

# Monitoring Information Flow

Gurvan Le Guernic      Thomas Jensen  
Université de Rennes 1 / CNRS  
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France  
{gleguern,jensen}@irisa.fr

April 7, 2006

## Abstract

We present an information flow monitoring mechanism for sequential programs. The monitor executes a program on standard data that are tagged with labels indicating their security level. We formalize the monitoring mechanism as a big-step operational semantics that integrates a static information flow analysis to gather information flow properties of non-executed branches of the program. Using the information flow monitoring mechanism, it is then possible to partition the set of all executions in two sets. The first one contains executions which *are safe* and the other one contains executions which *may be unsafe*. Based on this information, we show that, by resetting the value of some output variables, it is possible to alter the behavior of executions belonging to the second set in order to ensure the confidentiality of secret data.

**Keywords:** security, noninterference, language-based security, information flow control, monitoring, dynamic analyses, semantics

## 1 Introduction

This paper is concerned with the monitoring (or dynamic analysis) of information flow in sequential programs in order to ensure confidentiality. The goal of confidentiality analysis is to ensure that secret data will not be revealed to unauthorized parties by the execution of a program [3, 6]. A by now standard way of formalizing safe information flow is via the notion of *noninterference* introduced by Goguen and Meseguer [9]. Following the notation of Sabelfeld and Myers [18], noninterference (w.r.t. some low-equivalence relations  $=_L$  and  $\approx_L$  on states and observations) can be expressed as follows:

$$\forall s_1, s_2 \in S. s_1 =_L s_2 \Rightarrow \llbracket C \rrbracket s_1 \approx_L \llbracket C \rrbracket s_2 \quad (1)$$

This equation states that a command  $C$  is said to be *noninterfering* if and only if for any two states  $s_1$  and  $s_2$  that associate the same value to low (public) data (written  $s_1 =_L s_2$ ), the executions of the command  $C$  in the initial state  $s_1$  and  $s_2$  are “low-equivalent” ( $\llbracket C \rrbracket s_1 \approx_L \llbracket C \rrbracket s_2$ ). The “low-equivalent” relation characterizes the observational power of the attacker, by stating what he can distinguish. This may vary from requiring the output of low level of security to be equal for both executions, to requiring the two executions to have the same energy consumption. In the work presented in this paper, the attacker is considered to be only able to observe the low data of the initial state and of the final state.

As witnessed by the recent survey paper by Myers and Sabelfeld [18] there has been a substantial amount of research on static analysis for checking the noninterference property of programs, starting with the abstract interpretation of Mizuno and Schmidt [10] and the type based approach of Volpano, Smith and Irvine [21, 22]. Static analyses may reject a program because of *some* of its executions which might be unsafe; and thus deny

executions which are safe. The work presented in this paper attempt at preventing executions which are unsafe, while still allowing safe ones. This requires the definition of what is meant by “safe execution”. An execution of a command  $C$  starting in the original state  $s_1$  is said to be safe (or noninterfering) if and only if:

$$\forall s_2 \in S. s_2 =_L s_1 \Rightarrow \llbracket C \rrbracket s_1 \approx_L \llbracket C \rrbracket s_2 \quad (2)$$

In order to allow such noninterfering executions, one approach could consist in combining a standard static information flow analysis with other static analyses in order to determine conditions on input that lead to noninterfering executions. The determination of such conditions is a difficult problem. For example, it would be possible to run a partial evaluation of the program followed by a standard information flow analysis. However there would be infinitely many partial evaluations to run, one for each set of low-equivalent initial states. The approach presented in this paper extends the execution mechanism with a monitor that allows detecting illicit information flows and forbids final states which contain illicit information flows. This will allow validating certain executions of programs beyond the reach of current static analyses, at the price of additional run-time overhead incurred by the monitoring.

Monitoring information flow is more complicated than *e.g.* monitoring divisions by zero, since it has to take into account not only the current state of the program but also the execution paths that were not taken during execution. For example, executions in an initial state where  $h$  is false and  $x$  is 0 of

(a) `if h then x := 1 else skip;`  
and

(b) `if h then skip else skip;`

are equivalent concerning executed commands. However, if (b) is obviously a noninterfering program, the execution of (a) with the given initial state is not noninterfering. The execution of (a), with a low-equivalent initial state where  $h$  is true and  $x$  is 0, does not give the same final value for the low output  $x$ .

