Automaton-based Non-interference Monitoring

Gurvan Le Guernic, Anindya Banerjee, David Schmidt

Automaton-based Non-interference Monitoring [1]

Gurvan Le Guernic Anindya Banerjee David A. Schmidt

April 24, 2006

[1]Technical Report Nr. 2006-1. Department of Computing and Information Sciences, College of Engineering, Kansas State University (234 Nichols Hall, Manhattan, KS 66506, USA).

Contents

Abstract

This report presents a non-interference monitoring mechanism for sequential programs. Non-interference is a property of the information flows of a program. It implies the respect of the confidentiality of the secret information manipulated. The approach taken uses an automaton based monitor. During the execution, abstractions of the events occurring are sent to the automaton. The automaton uses those inputs to track the information flows and to control the execution by forbidding or editing dangerous actions. The mechanism proposed is proved to be sound and more efficient than a type system similar to the historical one developed by Volpano, Smith and Irvine.

1 Introduction

With the intensification of communication between information systems, the interest for researches on security has increased. Security is usually partitioned in three main domains:

confidentiality focuses on the control of the dissemination of information,

integrity is concerned by the incorruptibility of important information,

availability ensures the accessibility of resources to legal users.

This report deals with the concept of confidentiality; and more precisely with the notion of *non-interference* in sequential programs. This notion is based on ideas from classical information theory [Ash56]. It has first been introduced by Goguen and Meseguer [GM82] as the absence of *strong dependency* (a concept developed by Cohen [Coh77]).

> "information is transmitted from a source to a destination only when variety in the source can be conveyed to the destination" Cohen [Coh77, Sect.1].
>
> "One group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see." Goguen and Meseguer [GM82, Sect.1].

A sequential program is said to be *non-interfering* if the values of the public (or low) outputs do not depend on the values of the secret (or high) inputs. In other words, a program is non-interfering if the secret inputs do not interfere with the public outputs. Following the notation of Sabelfeld and Myers [SM03], the notion of non-interference (with regard to the equivalence relations $=_L$ and $\approx_L$) can be expressed as follows, with Σ denoting the set of all program states:

$$\forall \sigma_1, \sigma_2 \in \Sigma.\ \sigma_1 =_L \sigma_2 \Rightarrow [\![\mathtt{P}]\!]\sigma_1 \approx_L [\![\mathtt{P}]\!]\sigma_2 \tag{1}$$

This equation states that a program `P` is said to be *non-interfering* if and only if for any two states σ_1 and σ_2 that associate the same values to low (public) data (written $\sigma_1 =_L \sigma_2$), the executions of the program `P` in the initial states σ_1 and σ_2 are indistinguishable by an attacker having access only to the low (public) outputs. Those executions are termed *low-equivalent*; written $[\![\mathtt{P}]\!]\sigma_1 \approx_L [\![\mathtt{P}]\!]\sigma_2$. The low-equivalence relation characterizes the observational power of the attacker by stating what he can distinguish. This may vary from requiring the low (public) data of the final states to be equal for both executions, to requiring the two executions to have the same energy consumption.

As emphasized by the survey paper of Sabelfeld and Myers [SM03], there are already lots of works on non-interference. The particularity of the approach developed in this report lies in:

1. the fact that the proposed method analyzes executions and not programs,

2. the mechanism used in order to ensure the confidentiality of secret data.

The majority of previous research [BN05, Mye99a, PS03, ABHR99, BS99, MS01, SS01, Smi01], as the ones of Mizuno and Schmidt [MS92] and Volpano, Smith, and Irvine [VSI96], associate the notion of non-interference to the level of a program; they develop static analyses which *accept or reject programs* depending on the ability of *all its executions* to ensure the confidentiality of the secrets manipulated. The work presented here *accepts or rejects a single execution* of a program independently[1] of the behavior of all the other executions. This report introduces a monitoring mechanism which guaranties the respect of the confidentiality of secret data; either the monitor deduces that the current execution is non-interfering or it alters the behavior of the program in order to obtain a non-interfering execution.

The next section gives an overview of the approach. It defines some notions used, as well as introduces the scope of the work. Section 3 defines the monitoring semantics. This semantics is based on an automaton which is defined in the same section. The properties of monitored executions and a comparison with a type system are contained in Sect. 4. Then the report skims through related works in the domain of automata based monitoring and information flow monitoring. Finally, the conclusion comes in Sect. 6.

2 Outline

The work presented in this report aims at monitoring executions. So it is dealing with the notion of *non-interfering execution* and not with the notion of *non-interfering program*. An execution is said to be non-interfering if its public (low) outputs have the same values as the public outputs of any other execution having the same public (low) inputs. Based on the formal definition of non-interference given in (1), the property of being a *non-interfering execution* (started in the initial state σ) of a program $\mathtt{P}$ can be formalized as follows:

$$\forall \sigma' \in \Sigma : \ \sigma =_L \sigma' \Rightarrow [\![\mathtt{P}]\!]\sigma \approx_L [\![\mathtt{P}]\!]\sigma' \tag{2}$$

Compared to (1), the universal quantifier of σhas disappeared. The reason is that this property is specialized to a particular program state: the initial state of the execution of concern.

The notion of non-interference is intrinsically linked to the notion of information flow. This report distinguishes three types of information flows:

direct flows Such flows appear when an assignment is executed. For example, if the assignment $x := y$ is executed, then a direct flow from y to x is generated.

[1] Not *exactly* independently; but one execution can be detected as non-interfering even if some others are interfering.

indirect flows Such flows concern a flow from the *context* of execution to the value of a variable. For the sequential programs studied here, the context of execution consists only in the program counter. For multi-threaded programs, the context would also include the locks owned by other threads for example. There are two types of indirect flows:

explicit indirect flows Such flows appear when an assignment *is* executed. For example, if the statement **if** b **then** $x := y$ **else** `skip` **end** is executed with $b = \texttt{true}$, then an explicit indirect flow from b to x is generated.

implicit indirect flows Such flows appear when an assignment *is not* executed. For example, if the statement **if** b **then** $x := y$ **else** `skip` **end** is executed with $b = \texttt{false}$, then an implicit indirect flow from b to x is generated.

2.1 Background

The idea of using automata to monitor the *good behavior* of executions is not recent. Schneider [Sch00] characterized the type of security policies which can be enforced using execution monitors (also called truncation-automaton based monitors in [LBW05a] or reference monitors). Such types of monitors are only able to look at executed commands and to stop an execution. His conclusion is that this approach enables only the enforcement of *safety properties*. A safety property is above all a property. In the same paper, Schneider states:

> "In Alpern and Schneider [1985] and the literature on linear-time concurrent program verification, a set of executions is called a *property* if set membership is determined by each element alone and not by other members of the set." Schneider [Sch00, Sect. 2]

Considering a single program execution to be a sequence of actions, this definition of a property can be formalized as follows:

Definition 2.1 (Property [Sch00, LBW05a])**.** A security policy $\mathcal{P}$ is deemed to be a property of a set of executions Ω if and only if there exists a computable predicate $\widehat{\mathcal{P}}$ on executions such that $\mathcal{P}$ is a predicate over sets of executions with the following form:

$$\mathcal{P}(\Omega) = \forall \omega \in \Omega : \widehat{\mathcal{P}}(\omega)$$

Equation (2) shows that, for an execution, the property of being non-interfering depends on some other executions. Therefore, there is no predicate $\widehat{\mathcal{P}}$ on the sequence of actions evaluated by an execution such that $\widehat{\mathcal{P}}$ can decide if an execution is or is not non-interfering. So, truncation automata are not sufficient to enforce non-interference. The title of [TA05] ("Secure Information Flow as a Safety Problem"), by Terauchi and Aiken, may let the reader think that truncation automata can indeed enforce non-interference. In their paper, they reduce the non-interference problem to, what they call, a *2-safety problem*. Following the definition 2.1, 2-safety problems can be defined as follows:

Definition 2.2 (2-safety problem [TA05])**.** A security policy $\mathcal{P}$ is deemed to be a 2-safety problem of a set of executions Ω if and only if there exists a computable predicate $\widehat{\mathcal{P}}$ on pairs of executions such that $\mathcal{P}$ is a predicate over sets of executions with the following form:

$$\mathcal{P}(\Omega) = \forall \omega_1, \omega_2 \in \Omega : \widehat{\mathcal{P}}(\omega_1, \omega_2)$$

Although this definition enables non-interference to be expressed as a 2-safety problem and gives really good results with theorem provers and model checkers, it does not help execution monitors to deal with non-interference. First, it defines non-interference on the set of all executions. Following this definition, all the executions of a program, or none, are non-interfering. Furthermore, verifying that a set of executions Ω is non-interfering requires to apply $\widehat{\mathcal{P}}$ on all the pairs of Ω.

More powerful automata are described in [LBW05a]. They are called *Edit Automata*. On a given sequence of actions, those automata can insert, suppress or edit some actions in order to enforce what is called *infinite renewal* properties [LBW05b]. Those are still properties and thus non-interference does not belong to this set. However, as shown by this report, by increasing the information given to an automaton similar to Edit Automata, it is possible to enforce non-interfering executions.

The work presented in this report is not the first one trying to enforce confidentiality at run time. RIFLE [VBC+04] is a "runtime information-flow security system". It is designed to track the information flow during the execution of converted binaries. The system uses the collected information to enforce users' confidentiality policies. However, it does not enforce non-interference. The reason is that it does not take into consideration implicit indirect flows. By doing so, RIFLE ignores executions with equivalent public inputs but taking a different path. As Ashby states:

> "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." [Ash56, § 7/5 page 124].

The authors of RIFLE are conscious of this fact. They notice that, in order to enforce stronger constraints on the information flow (like non-interference), the monitoring mechanism must be aware of the commands which are not evaluated by a given execution. This is part of the approach taken in the work presented in this report.

2.2 The Approach Used

We consider a simple imperative language extended with an output statement. Its grammar is given in Fig. 1. Statements (S) are either sequences (S ; S), conditionals (B), or atomic statements (A). The output statement, "`output` e", is a generic statement used to represent any kind of public (low) output. For example, it can be used to represent the action of printing the value of an

expression e on the terminal running the program, producing a sound, or lay out a new window on the desktop screen. Only public outputs (i.e. outputs that are visible by standard users) are coded using the output statement. Secret outputs are simply ignored. For example, sending a message m on a public network is represented by the statement "`output` m"; on the other hand, sending an encrypted message n on a public network is abstracted by "`output` θ", where θ is a default constant which emphasizes the fact that an attacker is unable to decrypt an encoded message. 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.

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

Figure 1: Grammar of the language

The standard semantics of the language (Fig. 2) is described using evaluation rules, written $s \vdash S \stackrel{o}{\Rightarrow} s'$. This is read as follows: statement S executed in state σ yields state σ' and output o. A program state is simply a value store mapping variable names to their current value. $\sigma(x)$ is the value associated to the variable x in the store σ. This is extended to expressions, so that $\sigma(e)$ is the value of the expression e when the value store is σ. An output sequence is an empty sequence (written ϵ), a single value (for example, $\sigma(e)$) or the concatenation of two other sequences (written $o_1\ o_2$). The semantics is a standard *big-step operational semantics* [Kah87] (also called *natural semantics*); except for the output sequences which, however, come without any surprise.

The monitoring principles. Non-interference formalizes the fact that there is no information flowing from secret (high) inputs to public (low) outputs. In the approach taken in this report, the secret inputs of a program `P` are the initial values of the variables belonging to a set written $\mathcal{S}(\texttt{P})$. 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 considered as directly accessible by an attacker (even at the end of the execution). Consequently, the values of the variables in the program state are never considered as public outputs.

$$\sigma \vdash x := e \overset{\epsilon}{\Rightarrow} \sigma[x \mapsto \sigma(e)] \qquad (\mathrm{E}_{\mathcal{O}}\text{-ASSIGN})$$

$$\sigma \vdash \texttt{output}\ e \overset{\sigma(e)}{\Longrightarrow} \sigma \qquad (\mathrm{E}_{\mathcal{O}}\text{-PRINT})$$

$$\sigma \vdash \texttt{skip} \overset{\epsilon}{\Rightarrow} \sigma \qquad (\mathrm{E}_{\mathcal{O}}\text{-SKIP})$$

$$\frac{\sigma(e) = v \qquad \sigma \vdash S_v \overset{o}{\Rightarrow} \sigma'}{\sigma \vdash \textbf{if}\ e\ \textbf{then}\ S_{true}\ \textbf{else}\ S_{false}\ \textbf{end} \overset{o}{\Rightarrow} \sigma'} \qquad (\mathrm{E}_{\mathcal{O}}\text{-IF})$$

$$\frac{\sigma(e) = \texttt{true} \qquad \sigma \vdash S\ ;\ \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o}{\Rightarrow} \sigma'}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o}{\Rightarrow} \sigma'} \qquad (\mathrm{E}_{\mathcal{O}}\text{-WHILE}_{true})$$

$$\frac{\sigma(e) = \texttt{false}}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{\epsilon}{\Rightarrow} \sigma} \qquad (\mathrm{E}_{\mathcal{O}}\text{-WHILE}_{false})$$

$$\frac{\sigma \vdash S_1 \overset{o_1}{\Longrightarrow} \sigma' \qquad \sigma' \vdash S_2 \overset{o_2}{\Longrightarrow} \sigma''}{\sigma \vdash S_1\ ;\ S_2 \overset{o_1\ o_2}{\Longrightarrow} \sigma''} \qquad (\mathrm{E}_{\mathcal{O}}\text{-SEQ})$$

Figure 2: Semantics outputting the values of low-outputs

The main principle of the monitoring mechanism is based on notions of classical information theory [Ash56] about information transmission. 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)." Cohen [Coh77, Sect. 3].

Hence, for preventing information flows from secret inputs to public outputs, the monitoring mechanism will make sure that variety in 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 (this implies that the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ are fixed), will ensure that even if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ were different (bringing back variety in it) the output sequence would be identical; and hence, enforce that variety in $\mathcal{S}(\mathtt{P})$ is not conveyed to the output sequence.

The monitoring automaton has two jobs. The first one is to track "variety". By that, we mean to track entities (for example program variables, program counter, ...) which may have different values if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ were different. The second job is to prevent "variety" to be conveyed to the output sequence; in other words, to ensure that the output sequence would be identical for any execution for which the public inputs (i.e.

the initial values of the variables not belonging to $\mathcal{S}(\mathtt{P})$) are identical. In order 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 which have "variety" (i.e. which may have a different value if the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$ were different). The second element of the pair is a word belonging to the language whose alphabet is $\{\top, \bot\}$ and is described by the regular expression $(\top + \bot)^*$. This word tracks "variety" in the context of the execution (or the value of the program counter). The second job (preventing "variety" to be conveyed to the output sequence) is accomplished by authorizing, denying or editing output statements depending on the current state of the monitoring automaton.

