrightarrow} \sigma' \quad \sigma' \vdash S_2 \overset{o_2}{\Longrightarrow} \sigma''}{\sigma \vdash S_1\ ;\ S_2 \overset{o_1\ o_2}{\Longrightarrow} \sigma''} \qquad \frac{\sigma(e) = v \quad \sigma \vdash S_v \overset{o}{\Rightarrow} \sigma'}{\sigma \vdash \textbf{if}\ e\ \textbf{then}\ S_{\texttt{true}}\ \textbf{else}\ S_{\texttt{false}}\ \textbf{end} \overset{o}{\Rightarrow} \sigma'}
$$

$$
\frac{\sigma(e) = \texttt{true} \quad \sigma \vdash S\ ;\ \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o}{\Rightarrow} \sigma'}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o}{\Rightarrow} \sigma'} \qquad \frac{\sigma(e) = \texttt{false}}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{\epsilon}{\Rightarrow} \sigma}
$$

Figure 1. Semantics outputting the values of low-outputs

The monitoring principles. Non-interference formalizes that there is no information flow from secret (or high) inputs to public (or low) outputs. For any program P, let $\mathcal{S}(\texttt{P}) \subseteq \mathbb{X}$ be the set of variables whose initial values are the

secret inputs. The only public output is the output sequence resulting from the execution. Contrary to the majority of works on non-interference, the values of the variables in the program state are never *directly* accessible (even at the end of the execution). Consequently, the values of the variables in the program state are never considered public outputs. We made this choice in order to be more flexible with regard to what is considered as publicly accessible. All, and only, the values which are visible to low users must be displayed using an output statement at the position inside the program where they are visible. If the desired behavior is that low users have access to the values of low variables at the end of the execution, those values must be output at the end of the program.

The main monitoring mechanism principle is based on information transmission notions of classical information theory [16]. Cohen states it as follows:

> "information can be transmitted from a to b over execution of H [(a sequence of actions)] if, by suitably varying the initial value of a (exploring the variety in a), the resulting value in b after H's execution will also vary (showing that the variety is conveyed to b)." [2, Sect. 3].

Hence, for preventing information flows from secret inputs to public outputs, the monitoring mechanism must ensure that variety — which can be seen as the property of variability or mutability — of the initial values of the variables in the set $\mathcal{S}(\mathtt{P})$ is not conveyed to the output sequence. This means that the monitoring mechanism, which works on a single execution, must ensure that even if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ were different, the output sequence would be identical.

The monitoring automaton has two jobs. The first is to track "variety," that is, to track entities (program variables, program counter, ...) having different values when the initial values of variables in $\mathcal{S}(\mathtt{P})$ are different. Its second job is to prevent conveying of variety to the output sequence, that is, to ensure that the output sequence would be identical for any execution having the same public inputs (the initial values of the variables not in $\mathcal{S}(\mathtt{P})$). To complete the first job, the states of the monitoring automaton are pairs. The first element of this pair is a set of variables. At any step of the computation, it contains all the variables that have "variety" (i.e., have a different value if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ are different). The second element of the pair is a word in $\{\top, \bot\}^\star$. This word tracks "variety" in the context of the execution (the value of the program counter). The second job (avoiding transfer of variety to the output sequence) is accomplished by authorizing, denying, or editing output statements depending on the current state of the monitoring automaton.

3 Definition of the Monitoring Mechanism

The monitoring mechanism is divided into two main elements. The first is an automaton similar to *edit automata* [19]. Inputs to the automaton are abstractions of the actions accomplished during an execution. The automaton tracks information flow and authorizes, forbids or edits the actions of the monitored execution

to enforce non-interference. The second element of the monitoring mechanism is a semantics of monitored executions that merges together the behavior of the monitoring automaton and that of the standard output semantics in Fig. 2.

3.1 The Automaton

The automaton's transition function is independent of the monitored program, but the initial automaton's state is not. The automaton enforcing non-interference is the tuple $\mathcal{A}(\mathtt{P}) = (Q, \Phi, \Psi, \delta, q_0)$ where: Q is a set of states ($Q = 2^{\mathbb{X}} \times \{\top, \bot\}^\star$), Φ is the input alphabet, Ψ is the output alphabet, δ is a transition function ($Q \times \Phi) \longrightarrow (\Psi \times Q$), and $q_0 \in Q$ is the start state ($q_0 = (\mathcal{S}(\mathtt{P}), \epsilon)$).