This leads to a monitoring mechanism which integrates a static analysis of commands which were not executed. The monitor will be defined formally as an operational semantics ( $\llbracket \cdot \rrbracket$ ) computing with tagged values. At any step of the evaluation of a program, the tags associated to any data identify a set of inputs which may have influenced the current value of the data up to this evaluation step. This monitoring mechanism is combined with a predicate (Safe) on the final state of the computation to obtain the following property for any command  $C$ :

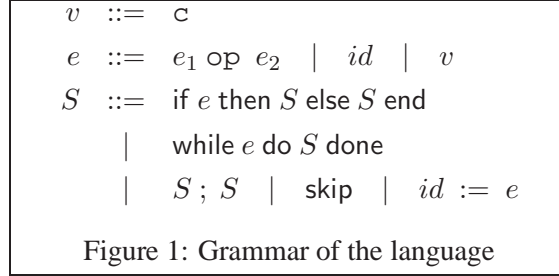
$$\forall s_1 \in S. \text{Safe}(\llbracket C \rrbracket s_1) \Rightarrow (\forall s_2 \in S. s_2 =_L s_1 \Rightarrow \llbracket C \rrbracket s_1 \approx_L \llbracket C \rrbracket s_2) \quad (3)$$

This states that all executions starting in a start state whose *low* (public) part is identical to the low part of the initial state of an execution satisfying Safe will be noninterfering (i.e. return the same values for low output). By comparison with static information flow analyses, we obtain information flow knowledge for a restricted set of input states, whereas static analyses infer a result valid for all executions. This implies a restriction of the potential paths taken into account; which enables the achievement of a better precision than with a standard static information flow analysis.

The paper is organized as follows. The next section presents a semantics integrating a monitor of information flow. It also gives a definition of the predicate Safe. This semantics and the predicate definition satisfy the equation (3), hence with this pair (semantics and predicate) it is possible to detect noninterfering executions. Once an information leak has been detected, the program behavior must be modified in order to prevent the leakage. Section 3 explores the idea of program behavior alteration based on information flow monitoring in order to ensure the respect of the confidentiality of secret data. It is observed that a simple analysis as the one developed in Section 2, although *sound* with regard to noninterference between secret inputs and public outputs, is not adequate to serve as a basis for behavior alteration. We then show a possible refinement of the analysis so that the information flow monitoring mechanism can “safely” direct the program’s behavior alteration. Finally, the paper concludes by presenting some related works and possible future developments of the information flow monitoring approach.

## 2 Detecting noninterfering executions

The programming language considered in this paper is a sequential language with integer and boolean expressions and including loop, conditional and assignment statements. The grammar is given in Figure 1.  $c$  stands for any constant value,  $op$  for any binary operator or relation on values, and  $id$  for any variable identifier (or name).



Variables and values are tagged with labels, intended for indicating their security level. In order to simplify our exploration of the concepts exposed in this paper, the security lattice considered is constituted of only two elements ( $\top$  and  $\perp$  with the usual ordering  $\perp \sqsubseteq \top$ ).

The special semantics on tagged data is defined as a “big step” evaluation semantics that defines an evaluation relation  $\Downarrow$ . It uses a value store to keep track of the value of variables. Similarly, a “tag store” is used to track the information flow between the inputs of the program and the current values in variables. Each input of the program receives a tag which reflects its security level ( $\top$  for high (secret) input and  $\perp$  for low (public) input). At any step of the execution, the set of tags associated to any variable by the “tag store” contains the tag of any input which has influenced the current value of the variable.

The forms of semantic judgments are described on top of Figure 2. The first environmental parameter is the value store (noted  $\sigma$  in the semantics rules); the second one is the tag store (noted  $\rho$  in the semantics rules). The evaluation of an expression returns a value and a set of tags. This set includes the tag of all input whose value influenced the value of the expression evaluated. For the evaluation of statements there is a third environmental parameter (noted  $T^{pc}$  in the semantics rules). It is a set of tags reflecting the information flowing through the “program counter”. It contains the tag of any input which has influenced the control flow of the program up to this point in the evaluation. The evaluation of a statement returns a new value store and a new tag store reflecting all the previous information flows created and those generated by the evaluation of this statement.