What precision is lost by the abstraction? The automaton described here does not have information about the real values of the variables. It just knows if a variable *may* or *may not* have "variety". This feature prevents the automaton to detect some "safe" executions. An example of such an execution follows. The program has two inputs h (containing secrets) and l (containing public information).

```
x := l;
if h
  then x := 1
  else x := x / 2
end;
output x
```

The execution, for which h is true and l is 2, is safe. Whatever the value of h is, the executions of the program, for which l is 2, just output 1 (the variety in h is not conveyed to the output sequence). So, in those cases, there is no flow from h to what is outputted. However, to be able to discover this fact, the automaton would need some information about the real values of variables. This is not the case with the work proposed here, so the above example is out of reach of the proposed monitoring mechanism.

We will demonstrate in Sect. 4 that this mechanism is still of interest. Before that, the next section gives a formal definition of the monitoring automaton and of the monitored semantics.

3 Definition of the Monitoring Mechanism

The monitoring mechanism is divided into two main elements. The first one is an automaton similar to *Edit Automata* [LBW05a]. It inputs are abstractions of the actions accomplished during an execution. Its role is to track the information flow and authorize, forbid or edit the actions of the monitored execution in order to enforce non-interference. The second element of the monitoring mechanism is a semantics of monitored executions which merge together the behavior of the

monitoring automaton and of the standard output semantics given in Fig. 2. This section first describes the monitoring automaton and then the semantics of monitored executions.

3.1 The Automaton

The transition function of the automaton used to monitor an execution is independent of the program monitored. However, the set of states of the automaton depends on the program monitored.

Let $A^\star$ be the set of all strings over the alphabet A; additionally, $\mathcal{L}(R)$ is the language defined by the regular expression R. For any program $\mathtt{P}$, whose variables belongs to $\mathcal{V}(\mathtt{P})$ and the set of secret input variables is $\mathcal{S}(\mathtt{P})$ ($\mathcal{S}(\mathtt{P}) \subseteq \mathcal{V}(\mathtt{P})$), the automaton $\mathcal{A}(\mathtt{P})$ enforcing non-interference is defined as $(Q, \Phi, \Psi, \delta, q_0)$ where:

- Q is a set of states ($Q = 2^{\mathcal{V}(\mathtt{P})} \times \{\top, \bot\}^\star$),[2]
- Φ is a finite set of the input alphabet (specified shortly),
- Ψ is a finite set of the output alphabet (specified shortly),
- δ is a transition function $(Q \times \Phi) \longrightarrow (\Psi \times Q)$,
- q_0, an element of Q, is the start state ($q_0 = (\mathcal{S}(\mathtt{P}), \epsilon)$).

A state of the automaton is a pair, (V, w), composed of a set (V) of variables belonging to $\mathcal{V}(\mathtt{P})$ and a word (w) belonging to a language whose alphabet is $\top$ and $\bot$. At any step of the execution, V contains all the variables whose values *may* have been influenced by the initial values of the variables in $\mathcal{S}(\mathtt{P})$. w tracks variety in the context of the execution. In our case, the context consist only in the program counter value. If w contains $\top$ then the statement executed belongs to a branching statement whose condition may have been influenced by the initial values of $\mathcal{S}(\mathtt{P})$. This means that, with different values for the variables in $\mathcal{S}(\mathtt{P})$, the current statement may not have been executed. V is obviously finite, as the number of variables used by a given program is finite. The length of w can be bounded. Assuming that there is no recursive function calls, the maximum length of this word is equal to the maximum depth of branching statements. For example, with a program having no functions and a single branching statement, the second element of a state is a word whose maximum length is 1. Hence, for any program $\mathtt{P}$ without recursive function calls, the number of states of $\mathcal{A}(\mathtt{P})$ is finite.

The input alphabet of the automaton (Φ) corresponds to an abstraction of the events occurring during an execution. This alphabet is composed of the following:

`branch` e is generated each time a branching statement has to be evaluated. e is the expression which (or whose value) determines the branch which

[2] 2^X is the power set of X, also written $\mathcal{P}(X)$

is executed. For example, before the evaluation of the statement "**if** $x >$ `10` **then** S_1 **else** S_2 **end**", the input `branch` $x >$ `10` is sent to the monitoring automaton.

`exit` is generated each time a branching statement has been evaluated. For example, after the evaluation of the statement "**if** $x > 10$ **then** S_1 **else** S_2 **end**", the input `exit` is sent to the monitoring automaton.

`not` S is generated each time a piece of code, S, is not evaluated. It is sent just after the execution of the piece of code which has been executed instead of S. For example, the statement "**if** x **then** S_1 **else** S_2 **end**", with x being `true`, generates the automaton input `not` S_2 just after the execution of S_1.

A is any atomic action of the language (assignment, skip or output statement). Any such action is sent to the automaton for validation before its execution.

The output alphabet (Ψ) is composed of the following:

ACK is used as answer for any input which is not an atomic action of the language. The automaton acknowledges the reception of information useful for tracking potential information flows but which does not require an intervention of the automaton.

OK is used whenever the monitoring automaton authorizes the execution of an atomic action.

NO is used whenever the monitoring automaton forbids the execution of an atomic action.

A is 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.

Fig. 3 specifies the transition function of the monitoring automaton. A transition is written $(q, \phi) \xrightarrow{\psi} q'$. It is read as follows: in the state q, on reception of the input ϕ, the automaton moves to state q' and outputs ψ. This transition function uses two special functions. $FV(e)$ returns the set of variables appearing in e. For example, $FV(x + y)$ returns the set $\{x, y\}$. *modified*(S) is the set of all variables whose value may be modified by an execution of the statement S. This function is used to take into account the implicit indirect flows created whenever a branch is not executed. A formal definition of this function follows:

- *modified*$(x := e) = \{x\}$
- *modified*(`output` e) $= \emptyset$
- *modified*(`skip`) $= \emptyset$

- $\mathit{modified}(\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)$
- $\mathit{modified}(\textbf{while } e \textbf{ do } S \textbf{ done}) = \mathit{modified}(S)$
- $\mathit{modified}(S_1 \ ; \ S_2) = \mathit{modified}(S_1) \cup \mathit{modified}(S_2)$

As can be seen in Fig. 3, the automaton forbids or edits only the executions of output statements. For other inputs, the only thing done is to keep track in the set V of the variables that may contain some secret information, and keep track in w of the branching conditions encountered so far that were secret.

On the reception of an input "`branch` e" in the state (V, w), the automaton checks if the value of the branching condition (e) may be influenced by the initial values of $\mathcal{S}(\mathtt{P})$. To do so, it computes the intersection of the variables appearing in e with the set V. If the intersection is empty, then the value of e *is not* influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Then the new state of the automaton is (V, w') where w' is the concatenation of w and $\bot$. Otherwise, if the intersection is not empty, the transition function adds $\top$ instead of $\bot$ at the end of w. In any case, the automaton acknowledges the reception of the input by outputting ACK.

Whenever the execution exits a branching statement, the last letter of w is removed. This restores the information about the context to what it was before this branching statement.

The input "`not` S" is used to let the automaton know that, due to the value of a previous branching condition, the statement S has not been executed. This is done in order to be able to detect *implicit indirect flows*. On the reception of an input "`not` S", the automaton verifies if the statement S may have been executed with different values for $\mathcal{S}(\mathtt{P})$. It is the case if the context of execution carries *variety* (i.e. if w does not belong to $\mathcal{L}(\bot^*)$). In that case, the first element of the new state of the automaton is the union of the first element of the old state with the set of variables whose values may be modified by an execution of S. Otherwise, nothing is done.

The atomic action `skip` is perfectly safe[3]. Hence, on the reception of such an input, the automaton authorizes its execution by outputting OK and does nothing else.

When executing an assignment ($x := e$), two types of flows are created. The first one is a direct flow from the right part of the assignment (e) to the left part (x). The execution of the assignment in "$x := y$" creates a flow from y to x. The second one is an explicit indirect flow from the context of execution (i.e. the program counter) to the left part of the assignment. The execution of the assignment in "**if** b **then** $x := y$ **else** `skip` **end**" creates a flow from b to x. Those two flows are *always* created whenever an assignment is executed. What is important is to check if secret information is carried by one of those flows (i.e. if variety in $\mathcal{S}(\mathtt{P})$ is conveyed by one of those flows). Hence, whenever receiving an 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 the

[3] `skip` is considered safe because the non-interference definition considered in this work is not time sensitive.

$$((V,w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\top) \qquad \text{iff } FV(e) \cap V \neq \emptyset \qquad \text{(T-BRANCH-high)}$$

$$((V,w),\ \texttt{branch}\ e) \xrightarrow{ACK} (V, w\bot) \qquad \text{iff } FV(e) \cap V = \emptyset \qquad \text{(T-BRANCH-low)}$$

$$((V,wa),\ \texttt{exit}) \xrightarrow{ACK} (V, w) \qquad \text{(T-EXIT)}$$

$$((V,w),\ \texttt{not}\ S) \xrightarrow{ACK} (V \cup \mathit{modified}(S), w) \qquad \text{iff } w \notin \mathcal{L}(\bot^*) \qquad \text{(T-NOT-high)}$$

$$((V,w),\ \texttt{not}\ S) \xrightarrow{ACK} (V, w) \qquad \text{iff } w \in \mathcal{L}(\bot^*) \qquad \text{(T-NOT-low)}$$

$$((V,w),\ \texttt{skip}) \xrightarrow{OK} (V, w) \qquad \text{(T-SKIP)}$$

$$((V,w),\ x\ :=\ e) \xrightarrow{OK} (V \cup \{x\}, w) \qquad \text{iff } w \notin \mathcal{L}(\bot^*) \text{ or } FV(e) \cap V \neq \emptyset \qquad \text{(T-ASSIGN-sec)}$$

$$((V,w),\ x\ :=\ e) \xrightarrow{OK} (V \setminus \{x\}, w) \qquad \text{iff } w \in \mathcal{L}(\bot^*) \text{ and } FV(e) \cap V = \emptyset \qquad \text{(T-ASSIGN-pub)}$$

$$((V,w),\ \texttt{output}\ e) \xrightarrow{OK} (V, w) \qquad \text{iff } w \in \mathcal{L}(\bot^*) \text{ and } FV(e) \cap V = \emptyset \qquad \text{(T-PRINT-ok)}$$

$$((V,w),\ \texttt{output}\ e) \xrightarrow{\texttt{output}\ \theta} (V, w) \qquad \text{iff } w \in \mathcal{L}(\bot^*) \text{ and } FV(e) \cap V \neq \emptyset \qquad \text{(T-PRINT-def)}$$

$$((V,w),\ \texttt{output}\ e) \xrightarrow{NO} (V, w) \qquad \text{iff } w \notin \mathcal{L}(\bot^*) \qquad \text{(T-PRINT-no)}$$

Figure 3: Transition function of monitoring automata

execution of the assignment of the program "**if** b **then** $x := y$ **else** `skip` **end**", y is the origin of the direct flow to x; and b is the origin of the explicit indirect flow to x. When receiving the input $x := e$ in the state (V, w), the origin of the explicit indirect flow is influenced by $\mathcal{S}(\mathtt{P})$ only if w does not belongs to the language defined by the regular expression $\bot^*$. If $w$contains $\top$, it means that the value of the condition of a previous (but still active) branching statement was potentially influenced by the initial values of $\mathcal{S}(\mathtt{P})$. When receiving the input $x := e$ in the state (V, w), the origin of the direct flow is influenced by $\mathcal{S}(\mathtt{P})$ only if the intersection of the variables appearing in e with $V$is not empty. If the value of e is influenced by the initial values of $\mathcal{S}(\mathtt{P})$ then at least one of the variable appearing in e has its value influenced by the initial values of $\mathcal{S}(\mathtt{P})$. Those variables are then members of V. Lets call (V', w') the new automaton state after the transition. If the origin of one of those flows is influenced by the initial values of $\mathcal{S}(\mathtt{P})$, then the variable on the left side of the assignment is added to $V(V' = V \cup \{x\})$. If none of the origins are influenced by the initial values of $\mathcal{S}(\mathtt{P})$, it means that the variable on the left side of the assignment receives a new value which is not influenced by $\mathcal{S}(\mathtt{P})$. In that case, the variable on the left side of the assignment is removed from $V(V' = V \setminus \{x\})$. This makes the mechanism flow-sensitive and enables it to deal with executions containing the action "$x := h$", followed later on by "$x := l$", and finally outputting the value of x.

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 outputted. In a public context (w belongs to $\mathcal{L}(\bot^*)$), if the program tries to output a secret (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 to the user letting him know 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 does exist because, depending on the path followed, some outputs may or may not be executed. In that case, *any* output must be forbidden; and the automaton outputs NO.

3.2 The Semantics

The semantics merging the standard output semantics given in Fig. 2 and the monitoring automaton is given in Fig. 4.

There are three rules for atomic actions (those actions are: `skip`, $x := e$ and `output` e). There is one rule for each possible answer of the automaton to the action which will be executed. Either the automaton authorizes the execution ($\mathrm{E}_{M(s)}$-OK), denies the execution ($\mathrm{E}_{M(s)}$-NO) or replaces the action by another one ($\mathrm{E}_{M(s)}$-EDIT). In the case where the execution is denied, the evaluation omits the current action (as if the action was a skip statement). In the case where the action to be executed (A) is replaced by another one (A'), on reception of the input A the monitoring automaton returns A'; and the monitoring semantics execute A' instead of A. With the transition function of

the automaton presented in this work, A' can only be the action outputting the default value ($\texttt{output}\ \theta$).