An automaton state is a pair (V, w). $V \subseteq \mathbb{X}$ contains all the variables whose current value *may* have been influenced by the initial values of the variables in $\mathcal{S}(\mathtt{P})$; w, which belongs to $\{\top, \bot\}^\star$, can be seen as a stack that tracks variety in the context of the execution. In our approach, the context consists of only the program counter's value. If $\top$ occurs in w, then the statement executed belongs to a conditional whose test may have been influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Hence the statement may not have been executed for a different choice of initial values of $\mathcal{S}(\mathtt{P})$. The input alphabet of the automaton (Φ) consists of abstractions of events that occur during an execution. It is defined below. The output alphabet (Ψ) is composed of the following: ACK, OK, NO, and atomic actions of the language. An atomic action is the answer of the monitoring automaton whenever an action other than the current one has to be executed.

Figure 2 specifies the transition function of the automaton. A transition is written $(q, \phi) \xrightarrow{\psi} q'$. It reads as follows: in the state q, on reception of the input ϕ, the automaton moves to state q' and outputs ψ. Let $\mathcal{V}(e)$ be the set of variables occurring in e and *modified*(S) be the set of all variables whose value may be modified by an execution of S.

Figure 2 shows that the automaton forbids (NO) or edits (**output** θ) only executions of output statements. For other inputs, it merely tracks, in the set V, the variables that may contain secret information (have *variety*), and it tracks, in w, the *variety* of the branching conditions.

Inputs "`branch` e" are generated at exit point of conditionals. On reception of such inputs in state (V, w), the automaton checks if the value of the branching condition (e) might be influenced by the initial values of $\mathcal{S}(\mathtt{P})$: only if some variable occurring in e belongs to V. If this is the case then the automaton pushes $\top$ at the end of w; otherwise it pushes $\bot$. In either case, the automaton acknowledges the reception of the input by outputting ACK.

Whenever execution exits a branch — which is a member of a conditional c — the input "`exit`" is sent to the automaton. The last letter of w is then removed. As any such input matches a previous input "`branch` e" — generated by the same conditional c — which adds a letter to the end of w, the effect of "`exit`" is to restore the context to its state before the conditional was processed.

Inputs "`not` S" are generated at exit point of conditionals. It means that, due to the value of a previous branching condition, statement S has not been

$$
\begin{array}{l}
\mathit{modified}(x := e) = \{x\} \\
\mathit{modified}(\textbf{output}\ e) = \mathit{modified}(\textbf{skip}) = \emptyset \\
\mathit{modified}(S_1\ ;\ S_2) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2) \\
\mathit{modified}(\textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2) \\
\mathit{modified}(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathit{modified}(S)
\end{array}
$$

$$
\begin{array}{lr}
((V,w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\top) & \text{iff } \mathcal{V}(e) \cap V \neq \emptyset \\
((V,w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\bot) & \text{iff } \mathcal{V}(e) \cap V = \emptyset \\
((V,wa),\ \texttt{exit}) \xrightarrow{ACK} (V,w) & \\
((V,w),\ \texttt{not}\ S) \xrightarrow{ACK} (V \cup \mathit{modified}(S), w) & \text{iff } w \notin \{\bot\}^\star \\
((V,w),\ \texttt{not}\ S) \xrightarrow{ACK} (V,w) & \text{iff } w \in \{\bot\}^\star \\
((V,w),\ \textbf{skip}) \xrightarrow{OK} (V,w) & \\
((V,w),\ x := e) \xrightarrow{OK} (V \cup \{x\}, w) & \text{iff } w \notin \{\bot\}^\star \text{ or } \mathcal{V}(e) \cap V \neq \emptyset \\
((V,w),\ x := e) \xrightarrow{OK} (V \setminus \{x\}, w) & \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V = \emptyset \\
((V,w),\ \textbf{output}\ e) \xrightarrow{OK} (V,w) & \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V = \emptyset \\
((V,w),\ \textbf{output}\ e) \xrightarrow{\textbf{output}\ \theta} (V,w) & \text{iff } w \in \{\bot\}^\star \text{ and } \mathcal{V}(e) \cap V \neq \emptyset \\
((V,w),\ \textbf{output}\ e) \xrightarrow{NO} (V,w) & \text{iff } w \notin \{\bot\}^\star
\end{array}
$$

Figure 2. Transition function of monitoring automata

executed. This detects *implicit indirect flows*. On input "`not` S" in state (V, w), the automaton verifies whether S may have been executed with differing values for $\mathcal{S}(\mathtt{P})$. This is the case if the context of execution carries *variety* (i.e. if w does not belong to $\{\bot\}^\star$). Let (V', w') be the new state of the automaton. If the context carries *variety* then V' is the union of V with the set of variables whose values may be modified by an execution of S. Otherwise, nothing is done.