### 2.1 SDIF: Static and Dynamic Information Flow analysis

The semantic rules are given in Figure 2 page 4. In order to reduce the number of rules and focus on the information flow computation mechanism, the semantics uses  $op$  and  $c$ .  $op$  is the function corresponding to the symbol  $op$ . Similarly,  $c$  is the value corresponding to the constant  $c$ .

The rule (E<sub>S</sub>-ASSIGN) updates the value of the variable “ $id$ ” with the result of the evaluation of the expression  $e$ . It also updates the tags set of the variable in the resulting tags store. The new tags set is the union of  $T^e$ , which reflects the information flowing through the expression, and  $T^{pc}$ , which reflects the information flowing through the control flow of the program. The rule (E<sub>S</sub>-IF) evaluates the statement designated by the evaluation of the condition  $e$ , and updates the resulting tags store with the information flows created by the branch not evaluated using a special function  $\Phi$ .

The function  $\Phi (: \mathbb{I}d \rightarrow \mathcal{P}(\text{Tag})) \times \mathcal{P}(\text{Tag}) \times \mathbb{S} \rightarrow (\mathbb{I}d \rightarrow \mathcal{P}(\text{Tag}))$  is used whenever an if-statement is evaluated. Its aim is to modify the tag store so that it reflects the information flow created by the fact that one branch of the statement is not executed. In the following program “if  $h$  then  $x := k$  else skip end”,

$(\text{Id} \rightarrow \text{Value}); (\text{Id} \rightarrow \mathcal{P}(\text{Tag})) \vdash_{\mathcal{S}} \text{Expr} \Downarrow \text{Value} : \mathcal{P}(\text{Tag})$ $(\text{Id} \rightarrow \text{Value}); (\text{Id} \rightarrow \mathcal{P}(\text{Tag})); \mathcal{P}(\text{Tag}) \vdash_{\mathcal{S}} \mathbb{S} \Downarrow (\text{Id} \rightarrow \text{Value}) : (\text{Id} \rightarrow \mathcal{P}(\text{Tag}))$	
$\frac{\sigma; \rho \vdash_{\mathcal{S}} e \Downarrow v : T^e \quad \sigma; \rho; T^{pc} \cup T^e \vdash_{\mathcal{S}} S_v \Downarrow \sigma' : \rho'}{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} \text{if } e \text{ then } S_{true} \text{ else } S_{false} \text{ end} \Downarrow \sigma' : \Phi(\rho', T^e, S_{-v})}$	(E <sub>S</sub> -IF)
$\frac{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} \text{if } e \text{ then } S; \text{ while } e \text{ do } S \text{ done else skip end} \Downarrow \sigma' : \rho'}{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} \text{while } e \text{ do } S \text{ done} \Downarrow \sigma' : \rho'}$	(E <sub>S</sub> -WHILE)
$\frac{\sigma; \rho \vdash_{\mathcal{S}} e \Downarrow v : T^e}{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} id := e \Downarrow [id \mapsto v]\sigma : [id \mapsto T^{pc} \cup T^e]\rho}$	(E <sub>S</sub> -ASSIGN)
$\frac{}{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} \text{skip} \Downarrow \sigma : \rho}$	(E <sub>S</sub> -SKIP)
$\frac{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} S_1 \Downarrow \sigma' : \rho' \quad \sigma'; \rho'; T^{pc} \vdash_{\mathcal{S}} S_2 \Downarrow \sigma'' : \rho''}{\sigma; \rho; T^{pc} \vdash_{\mathcal{S}} S_1; S_2 \Downarrow \sigma'' : \rho''}$	(E <sub>S</sub> -SEQUENCE)
$\frac{\sigma; \rho \vdash_{\mathcal{S}} e_1 \Downarrow v_1 : T_1 \quad \sigma; \rho \vdash_{\mathcal{S}} e_2 \Downarrow v_2 : T_2}{\sigma; \rho \vdash_{\mathcal{S}} e_1 \text{ op } e_2 \Downarrow op(v_1, v_2) : T_1 \cup T_2}$	(E <sub>S</sub> -OP)
$\frac{}{\sigma; \rho \vdash_{\mathcal{S}} id \Downarrow \sigma(id) : \rho(id)}$	(E <sub>S</sub> -VAR)
$\frac{}{\sigma; \rho \vdash_{\mathcal{S}} c \Downarrow c : \emptyset}$	(E <sub>S</sub> -VAL)
Figure 2: Semantics rules	