If the statement to be executed is a branching command, the evaluation begins by sending to the automaton the input "$\texttt{branch}\ e$" where e is the condition of the branching command. Then, the branch designated by e is executed (in the case where the branching command is a while statement and the condition is `false`, the branch executed is `skip`). The execution follows by sending the automaton input "$\texttt{not}\ S$" where S is the branch not executed (if the branching command is a while statement and the condition 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.

$$\frac{(q,\ A) \xrightarrow{OK} q' \qquad \sigma \vdash A \overset{o}{\Rightarrow} \sigma'}{(q,\ \sigma) \Vdash A \overset{o}{\Rightarrow} (q',\ \sigma')} \quad (\mathrm{E}_{M(s)}\text{-}OK)$$

$$\frac{(q,\ A) \xrightarrow{A'} q' \qquad \sigma \vdash A' \overset{o}{\Rightarrow} \sigma'}{(q,\ \sigma) \Vdash A \overset{o}{\Rightarrow} (q',\ \sigma')} \quad (\mathrm{E}_{M(s)}\text{-EDIT})$$

$$\frac{(q,\ A) \xrightarrow{NO} q}{(q,\ \sigma) \Vdash A \overset{\epsilon}{\Rightarrow} (q,\ \sigma)} \quad (\mathrm{E}_{M(s)}\text{-}NO)$$

$$\frac{\begin{array}{c}\sigma(e) = v \qquad (q,\ \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1,\ \sigma) \Vdash S_v \overset{o}{\Rightarrow} (q_2,\ \sigma_1) \\ (q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{ACK} q_3 \qquad (q_3,\ \texttt{exit}) \xrightarrow{ACK} q_4\end{array}}{(q,\ \sigma) \Vdash \textbf{if}\ e\ \textbf{then}\ S_{\texttt{true}}\ \textbf{else}\ S_{\texttt{false}}\ \textbf{end} \overset{o}{\Rightarrow} (q_4,\ \sigma_1)} \quad (\mathrm{E}_{M(s)}\text{-IF})$$

$$\frac{\begin{array}{c}\sigma(e) = \texttt{true} \qquad (q,\ \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1,\ \sigma) \Vdash S \overset{o_l}{\Rightarrow} (q_2,\ \sigma_1) \\ (q_2,\ \texttt{exit}) \xrightarrow{ACK} q_3 \\ (q_3,\ \sigma_1) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_w}{\Rightarrow} (q_4,\ \sigma_2)\end{array}}{(q,\ \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{o_l\ o_w}{\Rightarrow} (q_4,\ \sigma_2)} \quad (\mathrm{E}_{M(s)}\text{-WHILE}_{true})$$

$$\frac{\begin{array}{c}\sigma(e) = \texttt{false} \qquad (q,\ \texttt{branch}\ e) \xrightarrow{ACK} q_1 \\ (q_1,\ \texttt{not}\ S) \xrightarrow{ACK} q_2 \qquad (q_2,\ \texttt{exit}) \xrightarrow{ACK} q_3\end{array}}{(q,\ \sigma) \Vdash \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done} \overset{\epsilon}{\Rightarrow} (q_3,\ \sigma)} \quad (\mathrm{E}_{M(s)}\text{-WHILE}_{false})$$

$$\frac{\begin{array}{c}(q,\ \sigma) \Vdash S_1 \overset{o_1}{\Rightarrow} (q_1,\ \sigma_1) \\ (q_1,\ \sigma_1) \Vdash S_2 \overset{o_2}{\Rightarrow} (q_2,\ \sigma_2)\end{array}}{(q,\ \sigma) \Vdash S_1\ ;\ S_2 \overset{o_1\ o_2}{\Rightarrow} (q_2,\ \sigma_2)} \quad (\mathrm{E}_{M(s)}\text{-SEQ})$$

Figure 4: Semantics of monitored executions

3.3 Example of monitored execution

Figure 5 lays out the evolution of the monitoring mechanism during a monitored execution. The program whose execution is monitored is given in column "Program P". This program has two inputs h and l. The execution monitored is the one for which h equals `true` and l equals `22`. The set of secret inputs of P ($\mathcal{S}(\mathtt{P})$) is $\{h\}$. The atomic actions (i.e. assignments, skip statements and output statements) that the program will attempt to execute are given in column "Proposed action". The actions in this column are those which would have been executed if the execution was *not* monitored. The next column contains the input which is sent to the automaton for each "proposed" action. The two following columns contain the result of the automaton transition function on the automaton input found on the same line (the transition function is applied on the automaton input of the same line and the automaton state of the previous line). "Automaton output" contains the output of the automaton sent back to the semantics, and "Automaton 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.

As can be seen in the column "Automaton output", in the majority of cases the monitoring automaton just acknowledges the reception of an input or authorizes the execution of an action without altering the normal behavior of the program. In this example, there are only two alterations of the execution (on lines 6 and 11). The first one occurs when the program tries to output a value which has been influenced by $\mathcal{S}(\mathtt{P})$. This output action is "`output y;`". At this point of the execution, the value contained in y as been influenced by the initial values of the variables belonging to $\mathcal{S}(\mathtt{P})$. This is know because y belongs to the first element of the automaton state before the execution of line 6. This automaton state can be seen in the column "Automaton state" of the previous line (the value in line 6 correspond to the state *after* the execution of line 6). In consequence, the automaton disallows the output of this value. However, the fact of outputting something in itself is safe because the context of execution has not been influenced by $\mathcal{S}(\mathtt{P})$ (the second element of the automaton state belongs to $\mathcal{L}(\bot^\star)$). Hence, the automaton sends to the semantics an output action to execute. The value of this output action 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 11, the program tries to output something while the current context of execution (the program counter) has been influenced by $\mathcal{S}(\mathtt{P})$ (the second element of the automaton state does not belong to $\mathcal{L}(\bot^\star)$). Hence, if the output occurs then an attacker could learn something about the 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;`".

	Program P	Proposed action: $h = \texttt{true}$ & $l = 22$	Automaton input	Automaton output	Automaton state: $(\{h\}, \epsilon)$	Executed
1	$x := l + 3;$	$x := l + 3;$	$x := l + 3;$	OK	$(\{h\}, \epsilon)$	$x := l + 3;$
2	**if** $(x > 10)$		branch $x > 10$	ACK	$(\{h\}, \bot)$	
3	**then**					
4	$y := h;$	$y := h;$	$y := h;$	OK	$(\{h,y\}, \bot)$	$y := h;$
5	output $x;$	output $x;$	output $x;$	OK	$(\{h,y\}, \bot)$	output $x;$
6	output $y;$	output $y;$	output $y;$	output $\theta;$	$(\{h,y\}, \bot)$	output $\theta;$
7	$y := 0;$	$y := 0;$	$y := 0;$	OK	$(\{h\}, \bot)$	$y := 0;$
8	**if** (h)		branch h	ACK	$(\{h\}, \bot\top)$	
9	**then**					
10	$z := 0;$	$z := 0;$	$z := 0;$	OK	$(\{h,z\}, \bot\top)$	$z := 0;$
11	output $x;$	output $x;$	output $x;$	NO	$(\{h,z\}, \bot\top)$	
12	**if** $(l > 20)$		branch $l > 20$	ACK	$(\{h,z\}, \bot\top\bot)$	
13	**then**					
14	$x := l;$	$x := l;$	$x := l;$	OK	$(\{h,z,x\}, \bot\top\bot)$	$x := l;$
15	**else**					
16	skip;		not skip	ACK	$(\{h,z,x\}, \bot\top\bot)$	
17	**end;**		exit	ACK	$(\{h,z,x\}, \bot\top)$	
18	**else**					
19	$y := 1;$		not $y := 1;$	ACK	$(\{h,z,x,y\}, \bot\top)$	
20	**end;**		exit	ACK	$(\{h,z,x,y\}, \bot)$	
21	**else**					
22	skip;		not skip	ACK	$(\{h,z,x,y\}, \bot)$	
23	**end;**		exit	ACK	$(\{h,z,x,y\}, \epsilon)$	

Figure 5: Example of the automaton evolution during an execution.

4 Efficiency of the Monitoring Mechanism

The preceding section gives a formal definition of the monitoring mechanism and an example of its behavior. This section studies the efficiency of this monitoring mechanism by giving bounds on the set of executions obtained by using this monitoring mechanism. First, it is proved that any monitored execution belongs to the set of non-interfering executions. This is equivalent to a soundness proof. Then, it is proved that a non trivial set of unmonitored non-interfering executions is included in the set of monitored executions.

4.1 Soundness

The soundness property of the monitoring mechanism is based on a notion of non-interference between the secret inputs and the sequence outputted by an execution. An execution is considered *safe* if and only if this execution does not convey the variety in its secret inputs to the sequence outputted during this execution; in other terms, if the secret inputs have no influence on what is outputted during this execution.

Before stating the soundness theorem, this section gives the definition of a notation designating the output sequence of the execution of a program P started in the initial state σ.

Definition 4.1 (Output of a monitored execution: $[\![\mathtt{P}]\!]\sigma$)**.**
For all program P whose secret inputs belong to $\mathcal{S}(\mathtt{P})$, and value store σ, "$[\![\mathtt{P}]\!]\sigma$" is the sequence outputted by the execution of P with the initial state "$(\mathcal{S}(\mathtt{P}), \epsilon), \sigma$". In other words:

$$[\![\mathtt{P}]\!]\sigma = o \text{ if and only if } \exists\ \sigma' :\ (\mathcal{S}(\mathtt{P}), \epsilon), \sigma \vdash \mathtt{P} \overset{o}{\Rightarrow}_s \sigma'$$

The following theorem states that any monitored execution is *safe*; i.e. it is non-interfering. $\overset{X}{=}$ is an equivalence relation between value stores. This relation is true whenever the two stores associate the same value to any variable belonging to X. Using this relation, the fact that σ_1 and σ_2 are indistinguishable for X is written $\sigma_1 \overset{X}{=} \sigma_2$. X^c is the complement of the set X.

Theorem 4.1 (Monitored executions are non-interfering)**.**
For all program P *which set of secret inputs is* $\mathcal{S}(\mathtt{,})$ *and value stores* σ_1 *and* σ_2*:*

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

Proof. This theorem follows directly from lemma B.6 page 35. □

4.2 Monitoring Automaton versus Type System

It has been proved that any monitored execution is non-interfering. This is a required result. However, in order to achieve this goal, in some cases the monitoring mechanism modifies the output sequence of the execution. The

sequence of outputs resulting from the execution of a program P with the initial state σ may not be the same if the semantics used is the standard one (given in Fig. 2) or the monitoring semantics (given in Fig. 4). As long as the monitoring mechanism accomplishes its job, the lesser impact the better. The sequel gives a lower bound on the set of non-interfering executions on which the monitoring mechanism has no impact. It is shown that the mechanism proposed in this report does not interfere with the output sequence of any execution of a program which is well-typed under a security type system similar to the one of Volpano and al. [VSI96].

Figure 6 shows the security type system. It is the same one as [VSI96] except for a small modification of the typing environment and the addition of a rule for the output statement (which is not in the language of [VSI96]). The typing environment in [VSI96] is a pair (λ, γ) where λ prescribes types for locations and γ prescribes types for identifiers. As our language does not have locations, only γ has been kept for the type system given here. Additionally, γ 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 information 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. $\forall x \in \mathcal{S}(\mathtt{P}), \gamma(x) = H$).

$$\frac{\gamma(e) = \tau' \qquad \tau' \leq \tau}{\gamma \vdash e : \tau} \qquad \text{(T-EXP)}$$

$$\frac{\gamma(x) = \tau' \qquad \gamma \vdash e : \tau' \qquad \tau \leq \tau'}{\gamma \vdash x := e : \tau \text{ cmd}} \qquad \text{(T-ASSIGN)}$$

$$\frac{\tau \leq H}{\gamma \vdash \mathtt{skip} : \tau \text{ cmd}} \qquad \text{(T-SKIP)}$$

$$\frac{\gamma \vdash e : L}{\gamma \vdash \mathtt{output}\ e : L \text{ cmd}} \qquad \text{(T-PRINT)}$$

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash S_1 : \tau' \text{ cmd} \qquad \gamma \vdash S_2 : \tau' \text{ cmd} \qquad \tau \leq \tau'}{\gamma \vdash \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end} : \tau \text{ cmd}} \qquad \text{(T-IF)}$$

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash S : \tau' \text{ cmd} \qquad \tau \leq \tau'}{\gamma \vdash \textbf{while } e \textbf{ do } S \textbf{ done} : \tau \text{ cmd}} \qquad \text{(T-WHILE)}$$

$$\frac{\gamma \vdash S_1 : \tau \text{ cmd} \qquad \gamma \vdash S_2 : \tau \text{ cmd}}{\gamma \vdash S_1 ; S_2 : \tau \text{ cmd}} \qquad \text{(T-SEQ)}$$

Figure 6: The type system used for comparison

We can prove (Theorem 4.2) that any execution of a well-typed program

belongs to the set of monitored executions. This means that the monitoring mechanism does not interfere with executions of well-typed programs. To be convinced that the inclusion in question here is strict it is sufficient to have a look at the following program:

```
x := h;
x := 0;
output x
```

In this program, h is the only secret input. Any execution of this program is obviously non-interfering. However, 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 4.2 (Monitoring does not interfere with *type safe* programs)**.**
For all program P *whose secret inputs belong to* $\mathcal{S}(P)$*, typing environment* γ *such that variables belonging to* $\mathcal{S}(P)$ *are typed secret, type* τ*, and value stores* σ *and* σ'*:*

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

Proof. This theorem follows directly from lemma B.11 page 46. □

5 Related Work

Automaton-based monitoring and static information flow analyzes. There has already been research on reference monitors. For example, Erlingsson and Schneider have developed a monitoring tool called *SASI* [ES99]. The properties enforced by their monitors are expressed using security automata (as defined in [Sch00]). Their tool is then able to in-line monitors enforcing those properties directly into object code (either x86 assembly language or Java Virtual Machine Language). Following another approach, Hamlen et al. [HMS06] develop an extension to the .NET Common Intermediate Language called *Mobile*. This extension supports a type system enforcing the certification of in-lined reference monitors. *Mobile* can check that a property expressed as an ω-regular expression is enforced by a self-monitoring program. Therefore, it removes the need to trust the rewriting process in-lining the monitor. A restriction of reference monitors is their limited control over the execution . They are only able to halt an execution. Schneider [Sch00] shows that such monitors are limited to the enforcement of *safety properties*. In a successful attempt to increase the power of monitors, Ligatti et al. [LBW05a] introduce *edit automata*. Monitors based on such automata are able to alter the behavior of an execution by modifying the sequence of actions executed. Ligatti et al. [LBW05b] show that those monitors are able to enforce *infinite renewal properties*.

However, to the best of our knowledge, none of those monitors enforce security policies based on strong information flow properties. The vast majority of the research on information flow concerns static analyses and involves type systems [SM03]. In the recent years, this approach has reached a good level of maturity. Pottier and Conchon described in [PC00] a systematic way of producing a type system usable for checking *noninterference*. Profiting from this maturity, some "real size" languages including a security oriented type system have been developed. Among them are JFlow [Mye99a], JIF [MNZZ01], and FlowCaml [Sim02, PS03]. One of the drawbacks of type systems concerns the level of approximation involved. In order to improve the precision of those static analyses, dynamic security tests have been included into some languages and taken into account in the static analyses. The JFlow language [Mye99a, Mye99b], which is an evolution of Java, uses the *decentralized label model* of Myers and Liskov [ML98]. In this model, variables receive a label which describes allowed information flows among the principals of the program. Some dynamic tests of the principals hierarchy and variables labels are possible, as well as some labels modifications [ZM01]. Zheng and Myers [ZM04] include dynamic security labels which can be read and tested at run-time. Nevertheless, labels are not computed at run-time. Using dynamic security tests similar to the Java stack inspection, Banerjee and Naumann developed in [BN03] a type system guarantying noninterference for well-typed programs and taking into account the information about the calling context of method given by the dynamic tests.

