Automaton-based Confidentiality Monitoring of Concurrent Programs

Gurvan Le Guernic

Automaton-based Confidentiality Monitoring of Concurrent Programs

Gurvan Le Guernic [†]

IRISA - Campus universitaire de Beaulieu, 35042 Rennes - France
Kansas State University - Manhattan, KS 66506 - USA

E-mail: Gurvan.Le_Guernic@irisa.fr

Abstract

Noninterference is typically used as a baseline security policy to formalize confidentiality of secret information manipulated by a program. In contrast to static checking of noninterference, this paper considers dynamic, automaton-based, monitoring of information flow for a single execution of a concurrent 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. All monitored executions are proved to be noninterfering (soundness) and executions of programs that are well-typed in a security type system similar to the one of Smith and Volpano [23] are proved to be unaltered by the monitor (partial transparency).

1. Introduction

The proposed monitor for concurrent programs deals with confidentiality, more precisely with *noninterference in concurrent programs*. This notion has first been introduced in [8] as the absence of strong dependency [6]. A program is said to be noninterfering if the values of its public (low) outputs do not depend on the values of its private (high or secret) inputs. Static analyses for noninterference [1–3, 14, 15, 17, 21, 25] have been studied extensively and are well surveyed in [20].

The specificity of the approach lies in its decision unit (execution and not program), its run-time nature and the monitoring mechanism used (a security automaton). Similar approaches lack formal proofs for the majority of them [13, 24] and miss concurrency [11, 13, 22, 24] (even if some claim they do). Dealing with concurrency is really tricky as shown by the examples of Fig. 2 on synchronization and Fig. 3 on shared variables in Sect. 2.3. The monitor for *concurrent programs* introduced in this paper is supported by *formal proofs* and guarantees *confidentiality* of secret data: either the monitor deduces that the current execution is noninterfering or it alters the behavior of the execution to obtain a noninterfering execution.

There are three main benefits to this dynamic approach. First, its decision unit — the fact that the analysis states properties of executions and not programs — allows the monitor to safely run *noninterfering executions* of *unreliable programs*. Then, its run-time nature makes it a "lazy" polyvariant analysis. The monitor deals with any input partition between private and public ones without requiring prior analysis of any possible combinations. It can even easily deal with varying partition. Finally, a monitor follows the precise control flow of a program and thus calculation of control dependencies (as might be performed in static analyses) can be more accurate. Section 7 contains an example illustrating this improved accuracy. A distinguishing feature, 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 unexecuted commands.

The next section starts by presenting the syntax and semantics of the studied language which includes a *synchronization command* similar to the one used in [16]. It also introduces the principles used by the monitoring mechanism. Section 3 defines formally the *security automaton* which is at the heart of the monitoring mechanism. The following section characterizes the monitoring semantics which links the monitoring automaton previously presented with the standard semantics given in Sect. 2.1. Then, comes an example in Sect. 5 which examines the evolution of the monitoring automaton during a sample execution. Before concluding in Sect. 7, the monitor *soundness* and *partial transparency* are proved in Sect. 6.

[†]The author was partially supported by NSF grants CCR-0209205 and CCR-0296182.

2. Outline

A concurrent program is a pool of threads (Θ). Before execution, each thread contains a sequential program. Such a pool is formally defined as a partial function from integers to sequential programs: $\Theta(i)$ is the i^{th} sequential program of the pool. The grammar of sequential programs is given below. In order to gain simplicity while describing the semantics, the grammar is split into four different blocks. In this grammar, $\langle ident \rangle$ stands for a variable name (or identifier). As in any programming language, variables have an associated value which can be updated by an assignment. In the concurrent setting, each variable has also an associated unique lock. Variable locks can be acquired and then released by threads. Whenever a thread has acquired a lock but not released it yet, this thread is said to *own* this lock. A given lock can be owned by at most one thread at any time.

$$
\begin{array}{rcl}
\langle action \rangle & ::= & \langle ident \rangle \textbf{ := } \langle expr \rangle \;\mid\; \textbf{output } \langle expr \rangle \;\mid\; \textbf{skip} \\
\langle control \rangle & ::= & \textbf{if } \langle expr \rangle \textbf{ then } \langle prog \rangle \textbf{ else } \langle prog \rangle \textbf{ end} \\
& \mid & \textbf{while } \langle expr \rangle \textbf{ do } \langle prog \rangle \textbf{ done} \\
& \mid & \textbf{with } \langle idSet \rangle \textbf{ when } \langle expr \rangle \textbf{ do } \langle prog \rangle \textbf{ done} \\
\langle com \rangle & ::= & \langle action \rangle \;\mid\; \langle control \rangle \\
\langle prog \rangle & ::= & \langle prog \rangle \,;\, \langle prog \rangle \;\mid\; \langle com \rangle
\end{array}
$$

A sequential program ($\langle prog \rangle$) is either a sequence of sequential programs or a command ($\langle com \rangle$). Actions ($\langle action \rangle$) and control commands ($\langle control \rangle$) are the only two types of commands. An action is either an assignment of the value of an expression ($\langle expr \rangle$) to a variable ($\langle ident \rangle$), an output of the value of an expression, or a skip statement. A control command is either: a conditional executing one program (out of two) depending on the value of an expression, a loop executing a program as long as the value of a given expression remains `true`, or a synchronization command (**with** $\langle idSet \rangle$ **when** $\langle expr \rangle$ **do** $\langle prog \rangle$ **done**). This command is executed — and therefore is replaced by the program it encompasses — only if no other thread owns one of the locks of the given set of variables ($\langle idSet \rangle$) and the value of the expression ($\langle expr \rangle$) is `true`. Otherwise, the thread executing this synchronization command is blocked — i.e. it can not execute anything as long as the conditions are not fulfilled.

The syntax of the synchronization command comes from [4, 9, 16]. It has been chosen for its ability to encode easily lots of different synchronization constructions. It allows the monitoring mechanism to deal with only one synchronization construction while keeping the language expressive with regard to synchronization.

2.1. Standard semantics

During the execution, a thread does not necessarily contain a sequential program ($\langle prog \rangle$) as defined above. In fact, a thread under execution contains an *execution statement* ($\langle execStat \rangle$), which is either empty ($\emptyset$), a usual sequential program ($\langle prog \rangle$), an *execution statement* followed by a sequential program ($\langle execStat \rangle \,;\, \langle prog \rangle$), or a *locked statement* ($\odot \langle ident\text{-}set \rangle [\langle execStat \rangle]$) carrying additional information about the locks owned by this particular thread. The grammar of *execution statements*, which is based on the grammar of sequential programs ($\langle prog \rangle$), is given below.

$$
\begin{array}{rcl}
\langle execStat \rangle & ::= & \langle prog \rangle \;\mid\; \langle execStat \rangle \,;\, \langle prog \rangle \\
& \mid & \emptyset \;\mid\; \odot \langle idSet \rangle [\langle execStat \rangle]
\end{array}
$$

A *locked statement* ($\odot \bar{x}[P]$) is obtained after the execution of a synchronization command (**with** $\bar{x}$ **when** e **do** P **done**). When a thread $\Theta(i)$ executes a synchronization command, it acquires the locks of the variables given in parameter ($\bar{x}$). This information is registered in the program state to forbid other threads from acquiring those locks as long as the thread $\Theta(i)$ has not released them. However, the thread $\Theta(i)$ is still allowed to execute synchronization commands on those locks — i.e. acquire again the same locks. The information needed to allow a thread to acquire again the same lock is not registered in the program state. The locks a given thread is allowed to acquire again are registered in its sequential program itself by the substitution of the synchronization command just evaluated by a locked statement. This locked statement contains the variables whose locks have just been acquired and the program to be executed while holding those locks.

Let study one execution step of the following thread.

```
1  with x when true do
2    with x when b do skip done
3  done;
4  with y when true do skip done
```

If the lock of variable x is not owned by any other thread then one execution step of the previous thread can take place and yields the following program.

```
1  ⊙x[with x when b do skip done];
2  with y when true do skip done
```

As the thread studied is now the owner of x's lock, it will be allowed to acquire again this lock at its next execution step.