Atomic actions (assignment, skip or output) are sent to the automaton for validation before their execution. The atomic action **skip** is considered safe because the non-interference definition considered in this work is not time sensitive. Hence the automaton always authorizes its execution by outputting OK.

When executing an assignment $(x := e)$, two types of flows are created. The first is a direct flow from e to x. The second flow is an explicit indirect flow from the context of execution to x. For example, the execution of the assignment in "**if** b **then** $x := y$ **else skip end**" creates such a flow from b to x. Both forms of flows are *always* created when an assignment is executed. What is important is to check if secret information is carried by one of the flows (i.e., if variety in $\mathcal{S}(\mathtt{P})$ is conveyed by one of the flows). Hence, on input $x := e$, the automaton checks if the value of the *origin* of one of those two flows is influenced by the initial values of $\mathcal{S}(\mathtt{P})$. For instance, if b is true in "**if** b **then** $x := y$ **else skip end**", y is the origin of a direct flow to x and b is the origin of an explicit indirect flow to x. The origin of the explicit indirect flow is influenced by $\mathcal{S}(\mathtt{P})$ only if w contains $\top$, meaning that the condition of a previous (but still active) conditional

was potentially influenced by the initial values of $\mathcal{S}(\mathtt{P})$. The origin of the direct flow is influenced by $\mathcal{S}(\mathtt{P})$ only if $\mathcal{V}(e)$ and V are not disjoint. If the value of e is influenced by the initial values of $\mathcal{S}(\mathtt{P})$ then at least one of the variables appearing in e has been influenced by $\mathcal{S}(\mathtt{P})$. Such variables are members of V. Let (V', w') be the new automaton state after the transition. If the origin of either the direct flow or the explicit indirect flow is influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then x (the variable modified) is added to V: $V' = V \cup \{x\}$. On the other hand, if none of the origins are influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then x receives a new value which is not influenced by $\mathcal{S}(\mathtt{P})$. In that case, V' equals $V \setminus \{x\}$. This makes the mechanism flow-sensitive.

The rules for the automata input, "**output** e," prevent bad flows through two different channels. The first one is the actual content of what is output. In a public context, ($w \in \{\bot\}^\star$), if the program tries to output a secret (i.e., the intersection of V and the variables in e is not empty), then the value of the output is replaced by a default value. This value can be a message informing the user that, for security reasons, the output has been denied. To do so, the automaton outputs a new output statement to execute in place of the current one. The second channel is the behavior of the program itself. This channel exists because, depending on the path followed, some outputs may or may not be executed. Hence, if the automaton detects that this output may not be executed with different values for $\mathcal{S}(\mathtt{P})$ (the context carries *variety*) then *any* output must be forbidden; and the automaton outputs NO.

3.2 The Semantics

The semantics merging the standard output semantics in Fig. 2 with the monitoring automaton is given in Fig. 3. The semantics is described using evaluation rules written: $(q, \sigma) \Vdash S \stackrel{o}{\Rightarrow} (q', \sigma')$. This reads: statement S executed in automaton state q and program state σ yields automaton state q', program state σ', and output sequence o. There are three rules for atomic actions: **skip**, x **:=** e and **output** e. There is one rule for each possible automaton answer to the action executed. Either the automaton authorizes the execution (OK), denies the execution (NO), or replaces the action by another one. The rules use the standard semantics (Fig. 2) when an action must be executed. In the case where the execution is denied, the evaluation omits the current action (as if the action was "**skip**"). For the case where, on reception of input A, the monitoring automaton returns A', the monitoring semantics executes A' instead of A. Note from Fig. 2, that A' can only be the action that outputs the default value (**output** θ).