Dynamic information flow analyzes. Even so information flow monitoring is not as popular as information flow static analyses, there has continuously been some research concerning it.

At the level of languages, Abadi, Lampson, and Lévy expose in [ALL96] a dynamic analysis based on the labeled λ-calculus of Lévy. This analysis computes the dependencies between the different parts of a λ-term and its final result in order to save this result for a faster evaluation of any future equivalent λ-term. Also based on a labeled λ-calculus, Gandhe, Venkatesh, and Sanyal [GVS95] address the information flow related issue of *need*. It has to be noticed that even some "real world" languages dispose of similar mechanisms. The language Perl includes a special mode called "Perl Taint Mode" [WCO00]. In this mode, the *direct* information flows originating with user inputs are tracked. It is done in order to prevent the execution of "bad" commands. None of those works take into account *implicit indirect flows* (created by the non-execution of one of the branches of a branching statement).

At the level of operating systems, Weissman [Wei69] described at the end of the 60's a security control mechanism which dynamically computes the security level of newly created files depending on the security level of files previously opened by the current job. Following a similar approach, Woodward presents its *floating labels* method in [Woo87]. This method deals with the problem of over-classification of data in computer systems implementing the MAC security model [NSA95, Bra85]. The main difference between those two works and ours

lies in the granularity of label application. In those models [Wei69, Woo87], at any time, there is only one label for all the data manipulated. Data's "security levels" cannot evolve separately from each other. More recently, Suh, Lee, Zhang, and Devadas presented in [SLZD04] an architectural mechanism, called *dynamic information flow tracking*. Its aim is to prevent an attacker to gain control of a system by giving *spurious* inputs to a program which may be buggy but is not malicious. Their work looks at the problem of security under the aspect of integrity and does not take care of information flowing indirectly throw branching statements containing different assignments.

At the level of computers themselves, Fenton [Fen74] describes a small machine, in which storage locations have a *fixed* data mark. Those data marks are used to ensure a secure execution with regard to noninterference between private inputs and non-private outputs. However, the fixed characteristic of the data marks forbids modularity and reuse of code by disallowing a temporary variable to contain alternatively secrets and public information. As Fenton shows himself, his mechanism does not ensure confidentiality with *variable* data marks. At the same level, Brown and Knight [BK01] describe a machine which dynamically computes security level of data in memory words and try to ensure that there are no undesirable flows. This work does not take care of non-executed commands. As it has been shown in [LGJ05], this is a feature which can be used to gain information about secrets in some cases. With a program similar to the following one, their machine does not prevent the flow from `h` to `x` when `l` is true and `h` is false.

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

In [MPL04], Masri, Podgurski and Leon present a dynamic information flow analysis for structured or unstructured language. Their algorithm seems to achieve a good level of precision for a quite complete language. However, for the analysis of a method (which is the unit on which their algorithm applies), their approach "requires that its control flow graph [...] has been computed beforehand". Their approach is then not fully dynamic. Moreover, their work seems to focus more on dynamic slicing than on non-interference monitoring. Consequently, it does not study deeply the dynamic correction of "bad" flows. The solution proposed is to stop the execution as soon as a potential flow from a secret data to a public *sink* is detected. As explained in [LGJ05], if done without enough care, this can create a new covert channel revealing some secret information. To avoid this, the information flows computed must be the same for every low-input equivalent execution. This property is not proved for the proposed algorithm.

The case of RIFLE [VBC+04] is different from the research works presented above. It is a complete runtime information flow security system enforcing

"user-centric" security policies. It includes a binary translator and a specific architecture which, together, track the information flows. Based on this information, a security enhanced OS enforces user policies. However, as acknowledge by the authors, their system does not take into consideration implicit indirect flows. It is the impossible for their mechanism to enforce non-interference like policies. And, hence, it is impossible to enforce strict confidentiality.

6 Conclusion

This report addresses the security problem of confidentiality from the point of view of non-interference. It is usually a property of a program. Either a program is non-interfering or it is not. As we are interested in monitoring executions to ensure the respect of confidentiality, the notion of non-interference is refined to a notion of *non-interfering execution*. An execution ε is said to be non-interfering if and only if any execution of the same program with the same public inputs (as for ε) produces the same public outputs.

The monitoring mechanism proposed in the report is based on an automaton and a special semantics. During the execution, the semantics sends to the automaton inputs abstracting the events occurring. The automaton is in charge of two main jobs. The first one is to track the flows of information between the secret inputs and the current value of the variables used by the program. The second one is to validate the execution of atomic actions (mainly outputs) in order to ensure the respect of the confidentiality of the secret inputs. In Sect. 4.1, it has been proved that any execution monitored by the proposed mechanism is a non-interfering execution. This means that an attacker having access to the low outputs of a monitored execution is never able to deduce anything about the value of the secret inputs. An additional interesting property which has been proved is that the monitoring mechanism does not interfere with the executions of a program which is well-typed under a type system similar to the one of Volpano, Smith and Irvine [VSI96].

Typicality of non-interference monitoring. In order to enforce a property as strong as non-interference, this monitoring mechanism (as any non-interference monitor would) has a principal particularity compared to standard monitors. Usually, monitors are only aware of statements which are really executed. With the proposed mechanism, when exiting a branching statement, the branch which has not been executed is analyzed. This is done in order to take into account a special type of flows: implicit indirect flows. These flows appear between the condition of a branching statement and all the variables on the left side of an assignment in the branch which is not executed. As written by [VBC+04], this feature is required in order to enforce non-interference.

Of course, monitoring an execution has a cost. So, what are the main interests of non-interference monitoring compared to static analyzes? The first one lies in the granularity of the non-interference property. Static analyzes have to take into consideration all possible executions of the program analyzed. This

implies that if a single execution is unsafe then the program is rejected; and then all of its executions. With a monitoring mechanism, it is possible to allow the safe executions of a program which is known to have some unsafe executions.

Moreover, a monitoring mechanism may be more precise than static analyzes. The reason for it is that during the execution the monitoring mechanism gets some accurate information about the "path behavior" of the program. An example being sometimes more understandable, let us have a look at 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 outputted. However, if `test1` and `test2` are such that there exists no value such that both predicates are true then any execution of the program is perfectly safe. 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 than the current execution is taken care 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.

Future work. Of course, increasing the expressiveness of the language is a first potential future work. Adding method call or even records should be quite straightforward. Whereas, in my opinion, adding pointers to the language would not be trivial. Another interesting feature, which is under work, is concurrency. The language is extended with a synchronization construct and the monitor is adapted to deal with concurrent execution of a set of sequential programs.

The analysis used on unevaluated statements is another point which would be worth some extended work. The analysis used in this report is a really simple one collecting the variables appearing on the left side of an assignment. By increasing the precision of this analysis, a better precision would be achieved. The following example emphasizes the limitation of the current analysis.

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

`h` is the only input of the program. It is a secret input. This program outputs 0 whatever the value of `h`. It is then perfectly safe. However, the monitoring mechanism detects any execution of this program as unsafe. The reason is that the monitor does not take into consideration the values of branching conditions. In this case, it may seem simple to improve the monitoring mechanism. However, as explained in [LGJ05], if the goal of the mechanism is to correct bad flows and not only detect them then the analysis must be done with great care. It is required that, for a branching statement, whatever the branch executed and the branch analyzed, the result of the information flow tracking must be the same.

Finally, in the work proposed in this report, there are only two categories of data: public and secret. It would be interesting to have more. In order to achieve this goal, we can remark that, even if there are two categories, there is only one property on data: either data is secret or is not. Based on this idea of "property" of data, the number of categories can be extended by increasing the number of properties and changing the definition of automaton states. A new state is a set of old states, one for each property. The solution is quite simple, but its impact on the different proofs must be studied.

A Nomenclature

2^X is the power set of X, also written $\mathcal{P}(X)$.

$A^\star$ is the set of all strings over the alphabet A.

$\mathcal{L}(E)$ is the language described by the regular expression E.

$FV(e)$ is the set of free variables appearing in the expression e.

$\mathit{modified}(S)$ is the analysis used for unevaluated statements. It returns the set of variables which would be potentially be assigned to by an execution of S.

$\mathtt{P}$ is a program of name P.

$\mathcal{V}(\mathtt{P})$ is the set of variables used by the program $\mathtt{P}$.

Ω is a set of executions.

ω is an execution (a sequence of actions).

$y \overset{X}{=} z$ is an equivalence relation which states that y and z are indistinguishable for X.

$\mathcal{S}(\mathtt{P})$ is the set of variables used as secret inputs by the program $\mathtt{P}$.

$\widehat{\mathcal{P}}$ is a predicate on executions.

$\mathcal{P}$ is a security policy.

$\mathcal{A}(\mathtt{P})$ is the monitoring automaton for the program $\mathtt{P}$.

Q is the set of states of monitoring automata.

Φ is the input alphabet of monitoring automata.

Ψ is the output alphabet of monitoring automata.

δ is the transition function between automata states.

ϕ is an input to a monitoring automaton.

ψ is an output of a monitoring automaton.

q_0 is the start state of a monitoring automaton.

q is a state of a monitoring automaton.

V is the first element of an automaton state. It is a set of variables.

w is the second element of an automaton state. It is a word (or string) belonging to $\{\top, \bot\}^\star$.

L is the type for public information.

H is the type for secret information.

γ is a typing environment.

θ is the default output message sent whenever an action outputting a secret is detected.

σ represents a program state during an execution.

Σ is the set of all execution states.

σ is a value store mapping variable names to their current value.

B Proofs

The following proofs have been written before a modification of the notations used and has not been rewritten yet. In consequence, some of the notations found in the proofs are different from those used in the previous sections. However the modifications are straight forward.

An automaton state is now represented by a set of variables and a word (V, w). In the proofs, it is the same set of variables with an integer (V, n). w is a binary representation of n where $\top$ is 1 and $\bot$ is 0. Therefore, $w \in \mathcal{L}((\,)\bot^\star)$ is equivalent to $n = 0$.

Additionally, the function collecting the variable whose value may be modified by the execution of a statement S, called $\mathit{modified}(S)$ in the previous sections, is called $\mathcal{LA}(S)$ in the proofs.

Finally, the automaton output ACK is not used in the proofs. The automaton output OK is used instead.

B.1 Proofs of Sect. 4.1 (Soundness)

Lemma B.1 (Same context before and after an execution)**.** *For all statement S, automaton states $q = (V, n)$ and $q_f = (V_f, n_f)$, and value store σ, if $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$ then $n_f = n$.*

Proof. The fact that $n_f = n$ follows directly from the definition of the semantics ($\mathrm{E}_{M(O)}$), and the definition of the transition function of the monitoring automaton. □

Lemma B.2 (No outputs under secret context)**.**
For all statement S, automaton state (V, n), and value store σ, if $n > 0$ then $[\![S]\!]^{\mathcal{O}}_{M(\mathcal{O})}((V, n), \sigma) = \epsilon$.

Proof. It follows directly from the definitions of the semantics ($\mathrm{E}_{M(O)}$) and (E_O) and from the transition function of the monitoring automaton. The only statement outputting anything is "`output` e". The semantics ($\mathrm{E}_{M(O)}$) always call the automaton to verify any action it will evaluates. Whenever the context of execution is secret (i.e. $n > 0$), the monitoring automaton deny the execution of any print statement. □

Lemma B.3 (Automaton simulates execution in secret context)**.**
For all statement S, automaton state $q = (V, n)$, and value store σ, if $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$ and $n > 0$ then $(q, \texttt{not } S) \rightarrow q_f$.

Proof. The proof goes by induction on the derivation tree of "$((V, n), \sigma) \vdash S \overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{M(s)}$-OK) then we can conclude that :

(1) $S =$ "`skip`" or $S =$ "y := e" or $S =$ "`output` e", and "$(q, S) \xrightarrow{OK} q_f$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-OK) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{M(s)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

Case 1: $S =$ "`skip`" or $S =$ "`output` e"

(a) $q_f = q$.
It follows directly from the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(S) = \emptyset$.

Case 2: $S =$ "x := e"

(a) $q_f = (V \cup \{x\}, n)$.
It follows directly from the transition function of the monitoring automaton because $n > 0$.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(x := e) = \{x\}$.

($\mathbf{E}_{M(s)}$-EDIT) then we can conclude that :

(1) $S =$ "`output` e", and "$(q, S) \xrightarrow{A'} q_f$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-EDIT) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{M(s)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

(2) $q_f = q$.
It follows directly from the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(\texttt{output}\ e) = \emptyset$.

($\mathbf{E}_{M(s)}$-NO) then we can conclude that :

(1) $S =$ "`output` e", and "$(q, S) \xrightarrow{A'} q_f$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-NO) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{M(s)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

(2) $q_f = q$.
It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-NO).

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(\texttt{output}\ e) = \emptyset$.

($\mathbf{E}_{M(s)}$-IF) then we can conclude that :

(1) $S =$ "**if** e **then** S_1 **else** S_2 **end**", $\sigma(e) = v$, and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Longrightarrow}_{M(s)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-IF).

Lets define $q_i = (V_i, n_i)$ for all integer i from 1 to 3.

(2) $n_1 > 0$.
It follows directly from the global hypothesis $n > 0$ and the definition of the transition function of the monitoring automaton.

(3) $V_2 = V \cup \mathcal{LA}(S_v)$.
It follows from the application of the inductive hypothesis to the evaluation of S_v in the local conclusion (1) and the definition of the transition (T-NOT-high).

(4) $q_f = (V \cup \mathcal{LA}(S_v) \cup \mathcal{LA}(S_{\neg v}), n)$.
From the local conclusion (2), the evaluation of S_v in the local conclusion (1) and lemma B.1, it is possible to show that $n_2 > 0$. So, the definition of the transition function of the monitoring automaton imply that $V_3 = V_2 \cup \mathcal{LA}(S_{\neg v})$. Finally the desired result is obtain using the local conclusion (3), the definition of (T-EXIT) and lemma B.1.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(\textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) = \mathcal{LA}(S_1) \cup \mathcal{LA}(S_2)$.

($\mathbf{E}_{M(s)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) $S =$ "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow}_{M(s)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash \textbf{while}\ e\ \textbf{do}\ S_l\ \textbf{done} \overset{o_w}{\Longrightarrow}_{M(s)} (q_f,\ \sigma_2)$

It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-WHILE$_{\texttt{true}}$).