Execution statement semantics. The semantics of concurrent programs is based on the semantics of *execution statements* which is described in Fig. 1. This latter semantics is based on rules written in the format: $\langle\!| \varsigma \vdash S |\!\rangle \xrightarrow{o}_{\text{s}} \langle\!| \varsigma' \vdash S' |\!\rangle$. The rules mean that in the execution state ς statement S evaluates to statement S' yielding state ς' and output sequence o. Let $\mathbb{X}$ be the domain of variable identifiers and $\mathbb{D}$ be the semantic domain of values. An execution state ς is a pair (σ, λ) composed of a value store σ ($\mathbb{X} \to \mathbb{D}$)

and a lock set λ ($2^{\mathbb{X}}$). σ maps variable identifiers to their respective current value. The definition of value stores is extended to expressions, so that $\sigma(e)$ is the value of the expression e in a program state whose value store is σ. λ is a set of variable identifiers. It corresponds to the set of variables whose lock is currently owned by a thread. An output sequence is a word in $\mathbb{D}^\star$. It is either an empty sequence (written ϵ), a single value (for example, $\sigma(e)$) or the concatenation of two other sequences (written $o_1\ o_2$).

$$\langle\!|\, (\sigma,\lambda) \vdash x := e \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma[x \mapsto \sigma(e)],\lambda) \vdash \emptyset \,|\!\rangle$$

$$\langle\!|\, \varsigma \vdash \textbf{output}\ e \,|\!\rangle \xrightarrow{\sigma(e)}_{\mathrm{s}} \langle\!|\, \varsigma \vdash \emptyset \,|\!\rangle$$

$$\langle\!|\, \varsigma \vdash \textbf{skip} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, \varsigma \vdash \emptyset \,|\!\rangle$$

$$\frac{\sigma(e) = v}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{if}\ e\ \textbf{then}\ S^{\mathtt{true}}\ \textbf{else}\ S^{\mathtt{false}}\ \textbf{end} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma,\lambda) \vdash S^v \,|\!\rangle}$$

$$\frac{\sigma(e) = \mathtt{true}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{while}\ e\ \textbf{do}\ S^{\mathrm{l}}\ \textbf{done} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma,\lambda) \vdash S^{\mathrm{l}}\ ;\ \textbf{while}\ e\ \textbf{do}\ S^{\mathrm{l}}\ \textbf{done} \,|\!\rangle}$$

$$\frac{\sigma(e) = \mathtt{false}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{while}\ e\ \textbf{do}\ S^{\mathrm{l}}\ \textbf{done} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma,\lambda) \vdash \emptyset \,|\!\rangle}$$

$$\frac{\bar{x} \cap \lambda = \emptyset \qquad \sigma(e) = \mathtt{true}}{\langle\!|\, (\sigma,\lambda) \vdash \textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S^{\mathrm{s}}\ \textbf{done} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma,\lambda \cup \bar{x}) \vdash \odot\bar{x}[S^{\mathrm{s}}] \,|\!\rangle}$$

$$\frac{}{\langle\!|\, \varsigma \vdash \emptyset\ ;\ S^{\mathrm{t}} \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, \varsigma \vdash S^{\mathrm{t}} \,|\!\rangle}$$

$$\frac{\langle\!|\, \varsigma \vdash S^{\mathrm{h}} \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, \varsigma' \vdash S^{\mathrm{h}'} \,|\!\rangle}{\langle\!|\, \varsigma \vdash S^{\mathrm{h}}\ ;\ S^{\mathrm{t}} \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, \varsigma' \vdash S^{\mathrm{h}'}\ ;\ S^{\mathrm{t}} \,|\!\rangle}$$

$$\frac{}{\langle\!|\, (\sigma,\lambda) \vdash \odot\bar{x}[\emptyset] \,|\!\rangle \xrightarrow{\epsilon}_{\mathrm{s}} \langle\!|\, (\sigma,\lambda \setminus \bar{x}) \vdash \emptyset \,|\!\rangle}$$

$$\frac{\langle\!|\, (\sigma,\lambda \setminus \bar{x}) \vdash S^{\mathrm{s}} \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, (\sigma',\lambda') \vdash S^{\mathrm{s}'} \,|\!\rangle}{\langle\!|\, (\sigma,\lambda) \vdash \odot\bar{x}[S^{\mathrm{s}}] \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, (\sigma',\lambda' \cup \bar{x}) \vdash \odot\bar{x}[S^{\mathrm{s}'}] \,|\!\rangle}$$

Figure 1. Standard semantics

As shown in Fig. 1, if the prerequisites allow it, the execution of a synchronization command yields a *locked statement*, $\odot\bar{x}[\mathtt{P}]$, where $\bar{x}$ is a set of variable identifiers. This rule states that if the conditions hold, a synchronization command is replaced by the program P it includes. The current thread will now own the locks of the variables belonging to $\bar{x}$.

Concurrent program semantics. A concurrent program under execution is a pool of execution statements. Taking one execution step of a concurrent program means taking one execution step for one of the execution statements of the thread pool. There is no constraint on which thread has to be evaluated. The scheduler used for the thread interleaving is a nondeterministic one. After each step of execution, the next thread to execute is nondeterministically selected among those which can take an execution step. Such a scheduler avoids timing covert channels which are based on the hypothesis that the attacker is able to guess accurately in which order the scheduler will execute the different threads. When dealing with schedulers which are deterministic, solutions proposed in [18] should be applicable.

The semantics of unmonitored executions of concurrent programs (pool of threads) is given by the following rule:

$$\frac{\imath \in \mathrm{dom}(\Theta) \qquad \langle\!|\, \varsigma \vdash \Theta(\imath) \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, \varsigma' \vdash S'_\imath \,|\!\rangle}{\langle\!|\, \varsigma \vdash \Theta \,|\!\rangle \xrightarrow{o}_{\mathrm{s}} \langle\!|\, \varsigma' \vdash \Theta[\imath \mapsto S'_\imath] \,|\!\rangle}$$

2.2. Monitoring principles

The mechanism developed in this paper is a monitor aiming at enforcing the confidentiality of private data manipulated by any concurrent program. In order to achieve this goal, the monitoring mechanism is designed to enforce a stronger property based on the notion of noninterference. The informal definition of this notion [8] states that a program is "safe" if its private inputs have no influence on the observable behavior of the program. This definition is stated at the level of the program (at the level of all its executions as a whole). On the other hand, a monitor acts at the level of one execution and not of all executions of a program. It is then necessary to refine the notion of noninterference in order for it to be applicable to a single execution.

Terminology used to describe flows in this paper differs slightly from the terminology used in the literature on static analyses. In this paper, the flow from the right part of an assignment (the expression) to the left part (the variable) is called a *direct flow*. The flow from the test of a conditional to a variable, whose value is modified by an assignment in one branch of the conditional, is called an *indirect flow* — an *explicit indirect flow* if the assignment is executed and an *implicit indirect flow* if the assignment is not executed.

The remainder of the current section starts by giving some definitions which are then used to define formally the notion of noninterference for executions. Subsequently, it provides a short description of the way the monitor works.

Finally, by commenting upon two examples of concurrent programs, it illustrates some of the difficulties of monitoring noninterference and the solutions adopted.

What is a noninterfering concurrent execution? In order to prove noninterference soundness in Sect. 6.1 and state formally the effect of the monitor, it is required first to define precisely "noninterference" in our concurrent setting. A program $\mathtt{P}$ is said to be noninterfering if and only if its private inputs have no influence on the observable behavior of the program. This is usually characterized as follows: any two executions of $\mathtt{P}$ started with the same public inputs — but potentially different private inputs — have the exact same observable behavior. It is then required to compare the observable behavior of different executions. In this paper, the observable behavior is the output sequence generated. The output sequence generated by the evaluation of a given sequential program started in a given state is unique. However, the output sequence generated by the evaluation of a given concurrent program — a pool of sequential programs — started in a given state is not. For multi-threaded programs, the output sequence generated by the execution of a given program started in a given state depends on the thread interleaving which occurred during the evaluation. Therefore, the definition of the observable behavior of an execution designated by a program and an initial state must take into account all such possible output sequences. Additionally, as it is also desired to take into account non-terminating executions, our definition of observable behavior must also take into consideration the observable behavior of an execution at any step of its evaluation, not only the final one. Definition 2.1 states that the observable behavior — output sequences — of the execution of a given concurrent program started in a given state is the set of all prefixes of any output sequence which can be obtained by any thread interleaving.

This paper defines two small-step semantics: a standard semantics ($\rightarrow_{\mathrm{S}}$) in Fig. 1, and a monitoring semantics ($\rightarrow_{\mathrm{M}}$) in Fig. 5. Let $\rightarrow_s$ denote any of those semantics. For all small-step semantics ($\rightarrow_s$), $\xrightarrow{o}_s$ represent one execution step yielding the output sequence o; and $\xrightarrow{o}_s^\star$ represent any number of execution steps, which together yield the output sequence o. $\xrightarrow{o_1}_s^\star$ is the reflexive and transitive closure of $\xrightarrow{o_2}_s$. Let X denote a program state, either from the standard semantics or the monitoring semantics.

Definition 2.1 (Observable behavior: $\mathcal{O}[\![\Theta]\!]_s X$)**.**
For all small-step semantics $\rightarrow_s$, concurrent programs Θ, and program states X, the observable behavior of the execution with the semantics $\rightarrow_s$ of the concurrent program Θ started in the initial state X, written $\mathcal{O}[\![\Theta]\!]_s X$, is the set:

$$\{\, o \mid \exists\, X', \Theta' \,:\, \langle\!|\, X \vdash \Theta \,|\!\rangle \xrightarrow{o}_s^\star \langle\!|\, X' \vdash \Theta' \,|\!\rangle \,\}$$

The monitoring mechanism does not determine if a program is or is not noninterfering, but if a precise execution is or is not noninterfering. It has then to determine if, by looking at the current output sequence, it is possible to differentiate the current execution from executions of the same program started with the same public inputs but potentially different private inputs. This is possible only if the output sequence, the observable behavior, of the current execution — which follows a precise thread interleaving — of the program $\mathtt{P}$ can not be produced, for any thread interleaving, from another initial state X_2 having the same public inputs; in other words, only if there exists an initial state X_2 such that the output sequence of the current execution does not belong to $\mathcal{O}[\![\Theta]\!]_s X_2$. Definition 2.3 characterizes a noninterfering execution as an execution whose output sequence belongs to the observable behavior — which is independent from the thread interleaving — of any execution of the same program started in an initial state low equivalent (Definition 2.2) to the initial state of the current execution. In the following definitions, by convention, in any initial program state, no locks are owned by any thread.