For conditionals, the evaluation begins by sending to the automaton the input "`branch` e" where e is the test of the conditional. Then, the branch designated by e is executed (in the case of a while statement whose test is `false`, the branch executed is "**skip**"). The execution follows by sending the automaton input "`not` S" where S is the branch not executed (in case of a while statement whose test is `true`, what happens is equivalent to sending the automaton input "`not` **skip**"). Finally, the input "`exit`" is sent to the automaton and the execution

proceeds as usual. In the case of a while statement with a condition equals to `true`, the execution proceeds by executing the while statement once again.

$$\frac{(q,\, A) \xrightarrow{OK} q' \qquad \sigma \vdash A \overset{o}{\Longrightarrow} \sigma'}{(q,\, \sigma) \Vdash A \overset{o}{\Longrightarrow} (q',\, \sigma')} \qquad \frac{(q,\, A) \xrightarrow{A'} q' \qquad \sigma \vdash A' \overset{o}{\Longrightarrow} \sigma'}{(q,\, \sigma) \Vdash A \overset{o}{\Longrightarrow} (q',\, \sigma')} \qquad \frac{(q,\, A) \xrightarrow{NO} q}{(q,\, \sigma) \Vdash A \overset{\epsilon}{\Longrightarrow} (q,\, \sigma)}$$

$$\frac{(q,\, \sigma) \Vdash S_1 \overset{o_1}{\Longrightarrow} (q_1,\, \sigma_1) \qquad (q_1,\, \sigma_1) \Vdash S_2 \overset{o_2}{\Longrightarrow} (q_2,\, \sigma_2)}{(q,\, \sigma) \Vdash S_1 \,;\, S_2 \overset{o_1\, o_2}{\Longrightarrow} (q_2,\, \sigma_2)}$$

$$\frac{\sigma(e) = v \quad (q,\, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \qquad (q_1,\, \sigma) \Vdash S_v \overset{o}{\Longrightarrow} (q_2,\, \sigma_1) \qquad (q_2,\, \texttt{not}\ S_{\neg v}) \xrightarrow{ACK} q_3 \quad (q_3,\, \texttt{exit}) \xrightarrow{ACK} q_4}{(q,\, \sigma) \Vdash \textbf{if}\ e\ \textbf{then}\ S_{\texttt{true}}\ \textbf{else}\ S_{\texttt{false}}\ \textbf{end} \overset{o}{\Longrightarrow} (q_4,\, \sigma_1)}$$

$$\frac{\sigma(e) = \texttt{true} \quad (q,\, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \qquad (q_1,\, \sigma) \Vdash S \overset{o_l}{\Longrightarrow} (q_2,\, \sigma_1) \qquad (q_2,\, \texttt{exit}) \xrightarrow{ACK} q_3 \qquad (q_3,\, \sigma_1) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_w}{\Longrightarrow} (q_4,\, \sigma_2)}{(q,\, \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_l\, o_w}{\Longrightarrow} (q_4,\, \sigma_2)}$$

$$\frac{\sigma(e) = \texttt{false} \quad (q,\, \texttt{branch}\ e) \xrightarrow{ACK} q_1 \qquad (q_1,\, \texttt{not}\ S) \xrightarrow{ACK} q_2 \quad (q_2,\, \texttt{exit}) \xrightarrow{ACK} q_3}{(q,\, \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{\epsilon}{\Longrightarrow} (q_3,\, \sigma)}$$

Figure 3. Semantics of monitored executions

3.3 Example of monitored execution

Table 1 is an example of monitored execution. The monitored program is given in column "Program P". Its inputs are h and l. The execution monitored is the one for which h equals `true` and l equals 22. $\mathcal{S}(\mathtt{P})$, the set of secret inputs of P, is $\{h\}$. So the initial state of the automaton is $(\{h\}, \epsilon)$. Column "input" contains the inputs which are sent to the automaton, "output" contains the output sent back to the semantics, and "new state" shows the *new* internal state of the automaton *after* the transition. Finally, the last column shows the actions which are really fulfilled by the monitored execution.

In this example, there are only two alterations of the execution (on lines 5 and 8). The first occurs when the program attempts to output a value influenced by $\mathcal{S}(\mathtt{P})$ – **output** y. At this point of the execution, the value contained in y has been influenced by the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$. This is known because y belongs to the first element of the automaton state *before* execution of line 5. Consequently, the automaton disallows the output of this value. However, the fact of outputting something in itself is safe because the context of execution (the program counter) has not been influenced by $\mathcal{S}(\mathtt{P})$ (the second element of the automaton state belongs to $\{\bot\}^\star$). Hence, the automaton replaces the current action by an output action whose value is a default one (therefore not influenced by $\mathcal{S}(\mathtt{P})$). This value lets the user know that an output action has been denied for security reasons.