Lets define $q_i = (V_i, n_i)$ for all integer i from 1 to 3.

(2) $n_1 > 0$.
It follows directly from the global hypothesis $n > 0$ and the definition of the transition function of the monitoring automaton.

(3) $V_3 = V \cup \mathcal{LA}(S_l)$.
It follows from the application of the inductive hypothesis to the evaluation of S_l in the local conclusion (1) and the definition of the transition (T-NOT-high) and (T-EXIT).

(4) $V_f = V \cup \mathcal{LA}(S_l)$.
The local conclusion (2) and the definition of the transition function imply $n_3 > 0$. Using the inductive hypothesis on the evaluation of **while** e **do** S_l **done** in the local conclusion (1), it is possible to show that $V_f = V_2 \cup \mathcal{LA}(S_l)$ because $\mathcal{LA}($**while** e **do** S_l **done**$) = \mathcal{LA}(S_l)$. Then, the local conclusion (3) implies the desired result.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the local conclusion (4) and the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}($**while** e **do** S_l **done**$) = \mathcal{LA}(S_l)$.

($\mathbf{E}_{M(s)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) $S =$ "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_f$

It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-WHILE$_{\texttt{false}}$).

Lets define $q_i = (V_i, n_i)$ for all integer i from 1 to 2.

(2) $n_1 > 0$.
It follows directly from the global hypothesis $n > 0$ and the definition of the transition function of the monitoring automaton.

(3) $V_f = V \cup \mathcal{LA}(S_l)$.
It follows from the local conclusion (2) and the definition of the transition function of the monitoring automaton.

(•) $(q, \texttt{not}\ S) \rightarrow q_f$.
It follows directly from the local conclusion (3) and the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}($**while** e **do** S_l **done**$) = \mathcal{LA}(S_l)$.

($\mathbf{E}_{M(s)}$-SEQ) then we can conclude that :

(1) $S =$ "$S_1\ ;\ S_2$" and:

- $(q,\ \sigma) \vdash S_1 \overset{o_1}{\Longrightarrow}_{M(s)} (q_1,\ \sigma_1)$
- $(q_1,\ \sigma_1) \vdash S_2 \overset{o_2}{\Longrightarrow}_{M(s)} (q_f,\ \sigma_f)$

It follows directly from the definition of the rule ($\mathrm{E}_{M(s)}$-SEQ).

(2) $(q, \texttt{not}\ S_1) \to q_1$ and $(q_1, \texttt{not}\ S_2) \to q_f$.
Those results follow from the inductive hypothesis. The second result also makes use of lemma B.1 in order to prove that $n_1 > 0$.

(3) $q_f = (V \cup \mathcal{LA}(S_1) \cup \mathcal{LA}(S_2), n)$.
It follows directly from the local conclusion (2), the global hypothesis $n > 0$, lemma B.1, and the definition of the rule (T-NOT-high).

(•) $(q, \texttt{not}\ S) \to q_f$.
It follows directly from the transition function of the monitoring automaton because $n > 0$ and $\mathcal{LA}(S_1\ ;\ S_2) = \mathcal{LA}(S_1) \cup \mathcal{LA}(S_2)$.

□

Lemma B.4 ($\mathcal{LA}$ is an over-approximation of the assigned variables)**.** *For all statement S, automaton state $q = (V, n)$, and value store σ, if "$(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{M(O)} (q_f, \sigma_f)$" then:*

- $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$
- *Let* $(V_f, n_f) = q_f$ *in* $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$

Proof. The proof goes by induction on the derivation tree of "$(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{M(O)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{M(O)}$-OK) then we can conclude that :

(1) $S =$ "`skip`" or $S =$ "y := e" or $S =$ "`output` e", "$(q, S) \xrightarrow{OK} q_f$", and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma_f$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-OK) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{M(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

Case 1: $S =$ "`skip`" or $S =$ "`output` e"

(a) $q_f = q$ and $\sigma_f = \sigma$.
It follows directly from the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusion (a).

Case 2: $S =$ "x := e"

(a) $q_f = (V \cup \{x\}, n)$ or $q_f = (V - \{x\}, n)$, and $\sigma_f = \sigma[x \mapsto \sigma(e)]$.
It follows directly from the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusion (a) because $\mathcal{LA}(S) = \{x\}$.

($\mathbf{E}_{M(O)}$-EDIT) then we can conclude that :

(1) $S =$ "`output` e", "$(q, S) \xrightarrow{\texttt{output } \theta} q_f$", and "$\sigma \vdash \texttt{output } \theta \overset{o}{\Rightarrow}_O \sigma_f$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-EDIT), the grammar of the language, and the definition of the transition function of the monitoring automaton.

(2) $q_f = q$ and $\sigma_f = \sigma$.
It follows directly from the local conclusion (1), the transition function of the monitoring automaton and the definition of the semantics (E_O).

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusion (2).

($\mathbf{E}_{M(O)}$-NO) then we can conclude that :

(1) $q_f = q$ and $\sigma_f = \sigma$.
It follows directly from the definition of ($\mathrm{E}_{M(O)}$-NO), and the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusion (1).

($\mathbf{E}_{M(O)}$-IF) then we can conclude that :

(1) $S =$ "**if** e **then** $S_{\texttt{true}}$ **else** $S_{\texttt{false}}$ **end**", $\sigma(e) = v$, and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Rightarrow}_{M(O)} (q_2,\ \sigma_f)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-IF).

(2) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$.
It follows from the inductive hypothesis used on the derivation tree of S_v in the local conclusion (1) and the fact that $\mathcal{LA}(\textbf{if } e \textbf{ then } S_{\texttt{true}} \textbf{ else } S_{\texttt{false}} \textbf{ end}) = \mathcal{LA}(S_{\texttt{true}}) \cup \mathcal{LA}(S_{\texttt{false}})$.

Lets define $q_i = (V_i, n_i)$ for all integer i from 1 to 3.

(3) $V_1 = V$.
It follows directly from the definition of the transition function of the monitoring automaton.

(4) $V - \mathcal{LA}(S_v) \subseteq V_2 \subseteq V \cup \mathcal{LA}(S_v)$.
It follows from the inductive hypothesis applied to the derivation tree of S_v.

(5) $V2 \subseteq V_f \subseteq V_2 \cup \mathcal{LA}(S_{\neg v})$.
It follows from the local conclusion (1) and the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows from the fact that $\mathcal{LA}(\textbf{if } e \textbf{ then } S_{\texttt{true}} \textbf{ else } S_{\texttt{false}} \textbf{ end}) = \mathcal{LA}(S_{\texttt{true}}) \cup \mathcal{LA}(S_{\texttt{false}})$ and the local conclusions (2), (4), and (5).

($\mathbf{E}_{M(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) $S =$ "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow}_{M(O)} (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash \textbf{while } e \textbf{ do } S_l \textbf{ done} \overset{o_w}{\Longrightarrow}_{M(O)} (q_f,\ \sigma_f)$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{true}}$).

(2) $V_1 = V$.
It follows directly from the definition of the transition function of the monitoring automaton.

(3) $\forall x \notin \mathcal{LA}(S_l) : \sigma_1(x) = \sigma(x)$ and $V - \mathcal{LA}(S_l) \subseteq V_2 \subseteq V_2 \cup \mathcal{LA}(S_l)$.
It follows from the inductive hypothesis applied to the derivation tree of S_l found in the local conclusion (1).

(4) $V_3 = V_2$.
It follows directly from the definition of (T-EXIT).

(5) $\forall x \notin \mathcal{LA}(S_l) : \sigma_f(x) = \sigma_1(x)$ and $V_3 - \mathcal{LA}(S_l) \subseteq V_f \subseteq V_3 \cup \mathcal{LA}(S_l)$.
It follows from the inductive hypothesis applied to the derivation tree of **while** e **do** S_l **done** found in the local conclusion (1) because $\mathcal{LA}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathcal{LA}(S_l)$.

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows from the local conclusions (3), (4), and (5) because $\mathcal{LA}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathcal{LA}(S_l)$.

($\mathbf{E}_{M(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) $S =$ "**while** e **do** S_l **done**" and:

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_f$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$).

(2) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$.
It follows directly from the definition of ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$).

(3) $V_f = V$ or $V_f = V \cup \mathcal{LA}(S_l)$.
It follows directly from the definition of the transition function of the monitoring automaton.

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusions (2) and (3) because $\mathcal{LA}(\textbf{while } e \textbf{ do } S_l \textbf{ done}) = \mathcal{LA}(S_l)$.

($\mathbf{E}_{M(O)}$-SEQ) then we can conclude that :

(1) $S =$ "S_1 ; S_2" and:
- $(q,\ \sigma) \vdash S_1 \overset{o_1}{\Longrightarrow}_{M(s)} (q_1,\ \sigma_1)$
- $(q_1,\ \sigma_1) \vdash S_2 \overset{o_2}{\Longrightarrow}_{M(s)} (q_f,\ \sigma_f)$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-SEQ).

(2)
- $\forall x \notin \mathcal{LA}(S_1) : \sigma_1(x) = \sigma(x)$
- $V - \mathcal{LA}(S_1) \subseteq V_1 \subseteq V \cup \mathcal{LA}(S_1)$
- $\forall x \notin \mathcal{LA}(S_2) : \sigma_f(x) = \sigma_1(x)$
- $V_1 - \mathcal{LA}(S_2) \subseteq V_f \subseteq V_1 \cup \mathcal{LA}(S_2)$

Those results follow from the inductive hypothesis.

(•) $\forall x \notin \mathcal{LA}(S) : \sigma_f(x) = \sigma(x)$ and $V - \mathcal{LA}(S) \subseteq V_f \subseteq V \cup \mathcal{LA}(S)$.
It follows directly from the local conclusion (2) because $\mathcal{LA}(S_1 \ ; \ S_2) = \mathcal{LA}(S_1) \cup \mathcal{LA}(S_2)$.

□

Lemma B.5 (WDKL (while-dilemma killer lemma))**.** *For all statement S, expression e, automaton state $q = (V, n)$ such that $n \geq 0$, and value store σ, if "$\mathcal{FV}(e) \cap V \neq \emptyset$" and "$(q, \sigma) \vdash$ **while** e **do** S **done** $\overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$" then:*

- $o = \epsilon$
- *Let* $(V_f, n_f) = q_f$ *in* $V \subseteq V_f$

Proof. The proof goes by induction on the derivation tree of "$(q, \sigma) \vdash$ **while** e **do** S **done** $\overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$". The last rule used by the derivation tree is either ($\mathrm{E}_{M(O)}$-$\mathrm{WHILE}_{\mathtt{true}}$) or ($\mathrm{E}_{M(O)}$-$\mathrm{WHILE}_{\mathtt{false}}$). Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{M(O)}$-$\mathbf{WHILE}_{\mathtt{true}}$) then we can conclude that :

(1) $S =$ **while** e **do** S_l **done** and:
- $\sigma(e) = \mathtt{true}$
- $(q,\ \mathtt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow} M(O)\ (q_2,\ \sigma_1)$
- $(q_2,\ \mathtt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow} M(O)\ (q_f,\ \sigma_f)$
- $o = o_l\ o_w$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-$\mathrm{WHILE}_{\mathtt{true}}$).

(2) $n_1 > 0$.
It follows directly from the definition of the transition function of the monitoring automaton and the global hypotheses $n \geq 0$ and $\mathcal{FV}(e) \cap V \neq \emptyset$.

(3) $o_l = \epsilon$.
It follows from the local conclusions (1) and (2), and from lemma B.2.

(4) $n_3 > 0$.
It follows from the local conclusions (1) and (2), lemma B.1, and the definition of the transition function of the monitoring automaton.

(5) $o_w = \epsilon$.
It follows from the local conclusions (1) and (4), and from lemma B.2.

(6) $V \subseteq V_3$.
From lemma B.3 and the local conclusion (1), $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$. Then, the desired result follows from the definition of the transition function of the monitoring automaton.

(7) $V_3 \subseteq V_f$.
It follows directly from the inductive hypothesis which can be applied because of the local conclusion (1) and the fact that the global hypothesis $\mathcal{FV}(e) \cap V \neq \emptyset$ still holds.

(•) $o = \epsilon$ and $V \subseteq V_f$.
It follows directly from the local conclusions (1), (3), (5), (6) and (7).

($\mathbf{E}_{M(O)}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) $S =$ **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{false}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$).

(•) $o = \epsilon$ and $V \subseteq V_f$.
The fact that $o = \epsilon$ follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$). The fact that $V \subseteq V_f$ follows directly from the local conclusion (1) and the definition of the transition function of the monitoring automaton.

□

Lemma B.6 (Correctness)**.** *For all statement S, automaton state $q = (V, n)$, value stores σ and σ' such that:*

$\star_1$ $(q, \sigma) \vdash S \overset{o}{\Rightarrow}_{M(s)} (q_f, \sigma_f)$,

$\star_2$ $(q, \sigma') \vdash S \overset{o'}{\Longrightarrow}_{M(s)} (q'_f, \sigma'_f)$,

$\star_3$ $\forall x \notin V : \sigma(x) = \sigma'(x)$,

$\star_4$ $n \geq 0$

it is true that, there exist a variable set V_f and an integer n_f such that:

- $o = o'$, $q_f = q'_f = (V_f, n_f)$, *and* $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

Proof. The proof goes by induction on the derivation tree of "$((V, n), \sigma) \vdash S \overset{o}{\Longrightarrow}_{M(s)} (q_f, \sigma_f)$". Assume the lemma holds for any sub-derivation tree, if the last rule used is:

($\mathbf{E}_{M(O)}$-OK) then we can conclude that :

(1) S = `skip` or S = y := e or S = `output` e, "$(q, S) \xrightarrow{OK} q'$" and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma'$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-OK) and the grammar of the language. It can also be deduced from the rule ($\mathrm{E}_{M(O)}$-OK), the transition function of the monitoring automaton, and the semantics (E_O).

Case 1: S = `skip`

(a) $o = \epsilon$, $q_f = q$, and $\sigma_f = \sigma$
It follows from the case hypothesis, the facts that "$(q, S) \xrightarrow{OK} q'$" and "$\sigma \vdash S \overset{o}{\Rightarrow}_O \sigma'$" (in the local conclusion (1)), the definition of the only transition on `skip` for the monitoring automaton (T-SKIP), and the definition of the only rule applying to `skip` in (E_O) (E_O-SKIP).

(b) $o' = \epsilon$, $q'_f = q$, and $\sigma'_f = \sigma'$
It follows from the global hypothesis $\star_2$ and the reasons invoked for the local conclusion (a).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (a) and (b).