Definition 2.2 (Low Equivalent Initial States)**.**
Two *initial* states X_1, respectively X_2, containing the value stores σ_1, respectively σ_2, are *low equivalent* with regards to a set of variables V, written $X_1 \overset{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2:

$$X_1 \overset{V}{=} X_2 \quad \Longleftrightarrow \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 2.3 (Noninterfering Execution)**.**
Let V^{c} be the complement of V in the set $\mathbb{X}$. For all small-step semantics $\rightarrow_s$, concurrent programs Θ, program states X_1 and output sequences o, an execution with the semantics $\rightarrow_s$ of the initialized program (Θ, X_1) generating the output sequence o is noninterfering, written $ni(\Theta, s, X_1, o)$, if and only if, for any program state X_2:

$$X_1 \overset{\mathcal{S}(\Theta)^{\mathrm{c}}}{=} X_2 \;\Rightarrow\; o \in \mathcal{O}[\![\Theta]\!]_s X_2$$

How does the monitor enforce noninterference? As in [11], the monitoring mechanism is separated into two parts. The first element, described in Sect. 4, is a special semantics which delegates the main job to a security automaton described in Sect. 3. The purpose of the special semantics is to select and abstract the important events, with regard to noninterference, which occur during the execution. The abstractions of those events are then sent to the security automaton. For each input received, the automaton sends back to the semantics an output. Depending on those outputs received, the special semantics modifies the normal execution of the program. The security automaton is in charge of keeping track of the variables whose value carries *variety*

— i.e. their value is influenced by the values of the private inputs — and keeping track of *variety* in the context of execution — i.e. the program counter — of each thread independently.

With regard to synchronization commands, the solution adopted follows standard solutions found in the literature [19]. Synchronizations inside the branch of a conditional whose branching condition is influenced by the private inputs are forbidden. However, it is impossible to stop the execution when encountering a synchronization command inside a conditional whose branching condition carries variety. Doing so would create a new covert channel leaking information to low level users if a public output was supposed to occur later in the execution. One solution would be to ignore synchronization commands in a context carrying variety. The drawback of this solution is to suppress synchronizations that the programmer deemed wise to include. The solution adopted in this monitor is to force any synchronization to occur outside any conditional whose branching expression carries variety. Whenever the monitoring mechanism has to evaluate such a conditional, it executes the locking part of any synchronization command appearing in any branch of the conditional.

2.3. The monitor by the example

Synchronization commands are interference prone. Fig. 2 contains the code of a concurrent program. It is composed of two threads, has a single private input (h) and uses an internal variable v which is never output. The first thread (Fig. 2(a)) takes the lock of the variable v and then outputs "a" followed by "b" before releasing the lock. The second thread (Fig. 2(b)) outputs "c"; then, if the private input h is `true`, it tries to acquire the lock of v and then releases it immediately; finally, it outputs "d". What can be deduced

```
1 with v
2 when true do
3   output "a";
4   v := v + 1;
5   output "b"
6 done
```

(a) Thread $\Theta(1)$

```
1 output "c";
2 if h then
3   with v when true do
4     v := v + 2;
5   done
6 else skip end;
7 output "d"
```

(b) Thread $\Theta(2)$

Figure 2. Leakage due to synchronization

by a low level user if it sees the output sequence "a c d b"? From the code of thread $\Theta(1)$, it is possible to deduce that the first thread owns the lock of v at least from the time the program outputs "a" until it outputs "b". Consequently, as "c" and "d" are output between "a" and "b", thread $\Theta(2)$ has evaluated every command between the two outputs of "c" and "d" without needing to acquire the lock of v. It is then easy, by looking at the code of $\Theta(2)$, to deduce that the private input h is `false`. The "bad" flow from h to the output sequence is due to the synchronization commands on v.

To prevent the "bad" flow presented above, it is not a good idea to simply stop the execution whenever a synchronization command has to be execute inside a conditional whose condition carries variety. In the example presented, the synchronization command in $\Theta(2)$ is *not* executed. Stopping the program each time the program has to evaluate a conditional whose condition carries variety and which contains a synchronization command is not appealing either. And, it does not seem wise to ignore any synchronization command inside a conditional whose condition carries variety. There may be a good reason for having one synchronization command in the previous example, avoiding a concurrent write access to the variable v. The solution adopted by the monitoring mechanism presented in this paper consists in acquiring all the locks of any synchronization command appearing in any branch of a conditional whose condition carries variety before evaluating such a conditional. This mechanism prevents the program from outputting "a c d b" even if h is `false`.

Be cagey with newsmongers. Fig. 3(a) contains the code of a Very Important Program (VIP) which has a single private input (h) and is run concurrently with the program in Fig. 3(b). This latter program is a newsmonger which outputs indefinitely and as fast as it can some of the variables manipulated by the VIP (x and y). The VIP only sets x to 0, then y to 0 and finally resets both to 1. However, depending on the value of its private input, it resets first x or y to 1. The interference comes from the newsmonger. If the news-

```
1 x := 0; y :=0;
2 if h then
3   x := 1; y := 1
4 else
5   y := 1; x := 1
6 end;
```

(a) VIP

```
1 while true do
2   output x;
3   output y;
4 done
```

(b) Newsmonger

Figure 3. Leakage due to concurrent access

monger is lucky enough, and the VIP unlucky enough, the scheduler will let the newsmonger take at least two steps, so outputting x and y, while the VIP is in the middle of resetting those variables to 1. It is then easy, depending on which one of x and y has been reset to 1 first, to deduce the value of the private input h. The newsmonger is able to steal

the value of the private input of an unmonitored execution of the VIP. Less intuitively, it is also possible, if the scheduler gives a high priority to the newsmonger, to deduce the value of *h* if the execution is supervised by a monitor which takes into account indirect flows created by an assignment only when this assignment is evaluated. The reason is that, in between the two assignments resetting *x* and *y* to 1, those two variables do not have the same security level. Consequently, such a "bad" monitor does not act the same way for the commands "**output** x" and "**output** y" if only one of *x* and *y* has been reset to 1.

The monitoring mechanism proposed in this paper takes into account an indirect flow as soon as the conditional which is the source of this flow is executed. For any execution of the VIP, this means that *x* and *y* get a high security level at the same time — when the conditional branching on *h* is first evaluated. Therefore, the monitor has the same behavior for the two commands "**output** x" and "**output** y", to output a default value instead of the value of *x* or *y*. The newsmonger is then unable to steal the VIP's secret.

Section 5 follows precisely the behavior of the monitoring mechanism on an example. Before that, the next two sections give a formal definition of the monitoring mechanism: first of the security automaton used, and then of the special semantics communicating with this automaton.

3. The monitoring automaton

This section describes the automaton used in the monitoring mechanism. It is in charge of tracking the information flows and controlling the execution in order to enforce noninterference. The semantics described in Sect. 4 sends abstractions of the execution events to this automaton. In turn, the automaton sends back directions to the semantics to control the execution, thus *enforcing* noninterference.

Definitions coming in the remaining of the document use extensively the following notations. For any set $\mathbb{S}$, let $2^{\mathbb{S}}$ be the power set of $\mathbb{S}$. For any set $\mathbb{A}$, let $\mathbb{A}^\star$ be the set of all strings over the alphabet $\mathbb{A}$. And finally, for any program $\mathtt{P}$ whose variables belongs to $\mathbb{X}$, let the set of private input variables be $\mathcal{S}(\mathtt{P})$ ($\mathcal{S}(\mathtt{P}) \subseteq \mathbb{X}$).

General definition. The preceding sections presented the monitoring mechanism as if it is using a single monitoring automaton. However, the monitoring automaton for a concurrent program is defined by first defining monitoring automata for sequential programs. A concurrent program is a pool of threads, each of which contains a single sequential program. Each time the "automaton" of the concurrent program Θ receives an input ϕ, generated by an evaluation step in a sequential program $\Theta(i)$, a state for the monitoring automaton of $\Theta(i)$ is extracted from the state of the monitoring automaton of Θ. Then, the monitoring automaton for sequential programs takes a transition on input ϕ generating output ψ. Finally, the state of the monitoring automaton of Θ is updated using the new state returned by the transition of the monitoring automaton for sequential programs and the output ψ is sent back to the monitoring semantics.