On line 8, the program tries to output something while the current context of execution has been influenced by $\mathcal{S}(\mathtt{P})$. Hence, if the output occurs, the sequence

	Program	Automaton:			Actions
	P	input	output	new state	executed
1	x := l + 3;	x := l + 3	OK	$(\{h\}, \epsilon)$	x := l + 3
2	**if** (x > 10) **then**	branch x > 10	ACK	$(\{h\}, \bot)$	
3	y := h;	y := h	OK	$(\{h,y\}, \bot)$	y := h
4	**output** x;	**output** x	OK	$(\{h,y\}, \bot)$	**output** x
5	**output** y;	**output** y	**output** θ	$(\{h,y\}, \bot)$	**output** θ
6	**if** (h) **then**	branch h	ACK	$(\{h,y\}, \bot\top)$	
7	z := 0;	z := 0	OK	$(\{h,y,z\}, \bot\top)$	z := 0
8	**output** x	**output** x	NO	$(\{h,y,z\}, \bot\top)$	
9	**else** x := 1	not x := 1	ACK	$(\{h,y,z,x\}, \bot\top)$	
10	**end**	exit	ACK	$(\{h,y,z,x\}, \bot)$	
11	**else skip**	not **skip**	ACK	$(\{h,y,z,x\}, \bot)$	
12	**end**	exit	ACK	$(\{h,y,z,x\}, \epsilon)$	

Table 1. Example of the automaton evolution during an execution.

generated by the execution is influenced by some secret values. Therefore the automaton denies any output; it does not even give another action to execute in place of the current one. The semantics does as if the action was "**skip**".

4 Properties of the Monitoring Mechanism

A first theorem states soundness: any monitored execution is a non-interfering execution. A second one states "pseudo-completeness": the monitor does not alter observable behavior of a non-trivial set of non-interfering executions. Complete proofs of these theorems can be found in the companion technical report [13].

Soundness. The soundness property is based on the notion of non-interference between the secret inputs and the output sequence of an execution. An execution is considered *safe* — knowing that the program's source is public — if and only if it does not convey the variety in its secret inputs to the sequence output, that is, if the secret inputs have no influence on the execution's outputs.

For all programs P, with set of secret inputs $\mathcal{S}(\mathtt{P})$, and value store σ, let $[\![\mathtt{P}]\!]\sigma$ be the output sequence obtained via the monitored execution of P in the initial state $((\mathcal{S}(\mathtt{P}), \epsilon), \sigma)$. This definition is formally stated as follows:

$$[\![\mathtt{P}]\!]\sigma = o \text{ if and only if } \exists\ q', \sigma' :\ ((\mathcal{S}(\mathtt{P}), \epsilon), \sigma) \Vdash \mathtt{P} \stackrel{o}{\Longrightarrow} (q', \sigma')$$

The following theorem states that every monitored execution is *safe*, i.e. it is non-interfering. Let $\stackrel{X}{=}$ be an equivalence relation between value stores. Let $\sigma_1 \stackrel{X}{=} \sigma_2$ assert that σ_1 and σ_2 are indistinguishable for X, i.e., σ_1 and σ_2 associate the same value to every variable in X. Let X^c be the complement of set X in $\mathbb{X}$.

Theorem 1 (Soundness: monitored executions are non-interfering). *For all programs P, whose set of secret inputs is $\mathcal{S}(P)$, and value stores σ_1 and σ_2,*

$$\sigma_1 \stackrel{\mathcal{S}(P)^c}{=} \sigma_2 \ \Rightarrow\ [\![P]\!]\sigma_1 = [\![P]\!]\sigma_2$$

Proof (sketch). The proof — which can be found in [13] — goes by induction on the derivation tree of $[\![\mathtt{P}]\!]\sigma_1$. It relies on the fact that, after any "step" in the evaluation of $[\![\mathtt{P}]\!]\sigma_1$ and "equivalent step" in the evaluation of $[\![\mathtt{P}]\!]\sigma_2$, the automaton states of both executions are equal and the value stores are indistinguishable for the complement of the first element of the automaton states.