Case 2: S = y := e

Sub-case 2.a: $n > 0$ or $\mathcal{FV}(e) \cap V_f \neq \emptyset$

(α) $o = \epsilon$, $q_f = (V \cup \{y\}, n)$, and $\sigma_f = \sigma[y \mapsto \sigma(e)]$
It follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{M(O)}$) and (E_O).

(β) $o' = \epsilon$, $q'_f = (V \cup \{y\}, n)$, and $\sigma'_f = \sigma'[y \mapsto \sigma'(e)]$
It follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{M(O)}$) and (E_O).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (α) and (β).

Sub-case 2.b: $n \leq 0$ and $\mathcal{FV}(e) \cap V_f = \emptyset$

(α) $o = \epsilon$, $q_f = (V - \{y\}, n)$, and $\sigma_f = \sigma[y \mapsto \sigma(e)]$
It follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{M(O)}$) and (E_O).

(β) $o' = \epsilon$, $q'_f = (V - \{y\}, n)$, and $\sigma'_f = \sigma'[y \mapsto \sigma'(e)]$
It follows from the case hypothesis, the sub-case hypothesis, the definition (T) of the transition function of the monitoring automaton, and the definitions of the semantics ($E_{M(O)}$) and (E_O).

(γ) $\sigma(e) = \sigma'(e)$
It follows from the fact $\mathcal{FV}(e) \cap V_f = \emptyset$ (from the sub-case hypothesis) and the global hypothesis $\star_3$.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (α), (β), and (γ).

Case 2: S = `output` e

(a) $n \leq 0$ and $\mathcal{FV}(e) \cap V_f = \emptyset$
It follows from the definition of the rule ($\text{E}_{M(O)}$-OK), the case hypothesis, and the definition (T) of the transition function of the monitoring automaton.

(b) $o = \sigma(e)$, $q_f = q$, and $\sigma_f = \sigma$
It follows from the case hypothesis, the facts that "$(q, S) \xrightarrow{OK} q'$" and the only such transition for `output` e (T-PRINT-ok), and the definition of the semantics (E_O).

(c) $o' = \sigma(e)$, $q'_f = q$, and $\sigma'_f = \sigma$
It follows from the case hypothesis, the local conclusion (a), the definition (T) of the transition function of the monitoring automaton, and the definition of the semantics ($\text{E}_{M(O)}$) and (E_O).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (b) and (c).

($\text{E}_{M(O)}$-EDIT) then we can conclude that :

(1) "$(q, S) \xrightarrow{A'} q'$" and "$\sigma \vdash A' \stackrel{o}{\Rightarrow}_O \sigma'$".
It follows directly from the definition of the rule ($\text{E}_{M(O)}$-EDIT).

(2) $S =$ `output` e, $A' =$ `output` θ, $n \leq 0$, and $\mathcal{FV}(e) \cap V \neq \emptyset$.
It follows directly from the only transition outputting an evaluable transition (T-PRINT-def).

(3) $o = \sigma(\theta)$, $q_f = q$, and $\sigma_f = \sigma$
It follows from the local conclusion (2), the definition of the transition function of the monitoring automaton (T), and the definition of the semantics (E_O).

(4) $o' = \sigma(\theta)$, $q'_f = q$, and $\sigma'_f = \sigma$
It follows from the fact that $n \leq 0$ and $\mathcal{FV}(e) \cap V \neq \emptyset$ (from the local conclusion (2)), the definition of the semantics ($\mathrm{E}_{M(O)}$) for actions, the definition of the only transition of the monitoring automaton (T) for print actions whenever $n \leq 0$ and $\mathcal{FV}(e) \cap V \neq \emptyset$, and the definition of the semantics (E_O).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (3) and (4).

($\mathbf{E}_{M(O)}$-NO) then we can conclude that :

(1) "$(q, S) \xrightarrow{NO} q'$".
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-NO).

(2) $S = \texttt{output}\ e$, and $n > 0$.
It follows directly from the only transition outputting "NO" (T-PRINT-no).

(3) $o = \epsilon$, $q_f = q$, and $\sigma_f = \sigma$
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-NO).

(4) $o' = \epsilon$, $q'_f = q$, and $\sigma'_f = \sigma$
It follows from the fact that $n > 0$ (from the local conclusion (2)), the definition of the only transition of the monitoring automaton (T) for print actions in such a case, and the definition of the semantics ($\mathrm{E}_{M(O)}$) for actions whenever the automaton outputs "NO".

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis $\star_3$ and the local conclusions (3) and (4).

($\mathbf{E}_{M(O)}$-IF) then we can conclude that :

(1) $S = \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}$ and:

- $\sigma(\mathrm{e}) = \mathrm{v}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_v \overset{o}{\Rightarrow}_{M(s)} (q_2,\ \sigma_f)$
- $(q_2,\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3$
- $(q_3,\ \texttt{exit}) \xrightarrow{OK} q_f$

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
Case 1: $\sigma'(e) = v$

(a) • $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$

• $(q_1,\ \sigma') \vdash S_v \overset{o'}{\Longrightarrow}_{M(s)} (q_2',\ \sigma_f')$

• $(q_2',\ \texttt{not}\ S_{\neg v}) \xrightarrow{OK} q_3'$

• $(q_3',\ \texttt{exit}) \xrightarrow{OK} q_f'$

It follows from the case hypothesis, the local conclusion (1), the definition of the only rule applying to if-statements ($\text{E}_{M(O)}$-IF), and the definition of the transition rules (T-BRANCH-high) and (T-BRANCH-low) (Both evaluations use the same rule as the conditions for those depend only e and V).

(b) $o = o'$, $q_2 = q_2' = (V_2, n_2)$, and $\forall x \notin V_2 : \sigma_f(x) = \sigma_f'(x)$.
This result is obtained by applying the inductive hypothesis to the derivations of S_v found in the local conclusions (1) and (a).

(c) If $n_2 > 0$ then $q_f = (V_2 \cup \mathcal{LA}(S_{\neg v}), \lfloor n_2/2 \rfloor) = q_f'$ else $q_f = (V_2, \lfloor n_2/2 \rfloor) = q_f'$.
It follows directly from the definition of transition function of the monitoring automaton and the local conclusions (1), (a) and (b).

(•) $o = o'$, $q_f = q_f' = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma_f'(x)$
It follows directly from the local conclusions (b) and (c) because all x, which does not belong to V_f, does not belong to V_2.

Case 2: $\sigma'(e) = \neg v$

(a) $\mathcal{FV}(e) \cap V \neq \emptyset$.
The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b) • $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$

• $(q_1,\ \sigma') \vdash S_{\neg v} \overset{o'}{\Longrightarrow}_{M(s)} (q_2',\ \sigma_f')$

• $(q_2',\ \texttt{not}\ S_v) \xrightarrow{OK} q_3'$

• $(q_3',\ \texttt{exit}) \xrightarrow{OK} q_f'$

It follows from the case hypothesis, the local conclusion (1), the definition of the only rule applying to if-statements ($\text{E}_{M(O)}$-IF), and the definition of the transition rule (T-BRANCH-high).

(c) $o = o' = \epsilon$.
Let $q_1 = (V_1, n_1)$. The local conclusion (1), the global hypothesis $\star_4$, and the definition of (T-BRANCH-high) imply that n_1 is greater than 0. This and lemma B.2 imply the above result.

Let q_a and $q_b = (V_b, n_b)$ be monitoring automaton states such that "$(q_1, \texttt{not}S_v) \xrightarrow{OK} q_a$" and "$(q_a, \texttt{not}S_{\neg v}) \xrightarrow{OK} q_b$".

(d) There exists q_c such that "$(q_1, \texttt{not}S_{\neg v}) \xrightarrow{OK} q_c$" and "$(q_c, \texttt{not}S_v) \xrightarrow{OK} q_b$".
It is obvious from the definition of (T-NOT-high) and (T-NOT-low).

(e) $q_3 = q_b = q'_3$.
It follows from lemma B.3 and the local conclusions (1), (b), and (d).

(f) $q_f = q'_f$.
It follows directly from the definition of (T-EXIT) and the local conclusions (e), (1), and (b).

(g) $V_b = V \cup \mathcal{LA}(S_v) \cup \mathcal{LA}(S_{\neg v})$.
In the proof of local conclusion (c), we proved that n_1 is greater than 0. Then the above result follows directly from the definition of (T-NOT-high).

(h) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
From the definition of (T-EXIT) and the local conclusions (1), (e), and (g), $V_f = V \cup \mathcal{LA}(S_v) \cup \mathcal{LA}(S_{\neg v})$. From the evaluation of S_v (in the local conclusion (1)) and lemma B.4, if x does not belongs to V_f, and so does not belongs to $\mathcal{LA}(S_v)$, $\sigma_f(x) = \sigma(x)$. From the evaluation of $S_{\neg v}$ (in the local conclusion (b)) and lemma B.4, if x does not belongs to V_f, and so does not belongs to $\mathcal{LA}(S_{\neg v})$, $\sigma'_f(x) = \sigma'(x)$. Additionally, if x does not belongs to V_f then it does not belongs to V; which implies that $\sigma(x) = \sigma'(x)$ because of the global hypothesis $\star_3$. Finally, those three equalities imply that if x does not belongs to V_f then $\sigma_f(x) = \sigma'_f(x)$.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows directly from the local conclusions (c), (f) and (h).

($\mathbf{E}_{M(O)}$-WHILE$_{\texttt{true}}$) then we can conclude that :

(1) $S =$ **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{true}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma) \vdash S_l \overset{o_l}{\Longrightarrow} M(O)\ (q_2,\ \sigma_1)$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$
- $(q_3,\ \sigma_1) \vdash$ **while** e **do** S_l **done** $\overset{o_w}{\Longrightarrow} M(O)\ (q_f,\ \sigma_f)$
- $o = o_l\ o_w$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{true}}$).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

Case 1: $\sigma'(e) = \texttt{true}$

(a) • $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$

• $(q_1,\ \sigma') \vdash S_l \overset{o'_l}{\Longrightarrow} M(O)\ (q'_2,\ \sigma'_1)$

• $(q'_2,\ \texttt{exit}) \xrightarrow{OK} q'_3$

• $(q'_3,\ \sigma'_1) \vdash \textbf{while}\ e\ \textbf{do}\ S_l\ \textbf{done} \overset{o'_w}{\Longrightarrow} M(O)\ (q'_f,\ \sigma'_f)$

• $o' = o'_l\ o'_w$

It follows directly from the local conclusion (1), the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\text{E}_{M(O)}$-WHILE$_{\texttt{true}}$), and the definition of the transition function of the monitoring automaton.

(b) $o_l = o'_l$, $q_2 = q'_2 = (V_2, n_2)$, $\forall x \notin V_2 : \sigma_1(x) = \sigma'_1(x)$, and $n_2 = n$.

This result is obtained by applying the inductive hypothesis to the derivations of S_l found in the local conclusions (1) and (a).

(c) $q_3 = (V_2, \lfloor n_2/2 \rfloor) = q'_3$.

It follows directly from the definition of transition function of the monitoring automaton and the local conclusions (1), (a) and (b).

(d) $o_w = o'_w$, $q_f = q'_f$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

This result is obtained by applying the inductive hypothesis to the derivations of **while** e **do** S_l **done** found in the local conclusions (1) and (a). It is possible to apply the inductive hypothesis because of the local conclusions (c) and (b).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

It follows directly from the local conclusions (1), (a), (b), and (d).

Case 2: $\sigma'(e) = \texttt{false}$

(a) $\mathcal{FV}(e) \cap V \neq \emptyset$.

The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b) $n_1 \geq 0$.

The local conclusions (a) and (1), the global hypothesis $\star_4$, and the definition of (T-BRANCH-high) imply that n_1 is greater than 0.

(c) $o_l = \epsilon$.

It follows from the local conclusions (1) and (b), and from lemma B.2.

(d) • $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$

• $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q'_2$

• $(q'_2,\ \texttt{exit}) \xrightarrow{OK} q'_f$
It follows directly from the local conclusion (1), the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\text{E}_{M(O)}$-$\text{WHILE}_{\texttt{false}}$), and the definition of the transition function of the monitoring automaton.

(e) $o' = \epsilon$ and $\sigma'_f = \sigma'$.
It follows directly from the definition of the rule ($\text{E}_{M(O)}$-$\text{WHILE}_{\texttt{false}}$).

(f) $q_3 = (V_3, n_3) = q'_f$.
It follows from the local conclusions (1) and (d), lemma B.3, and the definition of the transition function of the monitoring automaton for the input `exit`..

(g) $V_3 = V \cup \mathcal{LA}(S_l)$.
It follows from the local conclusions (f), (d), and (b), and the definition of the transition function of the monitoring automaton.

(h) $o_w = \epsilon$ and $V_3 \subseteq V_f$.
It follows from the local conclusions (1), (a) and (g), and from lemma B.5.

(i) $\forall x \notin V_f : \sigma_f(x) = \sigma_1(x)$ and $V_f = V_3$.
Both results follow from lemma B.4, the fact that "$\mathcal{LA}(\textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}) = \mathcal{LA}(S)$". For the first result, from the local conclusion (g) and (h), any variable x, which does not belong to V_f, does not belong to $\mathcal{LA}(S_l)$. For the second result, the local conclusions (g) imply that $V_3 = V_3 \cup \mathcal{LA}(S_l)$. This result combine with the local conclusion (h) and the conclusions of lemma B.4 imply that $V_f = V_3$.

(j) $\forall x \notin V_f : \sigma_1(x) = \sigma'_f(x)$.
For all variable x, if x does not belong to V_f then the local conclusions (i) and (g) imply that x does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. For all variable x, if x does not belong to V_f then the local conclusions (i) and (g) imply that x does not belong to $\mathcal{LA}(S_l)$. Which, in turn, combined with the local conclusion (1) and lemma B.4, implies that $\sigma_1(x) = \sigma(x)$. Those two equalities combined with the local conclusion (e) imply the desired result.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.
From the local conclusion (1), $o = o_l\ o_w$. So, it follows from the local conclusions (c), (h), and (e) that $o' = o$. From the local conclusion (f), $q_3 = q'_f$. As done at the beginning of the proof of this lemma, it can be easily proved that $n_f = n_3$. This result combined with the local conclusions (i) and (f)

imply that $q_f = q_3 = q'_f$. Finally, from the local conclusions (i) and (j) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

($\mathbf{E_{M(O)}}$-WHILE$_{\texttt{false}}$) then we can conclude that :

(1) $S =$ **while** e **do** S_l **done** and:

- $\sigma(e) = \texttt{false}$
- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \texttt{not}\ S_l) \xrightarrow{OK} q_2$
- $(q_2,\ \texttt{exit}) \xrightarrow{OK} q_3$

It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$).