For any program $\mathtt{P}$, let $\mathbb{X}$ be the set of variables of $\mathtt{P}$. The automaton enforcing noninterference for the sequential program $\mathtt{P}$ is the tuple $\mathcal{A}(\mathtt{P}) = (Q, \Phi, \Psi, \delta, q_0)$ where:

- Q is a set of states;
$$Q = 2^{\mathbb{X}} \times (\mathbb{X} \to \mathbb{N}) \times 2^{\mathbb{X}} \times \{\top, \bot\}^\star$$
- Φ is the input alphabet, constituted of abstractions of a subset of program events, specified below;
- Ψ is the output alphabet, constituted of execution controlling commands, specified below;
- δ is a partial transition function;
$$\delta : (Q \times \Phi) \longrightarrow (\Psi \times Q)$$
- q_0, an element of Q, is the start state.
$$q_0 = (\mathcal{S}(\mathtt{P}),\ \emptyset,\ \emptyset,\ \epsilon)$$

The automaton states. An automaton state is a quadruple (V, W, L, w) composed of two sets (V and L) of variables belonging to $\mathbb{X}$, a multiset (W) of variables belonging to $\mathbb{X}$ and a word (w) belonging to a language whose alphabet is $\{\top, \bot\}$. At any step of the execution, V contains all the variables which *may* carry variety — i.e. whose values *may* have been influenced by the initial values of the secret input variables ($\mathcal{S}(\mathtt{P})$). L contains the variables whose lock *may* be owned by a thread at an "equivalent" step of an execution of the same program started with the same public inputs but potentially different private inputs. L is the part of the automaton state which protects secret data against attacks similar to the one described in the example about synchronization commands on the preceding page. W contains at least once each variable which is assigned in a branch of a still active conditional whose condition carries variety. For example, at any step of the execution of line 3 or 5 of the program in Figure 3(a), W contains *x* and *y*. W is the part of the automaton state which protects secret data against attacks similar to the one described in the newsmonger's example on the previous page. w, called a *branching context*, tracks variety in the context of the execution with regard to previous branching commands. The context consists in the level of *variety* the conditions of the previous, but still active, branching commands. If w contains $\top$, then the statement executed was initially a branch of a conditional whose test may have been influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Hence this statement may not be executed with a different choice of initial values for the private input variables ($\mathcal{S}(\mathtt{P})$).

Automaton inputs. The input alphabet of the automaton (Φ) contains abstractions of some events that can occur during an execution. The alphabet Φ consists of the following:

"$\texttt{branch}(\lambda, e, P^{\mathbf{e}}, P^{\mathbf{u}})$**"** is generated each time a conditional is evaluated. λ is the set of variables whose lock is currently owned by a thread, e is the expression whose value determines the branch which is executed, P^{e} is the branch which will be executed, and P^{u} is the branch which will not be executed. For example, before evaluation of "**if** $x > 10$ **then** S_1 **else** S_2 **end**", the input "$\texttt{branch}(\lambda, x > 10, S_1, S_2)$" is sent to the monitoring automaton if x is greater than 10.

"$\texttt{merge}(P^{\mathbf{e}}, P^{\mathbf{u}})$**"** is generated each time a conditional has been fully evaluated. P^{e} is the branch which has been executed, and P^{u} is the branch which has not been executed. For example, after evaluation of "**if** $x > 10$ **then** S_1 **else** S_2 **end**", the input "$\texttt{merge}(S_1, S_2)$" is sent to the monitoring automaton if $x > 10$.

"$\texttt{sync}(\bar{x}, e)$**"** is generated before any synchronization command. For example, **with** x, b **when** b **do** S^{s} **done** generates the input "$\texttt{sync}(\{x, b\}, b)$".

any atomic action of the language (assignment, skip or output statement) which has to be evaluated is first sent to the automaton for validation before its execution.

Automaton outputs. The automaton outputs are sent back from the automaton to the monitoring semantics in order to control the execution. The output alphabet (Ψ) is composed of the following:

"OK**"** is used whenever the monitoring automaton allows an execution step that it could have altered or denied.

"NO**"** is used whenever the monitoring automaton forbids the execution step. This step is simply skipped by the monitored execution.

any atomic action of the language. This is the answer of the monitoring automaton whenever another action than the current one has to be executed.

Automaton transitions. Figure 4 specifies the transition function of the automaton. A transition rule 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 $\mathit{FV}(e)$ be the set of variables occurring in e. For example, $\mathit{FV}(x+y)$ returns the set $\{x, y\}$. Let $\mathit{defines}(S)$ be the set of all variables which may be defined by an execution of S. This function is used in order to take into account the implicit indirect flows created when a conditional whose condition carries variety is evaluated. A formal definition of this function follows:

$$
\begin{aligned}
&\mathit{defines}(x := e) = \{x\}\\
&\mathit{defines}(\textbf{output}\ e) = \mathit{defines}(\textbf{skip}) = \emptyset\\
&\mathit{defines}(S\ ;\ S') = \mathit{defines}(S) \cup \mathit{defines}(S')\\
&\mathit{defines}(\textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S'\ \textbf{end}) = \mathit{defines}(S) \cup \mathit{defines}(S')\\
&\mathit{defines}(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathit{defines}(S)\\
&\mathit{defines}(\textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathit{defines}(S)
\end{aligned}
$$

Let $\mathit{needs}(S)$ be the set of all variables whose lock may be required in order to execute S. This function is used by the monitoring automaton in order to "virtually" acquire all the locks which may be needed for the complete evaluation of a conditional whose condition carries variety. Those locks are not acquired by the current thread. They will be only when their corresponding synchronization command will be executed. However, the monitoring automaton registers them in its state. Before allowing the evaluation of a conditional whose condition carries variety, the automaton checks that there is no needed lock which is already registered in the automaton state for an other thread. This is done in order to make sure that the evaluation of the branch designated by the condition — which carries variety — can not be stopped because of a needed lock which would be owned by an other thread. A formal definition of $\mathit{needs}(S)$ follows:

$$
\begin{aligned}
&\mathit{needs}(x := e) = \mathit{needs}(\textbf{output}\ e) = \mathit{needs}(\textbf{skip}) = \emptyset\\
&\mathit{needs}(S\ ;\ S') = \mathit{needs}(S) \cup \mathit{needs}(S')\\
&\mathit{needs}(\textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S'\ \textbf{end}) = \mathit{needs}(S) \cup \mathit{needs}(S')\\
&\mathit{needs}(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathit{needs}(S)\\
&\mathit{needs}(\textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S\ \textbf{done}) = \bar{x} \cup \mathit{needs}(S)
\end{aligned}
$$

Finally, let $\mathit{stops}(S)$ be true if the execution of S *may* be "stopped": either by looping or waiting on a synchronization command for the value of an expression to become `true`or `false`. This function is used by the automaton when finishing the complete evaluation of any conditional whose condition carries variety. This is done in order to prevent an attacker to learn some secret information by observing from the source code that, under some conditions, one of the branches of a conditional, whose condition carries variety, can not terminate. This function does not take into consideration the locks needed by a synchronization command. The reason being that the security automaton ensures that no secret information can flow through the lock state of shared variables. This is dealt by the set L in the

automaton state. $\mathit{stops}(S)$ is formally defined as follows:

$$\mathit{stops}(x \texttt{ := } e) = \mathit{stops}(\textbf{output } e) = \mathit{stops}(\textbf{skip}) = \mathit{false}$$
$$\mathit{stops}(S \ ; \ S') = \mathit{stops}(S) \vee \mathit{stops}(S')$$
$$\mathit{stops}(\textbf{if } e \textbf{ then } S \textbf{ else } S' \textbf{ end}) = \mathit{stops}(S) \vee \mathit{stops}(S')$$
$$\mathit{stops}(\textbf{while } e \textbf{ do } S \textbf{ done}) \equiv e \neq \texttt{false}$$
$$\mathit{stops}(\textbf{with } \bar{x} \textbf{ when } e \textbf{ do } S \textbf{ done}) \equiv e \neq \texttt{true}$$

Figure 4 shows that the automaton forbids (NO) or edits (**output** θ) only executions of output statements. For other inputs, it either allows the evaluation of the statement — a transition exists for the current input in the current state — or forces the monitoring semantics to take an execution step for another thread — there is no transition for the current input in the current state. Whenever a transition occurs, the automaton tracks, in the set V, the variables that may contain secret information — have *variety*. Additionally, it registers, in W, the variables whose variety may evolve differently due to a conditional whose condition carries variety; it tracks, in L, the variables whose lock status may carry variety; and it tracks, in w, the *variety* of the branching conditions.

Inputs "$\texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})$" are generated at entry points of conditionals. On reception of such inputs in state (V, W, L, w), if the conditional to be evaluated belongs to a branch of a conditional whose condition carries variety (w does not belong to $\{\bot\}^\star$) then the automaton simply pushes $\bot$ to the end of w. Otherwise, the automaton checks if the branching condition (e) carries variety. To do so, it computes the intersection of the variables appearing in e with the set V. If the intersection is empty, then the condition of the branching statement does not carry variety. In that case, the automaton simply pushes $\bot$ to the end of w. If the intersection is not empty, then the value of e *may be* influenced by the initial values of $\mathcal{S}(\texttt{P})$. If this is the case then the automaton checks if the execution can proceed without being stopped by a lock that it can not acquire. To do so, it verifies that there is no needed lock which is already owned by another thread or has already been booked previously for the execution of a conditional of an other thread ($\tilde{l} \cap (\lambda \cup L) = \emptyset$). If all needed locks are available, the automaton pushes $\top$ to the end of w, registers the needed locks in the new automaton state and deals with indirect flows. This is done by adding all variables which may be assigned to in one of the branches of the conditional into the set of variables potentially carrying variety (V) and into the multiset of variables whose order of assignment may carry variety (W).