the fact that  $x$  is different from  $k$  means that the `then`-branch has not been executed; and then that  $h$  is false. In this situation (where  $h$  is false), the final value of  $x$  is influenced by the initial value of  $h$  but not of  $k$ ; even if  $k$  is the expression appearing on the right side of the assignment. The function  $\Phi$  is built and used in order to take into account such information flows. A definition of the function  $\Phi$  is given in Figure 3 using a combining function  $\Pi$  ( $\Pi \equiv \lambda f \lambda g \lambda x. (f x) \cup (g x)$ ).  $\Phi$  adds the tags appearing in the tags set given in parameter to the tags set associated to any variable appearing on the left side of an assignment statement.

$\begin{aligned} \Phi(\rho, T, "S_1; S_2") &= \Phi(\rho, T, "S_1") \Pi \Phi(\rho, T, "S_2") \\ \Phi(\rho, T, \text{"if } e \text{ then } S_1 \text{ else } S_2 \text{ end"}) &= \Phi(\rho, T, "S_1") \Pi \Phi(\rho, T, "S_2") \\ \Phi(\rho, T, \text{"while } e \text{ do } S \text{ done"}) &= \Phi(\rho, T, "S") \\ \Phi(\rho, T, \text{"id := e"}) &= [id \mapsto (\rho(id) \cup T)]\rho \\ \Phi(\rho, T, \text{"skip"}) &= \rho \end{aligned}$
Figure 3: $\Phi$ 's semantics

The definition given here is a simple one. However it is sufficient to detect noninterfering executions with a reasonable level of precision. In the majority of cases, for programs manipulating more public inputs than secret

ones, the method presented in this section is more precise than flow insensitive analyses. Among those flow insensitive analyses are the standard security type systems which are widely studied in the domain of language based security. In the following program, where  $l$  is a public input,  $h$  a secret input,  $x$  a public output and  $\text{tmp}$  a temporary variable, a type system would give a security level to  $x$  at least as high than the one of  $h$ ; and then reject the program.

```

1  if ( l < 0 ) then { tmp := h } else { skip } end
2  if ( l > 0 ) then { x := tmp } else { skip } end

```

Using the semantics of Figure 2, all the executions of this program are detected as noninterfering (i.e. the tag of  $x$  at the end of the execution is  $\perp$ ). The reason of this better precision lies in the fact that the monitoring mechanism gives us the best possible *low control flow* information: *low control flow* designates the control flow produced by branching statements whose condition has a low level of security.

The evaluation of a command produces a value store and a tag store. The notation  $\llbracket C \rrbracket_{\sigma, \rho}^{\vee}$  designates the output value store produced by the evaluation of the command  $C$  with input values  $\sigma$  and input tags  $\rho$ .  $\llbracket C \rrbracket_{\sigma, \rho}^{\top}$  is similarly defined to be the output tag store. To summarize on those notations, the following holds:

$$\sigma; \rho; \emptyset \vdash_S C \Downarrow \llbracket C \rrbracket_{\sigma, \rho}^{\vee} : \llbracket C \rrbracket_{\sigma, \rho}^{\top}$$

Four sets of variables give a security specification of the program.  $H_i$  and  $L_i$  form a partition of the program's variables.  $H_i$  contains the variables holding a secret data in the initial state (i.e. secret inputs) and  $L_i$  contains public inputs. Similarly,  $H_o$  and  $L_o$  form a partition of the program's variables in which publicly observable variables in the final state belong to  $L_o$  and unobservable variables in the final state belong to  $H_o$ . A tag store  $\rho$  is said “well-tagged” if it respects the following properties:

$$\begin{aligned} \forall x \in H_i. \rho(x) &= \{\top\} \\ \forall x \in L_i. \rho(x) &= \{\perp\} \end{aligned}$$

**Definition 2.1** (Safe)

$$\text{Safe}(\llbracket C \rrbracket_{\sigma_1, \rho}^{\top}) \equiv \forall x \in L_o. \llbracket C \rrbracket_{\sigma_1, \rho}^{\top}(x) \subseteq \{\perp\}$$

Using the semantics and definition of Safe presented, the following theorem is an instance of the schema in equation (3).