Pseudo-completeness. Thus any terminating monitored execution is non-interfering. However, to achieve this goal, the monitor sometimes modifies the output sequence of the execution. The sequence of outputs resulting from the execution of program P with initial state σ might differ according to the semantics used, the standard one (Fig. 2) or the monitoring semantics (Fig. 3). The sequel gives a lower bound on the set of non-interfering executions on which the monitoring mechanism has no impact. It shows that the mechanism proposed in this paper preserves the output sequence of any execution of a program which is well-typed under a security type system similar to the one of Volpano et al. [11].

Figure 4 shows the security type system. It is the same one as that of Volpano et al. [11] except for a small modification of the typing environment and the addition of a rule for output statements (which are not in the language of [11]). The typing environment, γ, prescribes types for identifiers and is extended to handle expressions. $\gamma(e)$ is the type of the expression e in the typing environment γ. The lattice of types used has only two elements and is defined using the reflexive relation $\leq$ ($L \leq H$). L is the type for public data and H the type for secrets. A program P is well-typed if it can be typed under a typing environment γ in which every secret input is typed secret (i.e. $\forall x \in \mathcal{S}(\mathtt{P}), \gamma(x) = H$).

$$\frac{\gamma(e) = \tau' \quad \tau' \leq \tau}{\gamma \vdash e : \tau} \qquad \frac{\gamma(x) = \tau' \quad \gamma \vdash e : \tau' \quad \tau \leq \tau'}{\gamma \vdash x := e : \tau \text{ cmd}} \qquad \frac{\tau \leq H}{\gamma \vdash \textbf{skip} : \tau \text{ cmd}}$$

$$\frac{\gamma \vdash e : L}{\gamma \vdash \textbf{output}\ e : L \text{ cmd}} \qquad \frac{\gamma \vdash S_1 : \tau \text{ cmd} \quad \gamma \vdash S_2 : \tau \text{ cmd}}{\gamma \vdash S_1 ; S_2 : \tau \text{ cmd}}$$

$$\frac{\gamma \vdash e : \tau' \quad \gamma \vdash S_1 : \tau' \text{ cmd} \quad \gamma \vdash S_2 : \tau' \text{ cmd} \quad \tau \leq \tau'}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end} : \tau \text{ cmd}} \qquad \frac{\gamma \vdash e : \tau' \quad \gamma \vdash S : \tau' \text{ cmd} \quad \tau \leq \tau'}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} : \tau \text{ cmd}}$$

Figure 4. The type system used for comparison

The problem of non-interference is neither dynamically nor statically decidable. Consequently, the monitoring mechanism proposed in this paper is not complete or transparent. However, Theorem 2 states that the monitoring mechanism does not alter executions of well-typed programs. To show that the inclusion is strict, consider the following program: $x := h;\ x := 0;$ **output** x. h is the only secret input. Every execution is non-interfering. But as the type system is flow insensitive, this program is ill-typed. However, the monitoring mechanism does not interfere with the outputs of this program while still guaranteeing that any monitored execution is non-interfering.

Theorem 2 (Monitoring preserves *type-safe* programs).
For all programs $\mathtt{P}$ *with secret inputs* $\mathcal{S}(\mathtt{P})$*, typing environments* γ *with variables belonging to* $\mathcal{S}(\mathtt{P})$ *typed secret, types* τ*, and value stores* σ *and* σ'*,*

$$\left.\begin{array}{l}\gamma \vdash \mathtt{P} : \tau \ cmd \\ \sigma \vdash \mathtt{P} \stackrel{o}{\Longrightarrow} \sigma'\end{array}\right\} \Rightarrow [\![\mathtt{P}]\!]\sigma = o$$

Proof (sketch). The proof goes by induction on the derivation tree of the unmonitored evaluation of P. It relies on the fact that the unmonitored evaluation of any well-typed command is matched by an equivalent monitored evaluation.

5 Conclusion

This paper addresses the security problem of confidentiality from the point of view of non-interference. It presents a monitoring mechanism enforcing non-interference of any execution. This monitoring mechanism is based on a semantics that communicates with a security automaton. During the execution, the semantics generates automaton inputs abstracting the events occurring. The automaton tracks the flows of information between the secret inputs and the current value of the program's variables. It also validates the execution of atomic actions (mainly outputs) to ensure confidentiality of the secret inputs.