(2) $o = \epsilon$, $q_f = q_3$, $\sigma_f = \sigma$, and $V \subseteq V_f$.
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$) and the definition of the transition function of the monitoring automaton.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$

Case 1: $\sigma'(e) = \texttt{false}$

(a) $o' = \epsilon$, $q'_f = q_3$, $\sigma'_f = \sigma'$.
It follows from the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\mathrm{E}_{M(O)}$-WHILE$_{\texttt{false}}$), and the definition of the transition function of the monitoring automaton. In this case, the transitions depend only on the initial state q and the expression e.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
The first two equalities follow from the local conclusions (2) and (a). From the local conclusion (2), all variable x, which does not belong to V_f, does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. Then, from the local conclusions (2) and (a), $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

Case 2: $\sigma'(e) = \texttt{true}$

(a) $\mathcal{FV}(e) \cap V \neq \emptyset$.
The negation of this property is in contradiction with the case hypothesis, the local conclusion (1), and the global hypothesis $\star_3$.

(b) • $\sigma(e) = \texttt{true}$

- $(q,\ \texttt{branch}\ e) \xrightarrow{OK} q_1$
- $(q_1,\ \sigma') \vdash S_l \overset{o'_l}{\Longrightarrow}_{M(O)} (q'_2,\ \sigma'_1)$
- $(q'_2,\ \texttt{exit}) \xrightarrow{OK} q'_3$
- $(q'_3,\ \sigma'_1) \vdash$ **while** e **do** S_l **done** $\overset{o'_w}{\Longrightarrow}_{M(O)} (q'_4,\ \sigma'_f)$

- $o' = o'_l \; o'_w$

  It follows from the global hypothesis $\star_2$, the case hypothesis, the definition of the only rule applying to this evaluation ($\text{E}_{M(O)}$-$\text{WHILE}_{\texttt{true}}$), the local conclusion (a), and the definition of the transition function of the monitoring automaton.

Let $q_1 = (V_1, n_1)$, $q'_3 = (V'_3, n'_3)$, and $q'_f = (V'_f, n'_f)$.

(c) $n_1 \geq 0$.

The local conclusions (a) and (b), the global hypothesis $\star_4$, and the definition of (T-BRANCH-high) imply that n_1 is greater than 0.

(d) $o'_l = \epsilon$.

It follows from the local conclusions (b) and (c), and from lemma B.2.

(e) $q'_3 = q_3 = (V_f, n_f)$.

It follows from the local conclusions (1) and (b), lemma B.3, and the definition of the rule ($\text{E}_{M(O)}$-$\text{WHILE}_{\texttt{true}}$).

(f) $V_f = V \cup \mathcal{LA}(S_l)$.

It follows from the local conclusions (c), (1), and (2), and the definition of the transition function of the monitoring automaton.

(g) $\forall x \notin V_f : \sigma_f(x) = \sigma'_1(x)$.

For all variable x, if x does not belong to V_f then the local conclusion (f) implies that x does not belong to V. And so, the global hypothesis $\star_3$ implies that $\sigma(x) = \sigma'(x)$. For all variable x, if x does not belong to V_f then the local conclusion (f) implies that x does not belong to $\mathcal{LA}(S_l)$. Which, in turn, combined with the local conclusion (b) and lemma B.4, implies that $\sigma'_f(x) = \sigma'(x)$. Those two equalities combined with the local conclusion (2) imply the desired result.

(h) $o'_w = \epsilon$ and $V'_3 \subseteq V'_f$.

It follows from the local conclusions (b) and (a), and from lemma B.5.

(i) $\forall x \notin V_f : \sigma'_f(x) = \sigma'_1(x)$ and $V'_f = V'_3$.

Both results follow from lemma B.4, the fact that "$\mathcal{LA}(\textbf{while } e \textbf{ do } S \textbf{ done}) = \mathcal{LA}(S)$". For the first result, from the local conclusion (f), any variable x, which does not belong to V_f, does not belong to $\mathcal{LA}(S_l)$. For the second result, the local conclusions (e) and (f) imply that $V'_3 = V'_3 \cup \mathcal{LA}(S_l)$. This result combine with the local conclusion (h) and the conclusions of lemma B.4 imply that $V'_f = V'_3$.

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

From the local conclusion (b), $o' = o'_l \; o'_w$. So, it follows from the local conclusions (2), (d), and (h) that $o' = o$. From the local conclusion (e), $q_f = q'_3$. As done at the beginning of the

proof of this lemma, it can be easily proved that $n'_f = n'_3$. This result combined with the local conclusion (i) implies that $q'_f = q'_3 = q_f$. Finally, from the local conclusions (i) and (g) $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$.

($\mathbf{E}_{M(O)}$-SEQ) then we can conclude that :

(1) $S = S_1 \ ; \ S_2$, $(q, \sigma) \vdash S_1 \stackrel{o_1}{\Longrightarrow}_{M(O)} (q_1, \sigma_1)$, $(q_1, \sigma_1) \vdash S_2 \stackrel{o_2}{\Longrightarrow}_{M(O)} (q_f, \sigma_f)$, and $o = o_1 \ o_2$.
It follows directly from the definition of the rule ($\mathrm{E}_{M(O)}$-SEQ).

(2) $(q, \sigma') \vdash S_1 \stackrel{o'_1}{\Longrightarrow}_{M(O)} (q'_1, \sigma'_1)$, $(q'_1, \sigma'_1) \vdash S_2 \stackrel{o'_2}{\Longrightarrow}_{M(O)} (q'_f, \sigma'_f)$, and $o' = o'_1 \ o'_2$.
It follows from the global hypothesis $\star_2$, the local conclusion (2), and the definition of the rule ($\mathrm{E}_{M(O)}$-SEQ).

(3) $o_1 = o'_1$, $q_1 = q'_1 = (V_1, n_1)$, and $\forall x \notin V_1 : \sigma_1(x) = \sigma'_1(x)$
This result can be obtained from the inductive hypothesis using the evaluations of S_1 in the local conclusions (1) and (2).

(4) $o_2 = o'_2$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
This result can be obtained from the inductive hypothesis using the evaluations of S_2 in the local conclusions (1) and (2), and the fact that $\forall x \notin V_1 : \sigma_1(x) = \sigma'_1(x)$ (from the local conclusion (3)).

(•) $o = o'$, $q_f = q'_f = (V_f, n_f)$, and $\forall x \notin V_f : \sigma_f(x) = \sigma'_f(x)$
It follows from the global hypothesis the local conclusions (1), (2), (3), and (4).

□

B.2 Proofs of Sect. 4.2

Lemma B.7 (Simple Security)**.** *For all typing environment γ, expression e, and command type τ cmd, if $\gamma \vdash e : \tau$ then, for all variable x belonging to $\mathcal{FV}(e)$, $\gamma(x) \leq \tau$.*

Proof. This lemma is just a reformulation of the (Simple Security) lemma appearing in [VSI96]. □

Lemma B.8 (Confinement)**.** *For all typing environment γ, command C, and command type τ cmd, if $\gamma \vdash C : \tau$ cmd then, for all variable x belonging to $\mathcal{LA}(C)$, $\tau \leq \gamma(x)$.*

Proof. This lemma is just a reformulation of the (Confinement) lemma appearing in [VSI96]. □

Lemma B.9 (Confined Outputs)**.** *For all command C, typing environment γ, and value stores σ and σ' if:*

$\star_1$ $\gamma \vdash C : H$ *cmd,*

$\star_2$ $\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$,

then $o = \epsilon$.

Proof. The proof goes by contradiction. If $o \neq \epsilon$ then C contains a command `output` e. If it does, because of the typing rules, C must be typed L cmd. This is in contradiction with the hypothesis $\star_1$. □

Lemma B.10 (Helper 1)**.** *For all command C, automaton states (V, n) and (V', n), and value stores σ and σ', if $((V, n), \sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{M(O)} ((V', n), \sigma')$ then, for all n' greater or equal to n, $((V, n'), \sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{M(O)} ((V', n'), \sigma')$.*

Proof. The monitoring mechanism does not influence the final value store obtained after the execution. So, changing the automaton state does not change the final value store. If the command C does not contain print statements, then any execution of C (whatever the automaton state) outputs nothing. If the command C contains a print statement executed, then it implies that n is greater than 0 (otherwise something would be printed). The behavior of the automaton with regard to print statements is the same for any n above 0. So having n' greater than n (itself greater than 0) does not change the behavior of the automaton. So the output is identical. □

Theorem B.11 (Monitoring Automaton is more precise)**.** *For all command*[4] *C, typing environment γ, command type τ cmd, value stores σ and σ', and automaton state (V, n), if:*

$\star_1$ $\forall x \in V, \gamma(x) = H$,

$\star_2$ $n > 0 \Rightarrow \tau = H$,

$\star_3$ $\gamma \vdash C : \tau$ *cmd*,

$\star_4$ $\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$,

then there exists an automaton state (V', n) such that:

- $((V, n), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$,
- $\forall x \in V', \gamma(x) = H$.

Proof. The proof is done by induction on the size of the derivation tree of "$\sigma \vdash C \overset{o}{\Rightarrow}_O \sigma'$". Assume the theorem holds for any sub-derivation tree. If the last semantics rule used is:

($\mathbf{E}_O$-ASSIGN) then we can conclude that :

(1)
- C is "$x := e$",
- $\sigma \vdash C \overset{\epsilon}{\Rightarrow}_O \sigma[x \mapsto \sigma(e)]$.

It follows directly from the rule (E_O-ASSIGN).

[4] We use "command" and "statements" as synonyms

(2) $((V,n),\sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{M(O)} ((V',n),\sigma[x \mapsto \sigma(e)])$.
It follows directly from the local conclusion (1) and the rules (T-ASSIGN-sec), (T-ASSIGN-pub), ($\text{E}_{M(O)}$-OK), and (E_O-ASSIGN).

(3) $\forall y \in V', \gamma(y) = H$.

Case 1: $n = 0$ and $\mathcal{FV}(e) \cap V = \emptyset$.

(a) $V' \subseteq V$.
It follows directly from the rule (T-ASSIGN-pub).

(○) $\forall y \in V', \gamma(y) = H$.
It follows from the local conclusion (a) and the global hypothesis $\star_1$.

Case 2: $n > 0$ or $\mathcal{FV}(e) \cap V \neq \emptyset$.

(a) $V' = V \cup \{x\}$.
It follows directly from the rule (T-ASSIGN-sec).

(b) $\gamma(x) = H$.
If $n > 0$ then, because of the global hypothesis $\star_2$, $\tau = H$; and so, because of the global hypotheses $\star_3$ and the typing rule (T-ASSIGN) (which is the only one applying to "$x := e$"), $\gamma(x) = H$. If $\mathcal{FV}(e) \cap V \neq \emptyset$ then there exists a variable y in $\mathcal{FV}(e)$ such that $y \in V$. The global hypothesis $\star_1$ implies that $\gamma(y) = H$. Then, the global hypothesis $\star_3$ and lemma B.7 imply that $\tau = H$; and so, because of the typing rule (T-ASSIGN) (which is the only one applying to "$x := e$"), $\gamma(x) = H$.

(○) $\forall y \in V', \gamma(y) = H$.
It follows from the global hypothesis $\star_1$ and the local conclusions (a) and (b).

(•) there exists an automaton state (V',n) such that "$((V,n),\sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$" and "$\forall x \in V', \gamma(x) = H$".
If follows directly from the local conclusions (2) and (3).

(E_O-SKIP) then we can conclude that :

(1)
- C is "`skip`",
- $\sigma \vdash C \overset{\epsilon}{\Rightarrow}_O \sigma$.

It follows directly from the rule (E_O-SKIP).

(2) $((V,n),\sigma) \vdash C \overset{\epsilon}{\Rightarrow}_{M(O)} ((V,n),\sigma)$.
It follows directly from the local conclusion (1) and the rules (T-SKIP), ($\text{E}_{M(O)}$-OK), and (E_O-SKIP).

(•) there exists an automaton state (V',n) such that "$((V,n),\sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$" and "$\forall x \in V', \gamma(x) = H$".
If follows directly from the local conclusion (2) and from the global hypothesis $\star_1$.

(E_O-PRINT) then we can conclude that :

(1) • C is "`output` e",

• $\sigma \vdash C \xrightarrow{\sigma(e)}_O \sigma$.

It follows directly from the rule (E_O-PRINT).

(2) $n = 0$ and $\mathcal{FV}(e) \cap V = \emptyset$.
The global hypothesis $\star_3$, the local conclusion (1) and the typing rule (T-PRINT) (the only one applying to "`output` e") imply that $\tau = L$. Hence, the global hypothesis $\star_2$ implies $n = 0$. The typing rule (T-PRINT) also implies that "$\gamma \vdash e : \tau$". Then lemma B.7 implies that, for all y in $\mathcal{FV}(e)$, $\gamma(y) = L$. Hence, because of the global hypothesis $\star_1$, $\mathcal{FV}(e) \cap V = \emptyset$.

(3) $((V, n), \sigma) \vdash C \xrightarrow{\sigma(e)}_{M(O)} ((V, n), \sigma)$.
It follows from the transition (T-PRINT-ok) (which is the only one applying because of the local conclusions (1) and (2)) and the rules ($E_{M(O)}$-OK) and (E_O-PRINT).

(•) there exists an automaton state (V', n) such that "$((V, n), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".
If follows directly from the local conclusion (3) and from the global hypothesis $\star_1$.

(E_O-IF) then we can conclude that :

(1) • C is "**if** e **then** C_{true} **else** C_{false} **end**",

• $\sigma(e) = v$

• $\sigma \vdash C_v \overset{o}{\Rightarrow}_O \sigma'$.

It follows directly from the rule (E_O-IF).

(2) There exists a type τ' such that:

• $\tau \leq \tau'$,

• $\gamma \vdash e : \tau'$,

• $\gamma \vdash C_{true} : \tau'$ cmd,

• $\gamma \vdash C_{false} : \tau'$ cmd.

It follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" (T-IF).

(•) There exists an automaton state (V', n) such that "$((V, n), \sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".

Case 1: $\mathcal{FV}(e) \cap V \neq \emptyset$.

(a) $\tau' = H$.
From the local conclusion (2), $\gamma \vdash e : \tau'$. From the case hypothesis and the global hypothesis $\star_1$, there exists a variable y in $\mathcal{FV}(e)$ such that $\gamma(y) = H$. Using lemma B.7, those two properties imply that $\tau' = H$.

(b) There exists an automaton state $(V', 2n+1)$ such that "$((V, 2n+1), \sigma) \vdash C_v \stackrel{o}{\Rightarrow}_{M(O)} ((V', 2n+1), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and the local conclusions (a), (2) and (1).