**Theorem 2.1** For any command  $C$ , value stores  $\sigma_1$  and  $\sigma_2$ , and “well-tagged” tag store  $\rho$ , such that  $\text{Safe}(\llbracket C \rrbracket_{\sigma_1, \rho}^{\top})$  and  $\llbracket C \rrbracket_{\sigma_2, \rho}^{\vee} \neq \perp$ , if  $\sigma_1 =_{L_i} \sigma_2$  then  $\llbracket C \rrbracket_{\sigma_1, \rho}^{\vee} =_{L_o} \llbracket C \rrbracket_{\sigma_2, \rho}^{\vee}$ .

This theorem states that, for a given command, if the low outputs of an execution  $\epsilon$  are all tagged with  $\perp$ , then for all other terminating execution if the low inputs are equal to those of  $\epsilon$ , ( $\sigma_1 =_{L_i} \sigma_2$ ), then the low outputs will be equal to those of  $\epsilon$ , ( $\llbracket C \rrbracket_{\sigma_1, \rho}^{\vee} =_{L_o} \llbracket C \rrbracket_{\sigma_2, \rho}^{\vee}$ ).

The theorem 2.1 is similar to the equation (3) given in introduction. In fact, as the attacker can only observe the low outputs, the equality of the low outputs ( $\llbracket C \rrbracket_{\sigma_1, \rho}^{\vee} =_{L_o} \llbracket C \rrbracket_{\sigma_2, \rho}^{\vee}$ ) matches the equivalence of the final states as defined in the equation ( $\llbracket C \rrbracket_{s_1} \approx_L \llbracket C \rrbracket_{s_2}$ ). And similarly, the equality of low inputs ( $\sigma_1 =_{L_i} \sigma_2$ ) corresponds to the low equivalence of the initial states. The only visible difference is the statement “ $\llbracket C \rrbracket_{\sigma_2, \rho}^{\vee} \neq \perp$ ” in the theorem 2.1. However, as the attacker is unable to observe the termination behavior of the program, this statement is implied by the definition of low-equivalence of final states used in the equation (3). Therefore, we can conclude that if all the low outputs are tagged with  $\perp$  then the current execution is noninterfering, and then an attacker is unable to deduce any information about the high inputs.

To illustrate what precede, the result of the evaluation of the following program  $P$  is given in Table 1.

```

1  x := 0;
2  if l then
3    if h then { x := 1 } else { skip } end
4  else { skip } end

```

In this program,  $x$  is a low level output,  $l$  is a low level input (with tag  $\perp$ ), and  $h$  is a high input (with tag  $\top$ ).

$\sigma(l)$	$\sigma(h)$	$\llbracket P \rrbracket_{\sigma, \rho}^{\forall}(x)$	$\llbracket P \rrbracket_{\sigma, \rho}^{\top}(x)$
True	True	1	$\top$
True	False	0	$\top$
False	True	0	$\perp$
False	False	0	$\perp$

Table 1: Results for the output  $x$

### 3 Altering the program's behavior

The semantics described in the previous section enables the *detection* of a subset of noninterfering executions. The next step consists in the alteration of programs behavior in order to *enforce* the confidentiality of secret data. Our goal is to ensure that the set of all altered executions, for any program  $P$ , respects the noninterference property of Goguen and Meseguer as defined by the equation (1). This property states that any execution of the program is a noninterfering execution as defined by the equation (2). Consequently, the behavior alteration consists in:

- doing nothing for executions which are detected as noninterfering,
- modifying the output values of executions which may be interfering.

The altered execution of the program  $P$  started in the initial state  $s$  is noted  $\widetilde{\llbracket P \rrbracket} s$ .