Whenever an execution exits a branch — which is a member of a conditional c — the input "$\texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})$" is sent to the automaton. On reception of such input, the automaton checks if other values of private inputs may have induced the execution of the conditional to be stopped. This

$$((V, W, L, w),\ \texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W, L, w\bot)$$
$$\text{iff } \begin{cases} & \mathit{FV}(e) \cap V = \emptyset \\ \vee & w \notin \{\bot\}^\star \end{cases}$$

$$((V, W, L, w),\ \texttt{branch}(\lambda, e, P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V \cup \tilde{v}, W \uplus \tilde{v}, L \cup \tilde{l}, w\top)$$
$$\text{with } \begin{cases} \tilde{v} = \mathit{defines}(P^{\mathrm{e}}) \cup \mathit{defines}(P^{\mathrm{u}}) \\ \tilde{l} = \mathit{needs}(P^{\mathrm{e}}) \cup \mathit{needs}(P^{\mathrm{u}}) \end{cases}$$
$$\text{iff } \begin{cases} & \mathit{FV}(e) \cap V \neq \emptyset \\ \wedge & w \in \{\bot\}^\star \\ \wedge & \tilde{l} \cap (\lambda \cup L) = \emptyset \end{cases}$$

$$((V, W, L, w\bot),\ \texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W, L, w)$$

$$((V, W, L, w\top),\ \texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} (V, W \setminus \tilde{v}, L \setminus \tilde{l}, w)$$
$$\text{with } \begin{cases} \tilde{v} = \mathit{defines}(P^{\mathrm{e}}) \cup \mathit{defines}(P^{\mathrm{u}}) \\ \tilde{l} = \mathit{needs}(P^{\mathrm{e}}) \cup \mathit{needs}(P^{\mathrm{u}}) \end{cases}$$
$$\text{iff } \begin{cases} & \neg\mathit{stops}(P^{\mathrm{e}}) \\ \wedge & \neg\mathit{stops}(P^{\mathrm{u}}) \end{cases}$$

$$((V, W, L, w),\ \texttt{sync}(\bar{x}, e)) \xrightarrow{OK} (V, W, L, w)$$
$$\text{iff } \begin{cases} & \mathit{FV}(e) \cap V = \emptyset \\ \wedge & (\quad w \notin \{\bot\}^\star \\ & \vee \quad \bar{x} \cap L = \emptyset \quad) \end{cases}$$

$$(q,\ \textbf{skip}) \xrightarrow{OK} q$$

$$((V, W, L, w),\ x \texttt{ := } e) \xrightarrow{OK} (V \cup \{x\}, W, L, w)$$
$$\text{iff } \begin{cases} & \mathit{FV}(e) \cap V \neq \emptyset \\ \vee & x \in W \end{cases}$$

$$((V, W, L, w),\ x \texttt{ := } e) \xrightarrow{OK} (V \setminus \{x\}, W, L, w)$$
$$\text{iff } \begin{cases} & \mathit{FV}(e) \cap V = \emptyset \\ \wedge & x \notin W \end{cases}$$

$$((V, W, L, w),\ \textbf{output } e) \xrightarrow{OK} (V, W, L, w)$$
$$\text{iff } w \in \{\bot\}^\star \text{ and } \mathit{FV}(e) \cap V = \emptyset$$

$$((V, W, L, w),\ \textbf{output } e) \xrightarrow{\textbf{output } \delta} (V, W, L, w)$$
$$\text{iff } w \in \{\bot\}^\star \text{ and } \mathit{FV}(e) \cap V \neq \emptyset$$

$$((V, W, L, w),\ \textbf{output } e) \xrightarrow{NO} (V, W, L, w)$$
$$\text{iff } w \notin \{\bot\}^\star$$

Figure 4. Transition function

is the case if the condition of c carries variety — w ends with $\top$ — and the function *stops*() concludes that at least one of the branches of c can get stuck. If that is not the case, the automaton allows the evaluation step and the last letter of w is removed. Additionally, if c is the first conditional whose condition carries variety in this sequential program, the variables added into the multiset W for protection while executing c are removed from the multiset.

Before executing any synchronization command, an input of type "$\texttt{sync}(\bar{x}, e)$" is sent. The automaton allows the evaluation step only if the expression in the synchronization command does not carry variety and either the needed locks are not reserved ($\bar{x} \cap L = \emptyset$) or the automaton has already booked those locks for the current thread.

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

On reception of an input "x **:=** e" the automaton deals only with direct flows; indirect flows are already taken care for by the processing of inputs of type "$\texttt{branch}(\lambda, e, P^{\mathtt{e}}, P^{\mathtt{u}})$". Hence, on input x **:=** e, the automaton only checks if the value of e carries variety. This is the case only if $FV(e)$ and V are not disjoint. If the value of e carries variety, then x (the variable modified) is added to V: $V' = V \cup \{x\}$. Otherwise 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. There is an exception to this rule. If there exists an indirect flow to x from the condition carrying variety of a still active conditional then the x must not be removed from the set of variables carrying variety. If such an indirect flow exists then the assigned variable is still under the protection acquired when the corresponding "$\texttt{branch}(\lambda, e, P^{\mathtt{e}}, P^{\mathtt{u}})$" input has been processed ($x \in W$). In that case, the variable is left in the set V. Other parts of the automaton state do not change.

The rules for the automata input, "**output** e," prevent bad flows through two different channels. The first channel 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, the constant δ. 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 generation of an output by 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.

The automaton states for concurrent programs. Monitoring automaton states for concurrent programs are similar to the automata states for sequential programs. An automaton state is a quadruple $\mathtt{Q} = (V, W, L, \overline{w})$ which belongs to the following set:

$$2^{\mathbb{X}} \times (\mathbb{X} \to \mathbb{N}) \times 2^{\mathbb{X}} \times (\mathbb{N} \to \{\top, \bot\}^\star)$$

V, W and L are directly inherited from automata states for sequential programs. $\overline{w}$ is a function mapping thread identifiers to their respective branching context (as defined for sequential programs).

The functions used to extract, respectively update, the automaton state for a given thread from, respectively into, the automaton state of the concurrent program are defined below.

$$\mathit{extract}((V, W, L, \overline{w}), \imath) = (V, W, L, \overline{w}(\imath))$$
$$\mathit{update}((V, W, L, \overline{w}), \imath, (V', W', L', w')) = (V', W', L', \overline{w}[\imath \mapsto w'])$$

The current section defines formally the monitoring automata. Among other things, it defines an input alphabet and an output alphabet which are used to communicate with the monitoring semantics. This semantics, described in the next section, is in charge of the concrete evaluation of the concurrent program under the supervision of the monitoring automaton.

4. The monitoring semantics

During a monitored execution, a thread contains a *monitored statement* ($\langle \mathit{monitStat} \rangle$) which is hihgly similar to the *execution statements* described in Fig. 1. The only difference is the addition of *branched statements* ($\otimes(\langle \mathit{prog} \rangle, \langle \mathit{prog} \rangle)[\langle \mathit{monitStat} \rangle]$). The statement $\otimes(P^e, P^u)[S]$ states that statement S is a partial execution of P^e — i.e., S is the result of the application of some execution steps to P^e — and that P^e is the executed branch of a conditional whose unexecuted branch is P^u. The rule for if-statements of Fig. 5 gives a good intuition of the meaning of this statement. The grammar of *monitored statements* follows. It is based on the grammar of sequential programs given on page 2.

$$\begin{array}{rcl} \langle \mathit{monitStat} \rangle & ::= & \langle \mathit{prog} \rangle \mid \langle \mathit{monitStat} \rangle \,;\, \langle \mathit{prog} \rangle \\ & \mid & \emptyset \mid \odot \langle \mathit{idSet} \rangle [\langle \mathit{monitStat} \rangle] \\ & \mid & \otimes(\langle \mathit{prog} \rangle, \langle \mathit{prog} \rangle)[\langle \mathit{monitStat} \rangle] \end{array}$$

Thread semantics. The monitoring semantics of concurrent programs is based on the semantics of *monitored statements* which is described in Fig. 5. In this figure, A and A' denote atomic actions (skip statement, assignment or output statement). The monitoring semantics is based on rules written in the format: $\langle\!| \zeta \vdash S |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| \zeta' \vdash S' |\!\rangle$. It reads

as follows: in *monitored execution state* ζ, *monitored statement* S evaluates to S' yielding state ζ' and output sequence o. A monitored execution state ζ is a pair (ς, q) composed of an execution state (ς), of the standard semantics, and a monitoring automaton state (q), as defined in Sect. 3.