(c) $((V, n), \sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V' \cup \mathcal{LA}(C_{\neg v}), n), \sigma')$.
It follows from the only rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" ($\mathrm{E}_{M(O)}$-IF), the local conclusion (b), and the transition rules (T-BRANCH-high) (which is the only one applying to `branch`e because of the case hypothesis), (T-NOT-high), and (T-EXIT).

(d) $\forall x \in V' \cup \mathcal{LA}(C_{\neg v}), \gamma(x) = H$.
As, from the local conclusion (a) $\tau' = H$ and from the local conclusion (2) $\gamma \vdash C_{\neg v} : \tau'$ cmd, lemma B.8 implies that, for all x belonging to $\mathcal{LA}(C_{\neg v})$, $\gamma(x) = H$. This result, combined with the local conclusion (b), implies the desired result.

(∘) There exists an automaton state (V', n) such that "$((V, n), \sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the local conclusions (c) and (d).

Case 2: $\mathcal{FV}(e) \cap V = \emptyset$.

(a) There exists an automaton state $(V', 2n)$ such that "$((V, 2n), \sigma) \vdash C_v \stackrel{o}{\Rightarrow}_{M(O)} ((V', 2n), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2) and (1).

(b) $((V, n), \sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V'', n), \sigma')$ with $n > 0 \Rightarrow (V'' = V' \cup \mathcal{LA}(C_{\neg v}))$ and $n = 0 \Rightarrow V'' = V'$.
It follows from the only rule applying to "**if** e **then** C_{true} **else** C_{false} **end**" ($\mathrm{E}_{M(O)}$-IF), the local conclusion (a), and the transition rules (T-BRANCH-low) (which is the only one applying to `branch`e because of the case hypothesis), (T-NOT-high), (T-NOT-low), and (T-EXIT).

(c) $\forall x \in V'', \gamma(x) = H$.
If $n = 0$ then $V'' = V'$ and the desired property follows directly from the global hypothesis $\star_1$. If $n > 0$ then the global hypothesis $\star_2$ implies $\tau = H$. As, from the local conclusion (2) $\tau \leq \tau'$ and $\gamma \vdash C_{\neg v} : \tau'$ cmd, lemma B.8 implies that, for all x belonging to $\mathcal{LA}(C_{\neg v})$, $\gamma(x) = H$. This result, combined with the local conclusions (b) and (a), implies that if $n > 0$ then $\forall x \in V'', \gamma(x) = H$.

(∘) There exists an automaton state (V', n) such that

"$((V,n),\sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$", and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the local conclusions (b) and (c).

($\mathbf{E}_O$-WHILE$_{true}$) then we can conclude that :

(1)
- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{true}$
- $\sigma \vdash C_l \ ;$ **while** e **do** C_l **done** $\overset{o}{\Rightarrow}_O \sigma'$.

It follows directly from the rule (E_O-WHILE$_{true}$).

(2) There exists a type τ' such that:
- $\tau \leq \tau'$,
- $\gamma \vdash e : \tau'$,
- $\gamma \vdash C_l : \tau'$ cmd.

It follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**while** e **do** C_l **done**" (T-WHILE).

(3) $\gamma \vdash C_l \ ;$ **while** e **do** C_l **done** $: \tau'$ cmd.
From the local conclusion (2) and the typing rule (T-WHILE), $\gamma \vdash$ **while** e **do** C_l **done** $: \tau'$ cmd. Hence, as from the local conclusion (2) $\gamma \vdash C_l : \tau'$ cmd, the typing rule (T-SEQ) implies the desired result.

(4) There exists an automaton state (V',n) such that "$((V,n),\sigma) \vdash C_l \ ;$ **while** e **do** C_l **done** $\overset{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$", and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2), (3) and (1).

(5) There exist an automaton state (V_l,n) and a value store σ_l such that:
- $((V,n),\sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{M(O)} ((V_l,n),\sigma_l)$,
- $((V_l,n),\sigma_l) \vdash$ **while** e **do** C_l **done** $\overset{o_w}{\Longrightarrow}_{M(O)} ((V',n),\sigma')$,
- $o = o_l \ o_w$.

It follows from the local conclusion (4), the only semantics rule applying to $C_l \ ;$ **while** e **do** C_l **done** ($\mathrm{E}_{M(O)}$-SEQ), and lemma B.1.

(6) "$((V,n),\sigma) \vdash C \overset{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$".

Case 1: $\tau' = H$.

(a) $o_l = \epsilon$.
It follows from the case hypothesis and lemma B.9 applied to the local conclusions (2) and (5).

(b) For all n' greater than n, $((V,n'),\sigma) \vdash C_l \overset{o_l}{\Longrightarrow}_{M(O)} ((V_l,n'),\sigma_l)$.
It follows from lemma B.10 and the local conclusions (a) and (5).

(○) "$((V,n),\sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$".
It follows from the semantics rule ($\text{E}_{M(O)}$-WHILE_{true}), the transition (T-BRANCH-low) and (T-BRANCH-high), the local conclusion (b), the transition (T-EXIT), the local conclusion (5).

Case 2: $\tau' \neq H$.

(a) $n = 0$ and $\mathcal{FV}(e) \cap V = \emptyset$.
As $\tau \leq \tau'$ (from the local conclusion (2)), the case hypothesis and the global hypothesis $\star_2$ imply $n = 0$. As $\gamma \vdash e : \tau'$ (from the local conclusion (2)), the case hypothesis, lemma B.7, and the global hypothesis $\star_1$ imply $\mathcal{FV}(e) \cap V = \emptyset$.

(b) $((V,2n),\sigma) \vdash C_l \stackrel{o_l}{\Rightarrow}_{M(O)} ((V_l,2n),\sigma_l)$.
It follows from the local conclusion (5) and the fact that $2n = n$ (because of the local conclusion (a)).

(○) "$((V,n),\sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$".
It follows from the semantics rule ($\text{E}_{M(O)}$-WHILE_{true}), the transition (T-BRANCH-low) (which is the only one applying because of the local conclusion (a)), the local conclusion (b), the transition (T-EXIT), the local conclusion (5).

(•) There exists an automaton state (V',n) such that "$((V,n),\sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V',n),\sigma')$", and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the local conclusions (6) and (4).

(E_O-WHILE_{false}) then we can conclude that :

(1)
- C is "**while** e **do** C_l **done**",
- $\sigma(e) = \texttt{false}$
- $o = \epsilon$
- $\sigma' = \sigma$.

It follows directly from the rule (E_O-WHILE_{false}).

(2) There exists a type τ' such that:
- $\tau \leq \tau'$,
- $\gamma \vdash e : \tau'$,
- $\gamma \vdash C_l : \tau'$ cmd.

It follows directly from the local conclusion (1), the global hypothesis $\star_3$ and the only typing rule applying to "**while** e **do** C_l **done**" (T-WHILE).

(3) $((V,n),\sigma) \vdash C \stackrel{\epsilon}{\Rightarrow}_{M(O)} ((V',n),\sigma)$ with $(n > 0 \vee \mathcal{FV}(e) \cap V \neq \emptyset) \Rightarrow (V' = V \cup \mathcal{LA}(C_l))$ and $(n = 0 \wedge \mathcal{FV}(e) \cap V = \emptyset) \Rightarrow V' = V$.
It follows from the only rule applying to "**while** e **do** C_l **done**" whenever $\sigma(e) = \texttt{false}$ ($\text{E}_{M(O)}$-WHILE_{false}) and the transition rules (T-BRANCH-low), (T-BRANCH-high), (T-NOT-high), (T-NOT-low), and (T-EXIT).

(4) $(n > 0 \vee \mathcal{FV}(e) \cap V \neq \emptyset) \Rightarrow (\forall x \in \mathcal{LA}(C_l),\ \gamma(x) = H)$.
If $n > 0$ then the global hypothesis $\star_2$ imply that $\tau = H$. Hence, because of the local conclusion (2), $\tau' = H$. If $\mathcal{FV}(e) \cap V \neq \emptyset$ then, from the global hypothesis $\star_1$, there exists a variable y in $\mathcal{FV}(e)$ such that $\gamma(y) = H$. Using lemma B.7, as $\gamma \vdash e : \tau'$ (from the local conclusion (1)), this implies that $\tau' = H$. As $\tau' = H$ in both cases ($n > 0$ and $\mathcal{FV}(e) \cap V \neq \emptyset$), the local conclusion (2) ($\gamma \vdash C_l : \tau'$ cmd) and lemma B.8 imply the desired result.

(•) There exists an automaton state (V', n) such that "$((V, n), \sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$", and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the local conclusions (3) and (4) and from the global hypothesis $\star_1$.

($\mathbf{E}_O$-SEQ) then we can conclude that :

(1) There exists a value store σ'_1 such that:

- C is "$C_1 \ ; \ C_2$",
- $\sigma \vdash C_1 \stackrel{o_1}{\Longrightarrow}_O \sigma'_1$,
- $\sigma'_1 \vdash C_2 \stackrel{o_2}{\Longrightarrow}_O \sigma'$,
- $o = o_1 \ o_2$.

It follows directly from the rule (E_O-SEQ).

(2)

- $\gamma \vdash C_1 : \tau$ cmd,
- $\gamma \vdash C_2 : \tau$ cmd.

It follows directly from the global hypothesis $\star_3$, the local conclusion (1), and the only typing rule applying to "$C_1 \ ; \ C_2$" (T-SEQ).

(3) there exists an automaton state (V'_1, n) such that "$((V, n), \sigma) \vdash C_1 \stackrel{o_1}{\Longrightarrow}_{M(O)} ((V'_1, n), \sigma'_1)$" and "$\forall x \in V'_1, \gamma(x) = H$".
It follows directly from the inductive hypothesis, the global hypotheses $\star_1$ and $\star_2$, and the local conclusions (2) and (1).

(4) there exists an automaton state (V'_2, n) such that "$((V'_1, n), \sigma'_1) \vdash C_2 \stackrel{o_2}{\Longrightarrow}_{M(O)} ((V'_2, n), \sigma'_2)$", and "$\forall x \in V'_2, \gamma(x) = H$".
It follows directly from the inductive hypothesis, the local conclusion (3), the global hypothesis $\star_2$, and the local conclusions (2) and (1).

(5) $((V, n), \sigma) \vdash C \stackrel{o_1 o_2}{\Longrightarrow}_{M(O)} ((V'_2, n), \sigma'_2)$.
It follows from the rule ($\mathrm{E}_{M(O)}$-SEQ) and the local conclusions (3) and (4).

(•) There exists an automaton state (V', n) such that "$((V, n), \sigma) \vdash C \stackrel{o}{\Rightarrow}_{M(O)} ((V', n), \sigma')$" and "$\forall x \in V', \gamma(x) = H$".
It follows directly from the local conclusions (5) and (4).

□

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core calculus of Dependency. In *Proc. ACM Symp. Principles of Programming Languages*, pages 147–160, January 1999.

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proc. ACM International Conf. on Functional Programming*, pages 83–91, 1996.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[Ash56] William Ross Ashby. *An Introduction to Cybernetics*. Chapman & Hall, London, 1956. ISBN 0416683002.

[BK01] Jeremy Brown and Thomas F. Knight, Jr. A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information flow. Technical Report ARIES-TM-015, MIT, November 2001.

[BN03] Anindya Banerjee and David A. Naumann. Using Access Control for Secure Information Flow in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 155–169, 2003.

[BN05] Anindya Banerjee and David A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[Bra85] Sheila L. Brand. Department of Defense Trusted Computer System Evaluation Criteria. National Computer Security Center, Fort Meade, Maryland, December 1985.

[BS99] G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldrop and T. Sato, editors, *Proc. FLOPS*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, November 1999.

[Coh77] Ellis S. Cohen. Information Transmission in Computational Systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proc. of the New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 September 1999. ACM Press.

[Fen74] J. S. Fenton. Memoryless Subsystems. *Computing J.*, 17(2):143–147, May 1974.

[GM82] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[GVS95] Milind Gandhe, G. Venkatesh, and Amitabha Sanyal. Labeled Lambda-Calculus and a Generalized Notion of Strictness (An Extended Abstract). In *Proc. Asian C. S. Conf. on Algorithms, Concurrency and Knowledge*, pages 103–110, 1995.

[HMS06] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified In-lined Reference Monitoring on .NET. In *ACM Workshop on Programming Languages and Analysis for Security*, Ottawa, Ontario, Canada, 10 2006. ACM Press.

[Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Passau, Germany, February 1987. Springer-Verlag.

[LBW05a] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec*, 4(1-2):2–16, 2005.

[LBW05b] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing Non-safety Security Policies with Program Monitors. In *ESORICS*, pages 355–373, 2005.

[LGJ05] Gurvan Le Guernic and Thomas Jensen. Monitoring Information Flow. In Andrei Sabelfeld, editor, *Proceedings of the Workshop on Foundations of Computer Security*, pages 19–30. DePaul University, June 2005. Affiliated with LICS'05.

[ML98] A. C. Myers and B. Liskov. Complete, Safe Information Flow with Decentralized Labels. In *Proc. IEEE Symp. Security and Privacy*, pages 186–197, 1998.

[MNZZ01] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java Information Flow, 2001. Soft. release. http://www.cs.cornell.edu/jif.

[MPL04] Wes Masri, Andy Podgurski, and David Leon. Detecting and Debugging Insecure Information Flows. In *15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 198–209. IEEE Computer Society Press, 2004.

[MS92] 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.

[MS01] Heiko Mantel and Andrei Sabelfeld. A Generic Approach to the Security of Multi-Threaded Programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 126–142, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.

[Mye99a] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. ACM Symp. Principles of Programming Languages*, pages 228–241, 1999.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, 1999.

[NSA95] NSA. Module Eight - Mandatory Access Control and Labels, January 1995.

[PC00] Fran cois Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. ACM International Conf. on Functional Programming*, pages 46–57, 2000.

[PS03] Francois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[Sim02] Vincent Simonet. Fine-grained Information Flow Analysis for a λ-calculus with Sum Types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 223–237, 2002.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[Smi01] Geoffrey Smith. A New Type System for Secure Information Flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[SS01] Andrei Sabelfeld and David Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[TA05] Tachio Terauchi and Alexander Aiken. Secure Information Flow as a Safety Problem. In *SAS*, pages 352–367, 2005.

[VBC+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3rd Edition*. O'Reilly, July 2000.

[Wei69] Clark Weissman. Security controls in the ADEPT-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

[Woo87] John P. L. Woodward. Exploiting the Dual Nature of Sensitivity Labels. In *Proc. IEEE Symp. Security and Privacy*, pages 23–31, 1987.

[ZM01] S. Zdancewic and A. C. Myers. Robust Declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic Security Labels and Noninterference. In *Proc. Workshop Formal Aspects in Security and Trust*, 2004.