The predicate *Safe* partitions the set of executions of the program  $P$  into two sets  $\mathcal{E}_{ni}$  (containing the executions for which the predicate *Safe* is true) and  $\mathcal{E}_{?}$  (containing the executions for which the predicate *Safe* is false). From the equation (3), we know that all the executions in  $\mathcal{E}_{ni}$  are noninterfering. The problem lies in the executions of  $\mathcal{E}_{?}$  among which some are noninterfering and some are not, and so may reveal information about the secret data. The solution envisioned consists in using a default output state  $s_o^d$ . As it is possible to detect, during the execution of the program  $P$ , if the current execution belongs to  $\mathcal{E}_{ni}$  or  $\mathcal{E}_{?}$ , it is possible to force the output store of all the executions belonging to  $\mathcal{E}_{?}$  to be  $s_o^d$ . Then, for any program  $P$  and initial state  $s_1$  the following properties hold:

$$\text{Safe}(\llbracket P \rrbracket s_1) \Rightarrow ( \text{Safe}(\widetilde{\llbracket P \rrbracket} s_1) \wedge (\forall s_2 \in S. s_2 =_L s_1 \Rightarrow \widetilde{\llbracket P \rrbracket} s_1 = \llbracket P \rrbracket s_1 \approx_L \llbracket P \rrbracket s_2 = \widetilde{\llbracket P \rrbracket} s_2) ) \quad (4)$$

$$\neg \text{Safe}(\llbracket P \rrbracket s_1) \Rightarrow ( \neg \text{Safe}(\widetilde{\llbracket P \rrbracket} s_1) \wedge \widetilde{\llbracket P \rrbracket} s_1 = s_o^d ) \quad (5)$$

If the predicate *Safe* gives the same answer for any two executions started in low-equivalent states, then the equations (4) and (5) imply that for all altered executions of any program  $P$  the following holds:

$$\forall s_1, s_2 \in S. s_2 =_L s_1 \Rightarrow ( (\widetilde{\llbracket P \rrbracket} s_1 \approx_L \widetilde{\llbracket P \rrbracket} s_2) \vee \widetilde{\llbracket P \rrbracket} s_1 = \widetilde{\llbracket P \rrbracket} s_2 = s_o^d ) \quad (6)$$

It is then obvious that the set of all altered executions, for any program  $P$ , respects the noninterference property of Goguen and Meseguer as defined in equation (1).

The following example illustrates the ideas exposed above using a program transformation altering the final value of the output  $x$  depending on its final tag.

```

1  x := 0;
2  if h then
3    if l then { x := 1 } else { skip } end
4  else { skip } end
5  if (T in tag(x)) then { x := 2 }

```

The 4 first lines correspond to the original program in which  $x$  is a low level output,  $l$  is a low level input (with tag  $\perp$ ), and  $h$  is a high input (with tag  $\top$ ). The 5<sup>th</sup> line is added to prevent information leakage. If, at the beginning of line 5, the tag of  $x$  contains  $\top$ , then  $x$  is reset to a default value (2 in this case, it could be what ever value is desired). The idea behind the 5<sup>th</sup> line is that **if**, at the beginning of line 5,  $x$  may have different values for two executions having the same low inputs **then** the tag of  $x$  will be  $\top$ ; so the test of the 5<sup>th</sup> line will succeed for both executions **and then**  $x$  will be reset to the same value (2 in this case) for both executions. This way, the program has been corrected in order to respect the noninterference property.

The tag, as computed by the semantics given in Section 2, at the end of line 4 (i.e. just before the information flow test) is given in Table 2 as a function of the input value of  $l$  (horizontally) and  $h$  (vertically). In this program,

$\sigma(h) \backslash \sigma(l)$	True	False
True	$\top$	$\perp$
False	$\top$	$\top$

Table 2:  $\llbracket P \rrbracket_{\sigma, \rho}^{\top}(x)$

if  $l$  is true it is possible to deduce the value of  $h$  by looking at the value of  $x$  before line 5. If  $x$  is 1 then  $h$  is true, and if  $x$  is 0 then  $h$  is false. This is reflected by the tag of  $x$  which is  $\top$  in both cases. Consequently, the value of  $x$  will be reset in both cases; those two altered executions of the program will then respect the noninterference property (i.e. the value of the output  $x$  is identical whatever the value of the high input is). Nevertheless, the statement added for correction is troublesome in a situation which was safe without it.