The semantics of *monitored statements* interacts with the security automaton using automaton transitions written $(q, \phi) \xrightarrow{\psi} q'$. It also uses the semantics of *execution statements* — the standard semantics — for the evaluation of actions (assignments, outputs, or skip statements). Both semantics can be distinguished by the letter appearing on the bottom right of the arrow (M for the monitoring semantics and nothing for the standard semantics).

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. 1) 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. 4, 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 "$\texttt{branch}(\lambda, e, P^e, P^u)$" where λ is the set of variables whose lock is currently owned by a thread, e is the test of the conditional, P^e is the "branch" which will be executed and P^u is the "branch" which will not be executed. After the evaluation step, the monitored execution state is updated with the automaton state returned by the monitoring automaton and the conditional evaluated is replaced by the branch to execute.

"**with** $\bar{x}$ **when** e **do** P^{s} **done**", a synchronization command, can be evaluated only if the needed locks ($\bar{x}$) are not currently owned by another thread, the condition e is true, and the monitoring automaton allows the evaluation on reception of the input "$\texttt{sync}(\bar{x}, e)$". After the evaluation step, the needed locks are added to the set of locks currently owned by a thread, the automaton state is updated, and the command is replaced by a *locked statement* indicating that the locks $\bar{x}$ are owned by the current thread for the evaluation of the program P^{s}.

Evaluating a *branched statement* ($\otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}}]$) is equivalent to evaluating the enclosed statement S^{b} as long as this statement is not completely evaluated. If the evaluation of S^{b} is completed then the input "$\texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})$" is sent to the automaton and the evaluation of the thread can proceed only if the automaton allows it.

In order to take one evaluation step of a *locked statement* ($\odot\bar{x}[S^{\mathrm{s}}]$), the set of locks $\bar{x}$ is temporarily removed from the set of owned locks in the execution state. Then, one evalua-

$$\frac{(q,\ A) \xrightarrow{OK} q' \qquad \langle\!| \varsigma \vdash A |\!\rangle \xrightarrow{o} \langle\!| \varsigma' \vdash \emptyset |\!\rangle}{\langle\!| (\varsigma, q) \vdash A |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| (\varsigma', q') \vdash \emptyset |\!\rangle}$$

$$\frac{(q,\ A) \xrightarrow{A'} q' \qquad \langle\!| \varsigma \vdash A' |\!\rangle \xrightarrow{o} \langle\!| \varsigma' \vdash \emptyset |\!\rangle}{\langle\!| (\varsigma, q) \vdash A |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| (\varsigma', q') \vdash \emptyset |\!\rangle}$$

$$\frac{(q,\ A) \xrightarrow{NO} q'}{\langle\!| (\varsigma, q) \vdash A |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| (\varsigma, q') \vdash \emptyset |\!\rangle}$$

$$\frac{\sigma(e) = v \qquad (q,\ \texttt{branch}(\lambda, e, P^v, P^{\neg v})) \xrightarrow{OK} q'}{\langle\!| (\varsigma, q) \vdash \textbf{if } e \textbf{ then } P^{\texttt{true}} \textbf{ else } P^{\texttt{false}} \textbf{ end} |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| (\varsigma, q') \vdash \otimes(P^v, P^{\neg v})[P^v] |\!\rangle}$$

$$\frac{\begin{array}{c} P^{\texttt{true}} = P^{\mathrm{l}}\ ;\ \textbf{while } e \textbf{ do } P^{\mathrm{l}} \textbf{ done} \qquad P^{\texttt{false}} = \emptyset \\ \sigma(e) = v \qquad (q,\ \texttt{branch}(\lambda, e, P^v, P^{\neg v})) \xrightarrow{OK} q' \end{array}}{\langle\!| (\varsigma, q) \vdash \textbf{while } e \textbf{ do } P^{\mathrm{l}} \textbf{ done} |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| (\varsigma, q') \vdash \otimes(P^v, P^{\neg v})[P^v] |\!\rangle}$$

$$\frac{\begin{array}{c} \bar{x} \cap \lambda = \emptyset \qquad \sigma(e) = \texttt{true} \\ (q,\ \texttt{sync}(\bar{x}, e)) \xrightarrow{OK} q' \end{array}}{\langle\!| ((\sigma, \lambda), q) \vdash \textbf{with } \bar{x} \textbf{ when } e \textbf{ do } P^{\mathrm{s}} \textbf{ done} |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| ((\sigma, \lambda \cup \bar{x}), q') \vdash \odot\bar{x}[P^{\mathrm{s}}] |\!\rangle}$$

$$\frac{}{\langle\!| \zeta \vdash \emptyset\ ;\ S^{\mathrm{t}} |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| \zeta \vdash S^{\mathrm{t}} |\!\rangle}$$

$$\frac{\langle\!| \zeta \vdash S^{\mathrm{h}} |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| \zeta' \vdash S^{\mathrm{h}'} |\!\rangle}{\langle\!| \zeta \vdash S^{\mathrm{h}}\ ;\ S^{\mathrm{t}} |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| \zeta' \vdash S^{\mathrm{h}'}\ ;\ S^{\mathrm{t}} |\!\rangle}$$

$$\frac{(q,\ \texttt{merge}(P^{\mathrm{e}}, P^{\mathrm{u}})) \xrightarrow{OK} q'}{\langle\!| (\varsigma, q) \vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[\emptyset] |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| (\varsigma, q') \vdash \emptyset |\!\rangle}$$

$$\frac{\langle\!| \zeta \vdash S^{\mathrm{b}} |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| \zeta' \vdash S^{\mathrm{b}'} |\!\rangle}{\langle\!| \zeta \vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}}] |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| \zeta' \vdash \otimes(P^{\mathrm{e}}, P^{\mathrm{u}})[S^{\mathrm{b}'}] |\!\rangle}$$

$$\frac{}{\langle\!| ((\sigma, \lambda), q) \vdash \odot\bar{x}[\emptyset] |\!\rangle \xrightarrow{\epsilon}_{\mathrm{M}} \langle\!| ((\sigma, \lambda \setminus \bar{x}), q') \vdash \emptyset |\!\rangle}$$

$$\frac{\langle\!| ((\sigma, \lambda \setminus \bar{x}), q) \vdash S^{\mathrm{s}} |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| ((\sigma', \lambda'), q') \vdash S^{\mathrm{s}'} |\!\rangle}{\langle\!| ((\sigma, \lambda), q) \vdash \odot\bar{x}[S^{\mathrm{s}}] |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| ((\sigma', \lambda' \cup \bar{x}), q') \vdash \odot\bar{x}[S^{\mathrm{s}'}] |\!\rangle}$$

Figure 5. Monitoring semantics

tion step is taken for S^{s} and the locks $\bar{x}$ are put back in the set of owned locks in the execution state if the evaluation of S^{s} is not completed.

Concurrent program semantics. As explained in Sect. 3, the automaton states for monitoring the execution of a single thread are different from the states for monitoring concurrent programs. The monitoring semantics uses two functions, described in Sect. 3, for converting between automaton states for thread and automaton states for concurrent programs. The semantics of monitored executions of concurrent programs is given by the following rule:

$$\frac{\imath \in \mathrm{dom}(\Theta) \quad \langle\!| (\varsigma, \mathit{extract}(\mathtt{Q}, \imath)) \vdash \Theta(\imath) |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| (\varsigma', q) \vdash S'_\imath |\!\rangle}{\langle\!| (\varsigma, \mathtt{Q}) \vdash \Theta |\!\rangle \xrightarrow{o}_{\mathrm{M}} \langle\!| (\varsigma', \mathit{update}(\mathtt{Q}, \imath, q)) \vdash \Theta[\imath \mapsto S'_\imath] |\!\rangle}$$

Initial state of a monitored concurrent execution. There is a unique initial state for monitoring the execution of a given concurrent program with a given initial value store. Definition 4.1 states that the initial state of the monitored execution of a concurrent program is the monitoring execution state whose initial value store is the given one, whose set of owned lock is empty and whose automaton state designates the value of the private inputs as the only elements carrying variety.

Definition 4.1 (Initial State of Monitored Executions)**.** For all concurrent programs Θ and value stores σ, the initial state of the monitored execution of Θ with initial value store σ, written $\zeta^{\mathcal{I}}_{\Theta,\sigma}$, is:

$$(\ (\sigma,\ \emptyset),\quad (\mathcal{S}(\Theta),\ \emptyset,\ \emptyset,\ \{i \mapsto \epsilon \mid i \in \mathrm{dom}(\Theta)\})\ )$$

5. Example of monitored execution

Figure 6 is an example of a concurrent program having two threads. Fig. 6(a) contains the code of the sequential program of the first thread and Fig. 6(b) contains the code of the second thread. This program has only one private input (h) and no public input. It uses three internal variables (v, b and x). Both threads attempt to write in the variable v. Both assignments to v are protected by a synchronization on the lock of v. Additionally, before assigning a value to v, the second thread waits for b to be `true`. After assigning a value to v, the second thread outputs twice the value of x. The first thread, depending on the value of the private input h, either assigns a value to x and outputs "a", or assigns a value to v. The last command evaluated by the first thread resets the value of x to 0.