Because the monitoring mechanism enforces non-interference, it significantly differs from standard monitors [14,20]. Usually, monitors are only aware of statements which *are* really executed. With the mechanism proposed, when exiting a conditional, the branch which has *not* been executed is analyzed. This takes into account implicit indirect flows between the test of a conditional and those variables whose values would be modified by the execution of the branch which is not executed. As noticed — but not elaborated on — by Vachharajani et al. [17, Sect. 4.2.2], this feature is required in order to enforce non-interference.

Section 4 shows that any monitored execution is non-interfering. Thus a user having access to the low outputs of a monitored execution is unable to deduce anything about the values of the secret inputs. In addition, the monitoring mechanism is proved not to alter executions of a program which is well-typed under a type system similar to the one of Volpano, Smith and Irvine [11].

Future work, which is under way, addresses the extension of the monitoring mechanism to a concurrent setting that includes a synchronization command. The goal is to achieve more precision than a type system equivalent to, e.g., the concurrent one due to Smith and Volpano [21].

Related work. Reference monitors is a widely studied area [22, 23]. The use of automata to monitor "good behaviors" led Schneider to formalize reference monitors as *truncation automata* that enforce safety properties [14]; and develop, with Erlingsson, a monitoring tool called *SASI* [24]. Among other works on the subject, Hamlen et al. worked on an extension to the .NET Common Intermediate Language called *Mobile* [25]. Their extension supports a type system that certifies in-lined reference monitors. In a successful attempt to increase the power of

monitors, Ligatti et al. [19] introduced monitors, based on *edit automata*, able to modify the sequence of actions executed and to enforce *infinite renewal properties* [26]. Such properties include every safety property, some liveness properties and some properties that are neither. Because such monitors are limited to the enforcement of *trace* properties, it is not immediately obvious that they can handle non-interference (which is not a *trace* property).

The vast majority of research on non-interference concerns static analyses and involves type systems [3, 27]. Some "real size" languages together with security type system have been developed (for example, JFlow/JIF [5] and FlowCaml [6]).

Information flow monitoring is not as popular as static analyses for information flow, but there has been interesting research. For example, RIFLE [17] is a complete runtime information flow security system; which however lacks formal analysis and proofs. The majority of those works, including RIFLE, does not take into consideration flows created by un-executed commands. It has been shown [28] that this feature can be used to gain information about secrets in some cases. The only exception known by the authors – in the domain of information flow monitoring – is the work by Masri et al. [29] which presents a dynamic information flow analysis for structured or unstructured languages. However, their work does not study deeply the dynamic correction of "bad" flows and lacks formal statements and proofs of the correctness of the correction mechanism. The solution proposed is to stop the execution as soon as a potential flow from a secret data to a public *sink* is detected. In some cases, this can create a new covert channel revealing secret information — see, e.g., [28].

Benefits of monitoring compared to static analyses. Monitoring an execution has a cost. So, what are the main benefits of non-interference monitoring compared to static analyses? The first concerns the possibility that a monitoring mechanism can be used to change the security policy for each execution. In the majority of cases, running a static analysis before every execution would be more costly than using a monitor. The second reason is that non-interference is a rather strong property. Many programs are rejected by static analyses of non-interference. In such cases it is still possible to use a monitoring mechanism with the possibility that some executions will be altered by the monitoring mechanism. However behavior alteration is an intrinsic feature of any monitoring mechanism. Monitoring non-interference ensures confidentiality while still allowing testing with regard to other specifications using unmonitored executions as perfect oracle — at least as perfect as the original program.

There are two main reasons why it is interesting to use a non-interference monitor on a program rejected by a static analysis. The first one lies in the granularity of the non-interference property. Static analyses have to take into consideration all possible executions of the program analyzed. This implies that if a single execution is unsafe then the program (thus all its executions) is rejected. Whereas, even if some executions of a program are unsafe, a monitor still allows this program to be used. The unsafe executions, which are not useful, are altered to respect the desired property while the safe executions are still usable.