If  $l$  is false then  $x$  is equal to 0 whatever the value of  $h$  is. This means that those two executions respect the noninterference property before line 5. However, the tag of  $x$  is  $\perp$  if  $h$  is true, and  $\top$  if  $h$  is false. Both tags are correct because there is no flow from  $h$  to  $x$  and the tag reflects only a “*may* influence” relation. The problem with those tags is that, in the case where  $l$  is false, the correcting statement will change the value of  $x$  if and only if  $h$  is false. So, in the case where  $l$  is false, the value of  $x$  after the line 5 depends on the value of  $h$ . This implies that the set of all altered executions of the program does not respect the noninterference property.

### 3.1 A fully dynamic tag semantics

As shown in what precedes, in order for the equation (6) to holds, it is required that the predicate *Safe* returns the same answer for two executions started in low-equivalent states. If and only if that is the case, it is possible to secure programs based on the information flow computed dynamically. In our case, it means that the semantics must compute the same output tag stores for any two executions having the same low inputs. It is not the case for the semantics studied in Section 2.

Another semantics, whose rules can be found in Figure 4 page 8, has been developed. This semantics goes through all possible paths in order to compute adequate tags. When it encounters a branching statement it evaluates completely the branch that the condition designates (i.e. computes the new value store and tag store), and computes

$\frac{\begin{array}{l} \sigma; \rho \vdash_{\mathcal{F}} e \Downarrow v : T^e \\ \sigma; \rho; T^{pc} \cup T^e \vdash_{\mathcal{F}} S_{true} \Downarrow \sigma_{true} : \rho_{true} \\ \sigma; \rho; T^{pc} \cup T^e \vdash_{\mathcal{F}} S_{false} \Downarrow \sigma_{false} : \rho_{false} \end{array}}{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} \text{if } e \text{ then } S_{true} \text{ else } S_{false} \text{ end} \Downarrow \sigma_v : \{\!\!\{ \rho_{true}, \rho_{false} \}\!\!\}_v^{T^e}}$	(E <sub><math>\mathcal{F}</math></sub> -IF)
$\frac{\begin{array}{l} \sigma; \rho \vdash_{\mathcal{F}} e \Downarrow v : T^e \\ \sigma; \rho; T^{pc} \cup T^e \vdash_{\mathcal{F}} S ; \text{while } e \text{ do } S \text{ done} \Downarrow \sigma' : \rho' \end{array}}{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} \text{while } e \text{ do } S \text{ done} \Downarrow \{\!\!\{ \sigma', \sigma \}\!\!\}_v^{\emptyset} : \{\!\!\{ \rho', \rho \}\!\!\}_v^{T^e}}$	(E <sub><math>\mathcal{F}</math></sub> -WHILE)
$\frac{\sigma; \rho \vdash_{\mathcal{F}} e \Downarrow v : T^e}{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} id := e \Downarrow [id \mapsto v]\sigma : [id \mapsto T^{pc} \cup T^e]\rho}$	(E <sub><math>\mathcal{F}</math></sub> -ASSIGN)
$\frac{}{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} \text{skip} \Downarrow \sigma : \rho}$	(E <sub><math>\mathcal{F}</math></sub> -SKIP)
$\frac{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} S_1 \Downarrow \sigma' : \rho' \quad \sigma'; \rho'; T^{pc} \vdash_{\mathcal{F}} S_2 \Downarrow \sigma'' : \rho''}{\sigma; \rho; T^{pc} \vdash_{\mathcal{F}} S_1 ; S_2 \Downarrow \sigma'' : \rho''}$	(E <sub><math>\mathcal{F}</math></sub> -SEQUENCE)
$\frac{\sigma; \rho \vdash_{\mathcal{F}} e_1 \Downarrow v_1 : T_1 \quad \sigma; \rho \vdash_{\mathcal{F}} e_2 \Downarrow v_2 : T_2}{\sigma; \rho \vdash_{\mathcal{F}} e_1 \text{ op } e_2 \Downarrow \text{op}(v_1, v_2) : T_1 \cup T_2}$	(E <sub><math>\mathcal{F}</math></sub> -OP)
$\frac{}{\sigma; \rho \vdash_{\mathcal{F}} id \Downarrow \sigma(id) : \rho(id)}$	(E <sub><math>\mathcal{F}</math></sub> -VAR)
$\frac{}{\sigma; \rho \vdash_{\mathcal{F}} \odot \Downarrow \odot : \emptyset}$	(E <sub><math>\mathcal{F}</math></sub> -VAL)
$\{\!\!\{ x, y \}\!\!\}_v^{T^e} = \begin{cases} x \cup y & \text{if } \top \in T^e \\ x & \text{if } \top \notin T^e \text{ and } v = true \\ y & \text{if } \top \notin T^e \text{ and } v = false \end{cases}$	