Table 1 shows the evolution of the monitoring automaton for an execution of the program of Fig. 6. This execution starts in an initial state where the private input h is `true`, b

```
1 if h then
2   x := 1;
3   output "a"
4 else
5   with v when true do
6     v := v - 1
7   done
8 end;
9 x := 0
```

(a) Thread $\Theta(1)$

```
1 with v
2 when b do
3   v := v + 1
4 done;
5 output x;
6 output x
```

(b) Thread $\Theta(2)$

Figure 6. Another concurrent program

is `true` and $x = v = 0$. As the program has only one private input (h) and two threads, the initial state of the monitoring automaton is the following one: $(\{h\}, \emptyset, \emptyset, [1 \mapsto \epsilon, 2 \mapsto \epsilon])$. The execution's steps are numbered on the left. The first column of the table shows the value of the program counters of the two threads. "$\imath \rhd j$" and "$\imath \blacktriangleright j$" indicates that the program counter of the $\imath^{\text{th}}$ thread maps to the line j. 0 is used for j when the execution of the thread is completed. $\blacktriangleright$ is used to designate the thread which will be evaluated at this execution step. The following column contains the event abstractions which are sent by the semantics to the automaton. In this column, P^t stands for the lines 2 to 3 of the first thread and P^f stands for the lines 5 to 7 of the first thread. Then comes the answer of the automaton telling the semantics what action to take. The 4th column gives the new state of the automaton following the transition triggered by the automaton input that is shown on the same line (the automaton state before the transition is the one of the preceding line). Finally, the last column lists the actions which are actually evaluated by the monitored execution of the program.

At the execution step 2, the only thread which can be executed is the second one. The reason is that at the preceding step, the second thread took the lock of v. The first command to be executed by the first thread is a conditional containing a synchronization command on v. Therefore, to execute the first thread, the monitor requires this thread to be able to acquire the lock of v. This is impossible as this lock is owned by the second thread. Hence, even if the branch to be executed is not the one containing the synchronization command, the program can not evaluate the conditional of the first thread. An example that justifies this rule has been given in Fig. 2.

Execution step 3 evaluates the conditional of the first thread. This conditional, whose condition carries variety, contains an assignment to x in one branch and an assignment to v in the other one. In order to prevent an attack similar to the one exposed in Fig. 3, the monitor considers

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

	Program counters	Automaton: input	output	new state	Actions executed
1	$1 \triangleright 1 \quad 2 \blacktriangleright 1$	$\texttt{sync}(\{v\}, b)$	OK	$(\{h\} \quad , \quad \emptyset \quad , \quad \emptyset \quad , \quad [1{>}\epsilon, 2{>}\epsilon])$	
2	$1 \triangleright 1 \quad 2 \blacktriangleright 2$	$v := v + 1$	OK	$(\{h\} \quad , \quad \emptyset \quad , \quad \emptyset \quad , \quad [1{>}\epsilon, 2{>}\epsilon])$	$v := v + 1$
3	$1 \blacktriangleright 1 \quad 2 \triangleright 4$	$\texttt{branch}(\emptyset, h, P^t, P^f)$	OK	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	
4	$1 \triangleright 2 \quad 2 \blacktriangleright 4$	**output** x	**output** θ	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	**output** θ
5	$1 \blacktriangleright 2 \quad 2 \triangleright 5$	$x := 1$	OK	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	$x := 1$
6	$1 \blacktriangleright 3 \quad 2 \triangleright 5$	**output** "a"	NO	$(\{h, x, v\}, \{x, v\}, \{v\}, [1{>}\top, 2{>}\epsilon])$	
7	$1 \blacktriangleright 8 \quad 2 \triangleright 5$	$\texttt{merge}(P^t, P^f)$	OK	$(\{h, x, v\}, \quad \emptyset \quad , \quad \emptyset \quad , \quad [1{>}\epsilon, 2{>}\epsilon])$	
8	$1 \blacktriangleright 9 \quad 2 \triangleright 5$	$x := 0$	OK	$(\{h, v\} \quad , \quad \emptyset \quad , \quad \emptyset \quad , \quad [1{>}\epsilon, 2{>}\epsilon])$	$x := 0$
9	$1 \triangleright 0 \quad 2 \blacktriangleright 5$	**output** x	OK	$(\{h, v\} \quad , \quad \emptyset \quad , \quad \emptyset \quad , \quad [1{>}\epsilon, 2{>}\epsilon])$	**output** x

those variables (x and v) as carrying variety straight away, even if there respective assignments have not been evaluated yet. This is done by adding the variables to the first element of the new automaton state. A clever newsmonger may try to dupe the monitor into believing that those variables do not carry variety anymore by resetting their value. To prevent it, the monitor also registers that those variables carry variety because of the processing of a conditional. This is done by adding them into the multiset which is the second element of the new state. Variables x and v will appear at least once in this multiset as long as the processing of the conditional which added them into it is not over. Furthermore, as stated in the explanation of step 2, the first thread acquires the lock of v because the conditional processed at this step is conditioned by an expression carrying variety and contains a synchronization command on v in one of its branches. Finally, the monitor registers the new context of execution — reflecting the fact that the branch that will be executed depend on an expression carrying variety — by pushing $\top$ into the branching context of the first thread in the new state.

In step 4 of the execution, the second thread attempts to output the value of x. This variable belongs to the first element of the automaton state before the transition. This means that the value of x may carry variety. Therefore, the monitor replaces the value of x by a default value. It allows the user to know that an output as been prevented for security reasons. Five steps later, at step 9, the second thread tries again to output the value of x. However, in the meanwhile, the first thread has reset the value of x to a new value. Doing so, it removed the variety in this variable. This is reflected by the fact that x does not belong anymore to the first element of the automaton state. Therefore, the monitoring automaton lets the output occur.

In step 6 of the execution, the first thread attempts to output a constant while still in one of the branch of a conditional whose condition carried variety. This is an unsafe behavior that the monitor forbids. Consequently, the execution simply skips this command.

This section shows an example for which the monitoring mechanism is able to ensure the confidentiality of private inputs. The next section proves this fact for any monitored execution.

6. Properties of the monitoring mechanism

After formally defining the monitoring mechanism, this section collates the monitor's characteristics to the standard properties of *soundness* with regards to noninterference and, specific to monitors, *transparency* with regards to the output sequence generated. A monitor is transparent with regard to a program behavior if, for any execution, this behavior is the same whether the execution is monitored or not. Section 6.1 proves the soundness of the monitoring mechanism with regard to the notion of noninterference. However, it is of course impossible for it to be complete, as the noninterference problem is undecidable. As well, it is impossible to have a sound and transparent monitor for noninterference, as with an interfering execution the monitor can be either transparent or sound but not both. However, Sect. 6.2 still proves that the monitor is transparent for a nontrivial set of executions.

6.1. Soundness

The monitoring mechanism is proved to be sound with regard to the noninterference property for executions. This means that, for any output sequences generated during the monitored execution of any program Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ and value stores σ' low equivalent to σ, there exists a thread interleaving such that during the monitored execution of Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma'}$ the same output sequence is generated. This property is formally stated by theorem 6.1.

Theorem 6.1 (Soundness)**.**
For all programs Θ and value stores σ, any monitored execution of Θ started in the initial state $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ is noninterfering. For any output sequence o, this is formally stated as follows:

$$o \in \mathcal{O}[\![\Theta]\!]_{M}\zeta^{\mathcal{I}}_{\Theta,\sigma} \;\Rightarrow\; ni(\Theta, {}_{M}, \zeta^{\mathcal{I}}_{\Theta,\sigma}, o)$$

Proof sketch. The proof — which can be found in [10] — goes by induction on the derivation tree of the current execution (e_c). First, it relies on the fact that, because of the monitoring mechanism, for any evaluation started with the same public inputs there exists an execution (e_2) having a thread interleaving "similar" to the one of the current execution. Additionally, it is demonstrated that, if a thread of e_2 follows a path in its sequential program different from the path followed by the equivalent thread in e_c, nothing will be output by those threads until they reach an "equivalent state". It is then possible to show that e_c and e_2 have the same output sequence. □

6.2. Partial transparency

A common property stated for monitors is *transparency*. A transparent monitor is one that does not alter the behavior of the monitored program. An interfering execution has, with regard to confidentiality, a faulty observable behavior. Therefore, a sound monitor enforcing noninterference has to alter the observable behavior of such an execution. Consequently, it is impossible for a sound monitor enforcing noninterference to also be transparent. However, for a precise set of programs, it is possible to prove that the monitoring mechanism presented in this paper achieves transparency — i.e. it does not alter the observable behavior of any execution of those programs. This set of programs is the set of all programs which are well-typed under a type system similar to the one of Smith and Volpano [23].