The second one is that a monitoring mechanism may be more precise than static analyses because during execution the monitoring mechanism gets some accurate information about the "path behavior" of the program. As an example, let us consider the following program where h is the only secret input and l the only other input (a public one).

```
if ( test1(l) ) then tmp := h else skip end;
if ( test2(l) ) then x := tmp else skip end;
output x
```

Without information on *test1* and *test2* (and often, even with), a static analysis would conclude that this program is unsafe because the secret input information could be carried to x through *tmp* and then to the output. However, if *test1* and *test2* are such that no value of l makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitoring mechanism would allow any execution of this program. The reason is that, l being a public input, only executions following the same path as the current execution are taken care of by the monitoring mechanism. So, for such configurations where the branching conditions are not influenced by the secret inputs, a monitoring mechanism is at least as precise as any static analysis — and often more precise.

Acknowledgments. The authors are grateful to the reviewers for their remarks; and to Jay Ligatti, David Naumann and Andrei Sabelfeld for insightful and helpful comments on an earlier version of this paper.

References

1. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: Proc. Symp. Security and Privacy. (1982) 11–20
2. Cohen, E.S.: Information Transmission in Computational Systems. ACM SIGOPS Operating Systems Review **11**(5) (1977) 133–139
3. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE J. Selected Areas in Communications **21**(1) (January 2003) 5–19
4. Banerjee, A., Naumann, D.A.: Stack-based Access Control and Secure Information Flow. Journal of Functional Programming **15**(2) (2005) 131–177
5. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Proc. ACM Symp. Principles of Programming Languages. (1999) 228–241
6. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. on Programming Languages and Systems **25**(1) (2003) 117–158
7. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A Core calculus of Dependency. In: Proc. ACM Symp. Principles of Programming Languages. (1999) 147–160
8. Barthe, G., Serpette, B.: Partial evaluation and non-interference for object calculi. In: Proc. FLOPS. Volume 1722 of LNCS., Springer-Verlag (November 1999) 53–67
9. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. Higher Order and Symbolic Computation **14**(1) (March 2001) 59–91
10. Mizuno, M., Schmidt, D.: A Security Flow Control Algorithm and Its Denotational Semantics Correctness Proof. J. Formal Aspects of Comp. **4**(6A) (1992) 727–754

11. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. J. Computer Security **4**(3) (1996) 167–187
12. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. **28**(1) (2006) 175–205
13. Le Guernic, G., Banerjee, A., Schmidt, D.: Automaton-based Non-interference Monitoring. Technical Report 2006-1, Kansas State University, Manhattan, KS, USA (April 2006) http://www.cis.ksu.edu/~schmidt/techreport/2006.list.html.
14. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1) (2000) 30–50
15. McLean, J.: A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions. In: Proc. Symp. Security and Privacy. (1994) 79–93
16. Ashby, W.R.: An Introduction to Cybernetics. Chapman & Hall, London (1956) ISBN 0416683002.
17. Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., August, D.I.: RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In: Proc. Symp. Microarchitecture. (2004)
18. Kahn, G.: Natural Semantics. In: Proc. Symp. on Theoretical Aspects of Computer Science. Volume 247 of LNCS., Springer-Verlag (1987) 22–39
19. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Sec **4**(1-2) (2005) 2–16
20. Viswanathan, M.: Foundations for the Run-time Analysis of Software Systems. PhD thesis, University of Pennsylvania (December 2000)
21. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In: Proc. ACM Symp. on Principles of Programming Languages. (January 1998) 355–364
22. Schneider, F.B., Morrisett, G., Harper, R.: A Language-Based Approach to Security. In: Informatics—10 Years Back, 10 Years Ahead. Volume 2000 of Lecture Notes in Computer Science. Springer-Verlag (2000) 86–101
23. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Department of Computer Science, Cornell University (2003)
24. Erlingsson, Ú., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: Proc. New Security Paradigms Workshop, ACM Press (1999) 87–95
25. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified In-lined Reference Monitoring on .NET. In: ACM Workshop on Programming Languages and Analysis for Security. (2006)
26. Ligatti, J., Bauer, L., Walker, D.: Enforcing Non-safety Security Policies with Program Monitors. In: ESORICS. (2005) 355–373
27. Pottier, F., Conchon, S.: Information flow inference for free. In: Proc. ACM International Conf. on Functional Programming. (2000) 46–57
28. Le Guernic, G., Jensen, T.: Monitoring Information Flow. In Sabelfeld, A., ed.: Proceedings of the Workshop on Foundations of Computer Security, DePaul University (June 2005) 19–30
29. Masri, W., Podgurski, A., Leon, D.: Detecting and Debugging Insecure Information Flows. In: Symp. on Software Reliability Engineering. (2004) 198–209