This type system is described in Figure 7. The language used in this paper includes two structures which do not appear in the language used in [23]. The two typing rules added for those structures are the only salient differences with the type system of Smith and Volpano [23]. 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 private data. 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. It is equal to the least upper bound of the types of the free variables appearing in e, or L if there is no free variable in e. It is formally defined as follows:

$$\gamma(e) = \bigsqcup_{x \in FV(e)} \gamma(x)$$

For any sequential program $\mathtt{P}$, if "$\gamma \vdash \mathtt{P} : \tau$ cmd" for some τ and γ in which every secret input is typed secret — i.e. $\forall x \in \mathcal{S}(\mathtt{P}), \gamma(x) = H$ — then $\mathtt{P}$ is said to be well-typed under the typing environment γ. For any concurrent program Θ, if all its threads contain a program well-typed under the same typing environment γ, then Θ is said to be well-typed under this typing environment γ. This is written "$\gamma \vdash \Theta$".

$$\frac{}{\gamma \vdash \textbf{skip} : \tau \text{ cmd}}$$

$$\frac{\gamma(e) \leq L}{\gamma \vdash \textbf{output}\ e : L \text{ cmd}}$$

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

$$\frac{\gamma \vdash S^{\text{h}} : \tau \text{ cmd} \qquad \gamma \vdash S^{\text{t}} : \tau \text{ cmd}}{\gamma \vdash S^{\text{h}}\ ;\ S^{\text{t}} : \tau \text{ cmd}}$$

$$\frac{\gamma(e) \leq \tau' \qquad \tau \leq \tau' \qquad \gamma \vdash S^{\text{true}} : \tau' \text{ cmd} \qquad \gamma \vdash S^{\text{false}} : \tau' \text{ cmd}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ S^{\text{true}}\ \textbf{else}\ S^{\text{false}}\ \textbf{end} : \tau \text{ cmd}}$$

$$\frac{\gamma(e) \leq L \qquad \gamma \vdash S^{\text{l}} : L \text{ cmd}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ S^{\text{l}}\ \textbf{done} : L \text{ cmd}}$$

$$\frac{\gamma(e) \leq L \qquad \gamma \vdash S^{\text{s}} : L \text{ cmd}}{\gamma \vdash \textbf{with}\ \bar{x}\ \textbf{when}\ e\ \textbf{do}\ S^{\text{s}}\ \textbf{done} : L \text{ cmd}}$$

Figure 7. Type system for confidentiality

Theorem 6.2 states that the monitoring mechanism does not alter executions of well-typed programs. In other words, the monitoring mechanism is transparent for any well-typed program. To understand that the monitor is transparent, and still sound, for a bigger set of programs, consider the concurrent program composed only of this sequential program: `x := h; x :=0; output x.` `h` is the only secret input. Every execution is noninterfering. 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 noninterfering.

Theorem 6.2 (Partial Transparency: monitoring preserves *type-safe* programs)**.**
For all programs Θ with secret inputs $\mathcal{S}(\Theta)$, typing environments γ with variables belonging to $\mathcal{S}(\Theta)$ typed secret, and value stores σ, if Θ is well-typed under γ then any mon-

itored execution of Θ *started in the initial state* $\zeta^{\mathcal{I}}_{\Theta,\sigma}$ *outputs a sequence which belongs to the observable behavior of the unmonitored execution of* Θ *started in the initial state* $(\sigma, \emptyset)$. *This is formally stated as follows:*

$$\gamma \vdash \Theta \quad \Rightarrow \quad \mathcal{O}[\![\Theta]\!]_M \zeta^{\mathcal{I}}_{\Theta,\sigma} \;=\; \mathcal{O}[\![\Theta]\!]_S(\sigma, \emptyset)$$

Proof sketch. The proof — which can be found in [10] — goes by induction on the derivation tree of the typing judgment. It shows that for any well-typed program, the additional constraints imposed by the monitoring mechanism for a "normal" execution are always true. Therefore, the observable behaviors of monitored or unmonitored executions of a well-typed program are the same. □

7. Conclusion

This paper addresses the matter of confidentiality in concurrent programs. This problem is formalized using the noninterference property which states that a program is safe if and only if its publicly observable behavior is not influenced by the values of its private inputs. The solution proposed consists in a monitoring mechanism enforcing noninterference on the fly. It is defined as a special semantics communicating with a security automaton. The role of the semantics is to send abstractions of the events occurring during the execution to the automaton. Then, it executes the program in accordance to the answers sent back by the automaton. This latter tracks the information flows and controls — allows, modifies or denies — the execution of output statements and synchronization commands. Section 6 not only proves that this monitoring mechanism is sound, in the sense that it enforces noninterference for any execution, but also that it is transparent for any program well-typed under a type system similar to the one of Smith and Volpano [23].

7.1. Related work

The vast majority of research on noninterference concerns static analyses and involves type systems [20]. Some "real size" languages together with security type system have been developed (for example, JFlow/JIF [15] and FlowCaml [17]).

Dynamic information flow analyses [5, 7, 26, 27] are not as popular as static analyses for information flow, but there has been interesting research. For example, RIFLE [24] is a complete runtime information flow security system based on an architectural framework and a binary translator. Masri et al. [13] present a dynamic information flow analysis for structured or unstructured languages. However, both works lack a final formal proof of noninterference. Both works propose to stop execution as soon as a "bad" flow is detected. This behavior creates a new covert channel that can reveal secret information — see, e.g., [12]. Moreover, when multi-threaded programs are handled by [13], an information leakage can arise in the example of Fig. 3. The reason is that, depending on the value of the branching condition, indirect flows are not taken into account in the same order. Shroff et al. [22] propose two dynamic information flow analyses for sequential programs which are supported by formal proofs. The first one is a purely dynamic analysis which increases its knowledge about indirect flows as the number of executions increase. This analysis will eventually detect all information flows after an undetermined number of executions. The second dynamic analysis is run with a prior knowledge of the indirect flows existing in any execution of the analyzed program. This knowledge is obtained by executing, at compile time, a static analysis computing a *fixed point of dependencies*. The proposed analyses are more precise than the majority of flow-insensitive static analyses. However, Shroff et al.'s analyses are not fully flow-sensitive, and indirect flows are handle in a "static" way which must reflect indirect flows in any possible executions. Therefore, the proposed dynamic analyses does not take a full advantage of their dynamic nature. For example, their analyses are unable to "safely" detect that any execution of the following program, where h is the only secret input, is noninterfering.

```
if h then x := 1 else skip end;
x := 0; output x
```

7.2. Specifics of the approach

To the author's knowledge, the solution proposed in this paper is the only dynamic analysis supported by formal proofs dealing with noninterference in a concurrent setting including synchronization commands. As the proposed mechanism deals with noninterference, which is not a property of an execution trace, it significantly differs from usual monitors. The dynamic analysis proposed is required to take into account the content of branches which are not executed. Additionally, due to the fact that synchronization is interference prone, it is also required to change the position of synchronization commands while still enforcing the thread interleaving constraints induced by the original position of those commands.

There are two main advantages of monitors over static analyses. First, 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. Moreover, a monitoring mechanism may be more precise than static analyses

because during execution the monitor 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 private input and l the only public input.

```
if ( f(l) ) then t := h else skip end;
if ( g(l) ) then x := t else skip end;
output x
```

Without information on f and g (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 t and then to the output. However, if f and g are such that no value of l makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitor 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 author is grateful to the reviewers for their pertinent remarks; and to Anindya Banerjee, Thomas Jensen and David Schmidt for insightful and helpful comments while developing the work presented in this paper.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[3] G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In *Proc. Functional and Logic Programming*, volume 1722 of *LNCS*, pages 53–67, Nov. 1999.

[4] S. D. Brookes. A semantics for concurrent separation logic. In *Proc. Concurrency Theory*, pages 16–34, 2004.

[5] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, Nov. 2001.

[6] E. S. Cohen. Information transmission in computational systems. *Operating Systems Review*, 11(5):133–139, 1977.

[7] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[8] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[9] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[10] G. Le Guernic. Automaton-based Non-interference Monitoring of Concurrent Programs. Technical Report 2007-1, Kansas State University, Manhattan, KS, USA, 2007. `http://www.cis.ksu.edu/~schmidt/techreport/2007.list.html`.

[11] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automaton-based confidentiality monitoring. In *Proc. Asian Computing Science*, LNCS, Dec. 2006. To appear.

[12] G. Le Guernic and T. Jensen. Monitoring Information Flow. In *Proc. Workshop on Foundations of Computer Security*, pages 19–30. DePaul University, June 2005.

[13] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proc. Software Reliability Engineering*, pages 198–209, 2004.

[14] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *J. Formal Aspects of Computing*, 4(6A):727–754, 1992.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. Principles of Programming Languages*, pages 228–241, Jan. 1999.

[16] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *Proc. Concurrency Theory*, pages 49–67, 2004.

[17] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.

[18] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. Asian Computing Science*, LNCS, Dec. 6-8 2006.

[19] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 227–241, July 2001.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[21] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

[22] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE Computer Society Press, July 2007. Draft version.

[23] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. Principles of Programming Languages*, pages 355–364, Jan. 1998.

[24] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. Microarchitecture*, 2004.

[25] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[26] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

[27] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. Security and Privacy*, pages 23–31, 1987.