Figure 4: Rules of the full-paths semantics

the new tag store returned by the evaluation of the other branch. The tag store the semantics returns in such a situation is the join of the two tag stores (one for each branch). Using this semantics, the following theorem has been proved to hold.

**Theorem 3.1** *For any command  $C$ , value stores  $\sigma_1$  and  $\sigma_2$ , and “well-tagged” tag store  $\rho$ , such that  $\text{Safe}(\llbracket C \rrbracket_{\sigma_1, \rho}^{\top})$  and  $\llbracket C \rrbracket_{\sigma_2, \rho}^{\vee} \neq \perp$ , if  $\sigma_1 =_{L_i} \sigma_2$  then  $\llbracket C \rrbracket_{\sigma_1, \rho}^{\vee} =_{L_o} \llbracket C \rrbracket_{\sigma_2, \rho}^{\vee}$  and  $\llbracket C \rrbracket_{\sigma_1, \rho}^{\top} =_{L_o} \llbracket C \rrbracket_{\sigma_2, \rho}^{\top}$ .*

This is sufficient to be able to safely alter the behavior of programs in order to ensure the respect of the noninterference property. Nevertheless, the semantics used is highly inefficient. For any execution of a program, the semantics evaluates all paths which are accessible by any execution started in a low-equivalent initial state. Moreover, as soon as the semantics encounters a **while**-statement branching on a condition influenced by a high level input (but not if the condition depends only on public inputs), the semantics loops forever. This is quite disturbing and the reason for the current development of another semantics.

## 4 Related Works

The vast majority of information flow analyses are static and involve type systems [18]. In the recent years, this approach has reached a good level of maturity. Pottier and Conchon described in [16] 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 [11], JIF [14], and FlowCaml [19, 17]. There also exists an interpreter for FlowCaml. This interpreter dynamically type data, commands and functions which are successively evaluated. Nevertheless, it types commands the same way the static analysis does. And then, the interpreter merges the types of both branches of an **if**-statement without taking into account, when possible, the fact that one branch is executed and the other one is not.

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 [11, 12], which is an evolution of Java, uses the *decentralized label model* of Myers and Liskov [13]. 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 [26]. Zheng and Myers [27] 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 [2] 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.

Going further than *testing* dynamically labels, there has been research on dynamically *computing* labels. At the level of languages, Abadi, Lampson, and Lévy expose in [1] a dynamic analysis based on the labeled  $\lambda$ -calculus of Lévy. This analysis computes the dependencies between the different parts of a  $\lambda$ -term and its final result in order to save this result for a faster evaluation of any future equivalent  $\lambda$ -term. Also based on a labeled  $\lambda$ -calculus, Gandhe, Venkatesh, and Sanyal [8] 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” [23]. 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 *indirect flows* created by the non-execution of one of the branches of a statement. At the level of operating systems, Weissman [24] 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 [25]. This method deals with the problem of over-classification of data in computer systems implementing the MAC security model. The main difference between those two works and ours lies in the granularity of label application. In those models [24, 25], 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 [20] 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 through branching statements containing different assignments. At the level of computers themselves, Fenton [7] 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. As Fenton shows himself, his mechanism does not ensure confidentiality with *variable* data marks. At the same level, Brown and Knight [4] 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 this paper, this is a feature which can be used to gain information about secrets in some cases. For example, Table 1 shows that it

is possible to deduce the value of  $h$  when  $l$  is true and  $\llbracket P \rrbracket_{\sigma, \rho}^{\forall}(x)$  is 0; even if no assignment to  $l$  or  $x$  has been executed. With a program similar to the one used as example in page 6, their machine does not prevent the flow from  $h$  to  $x$  when  $l$  is true and  $h$  is false.

## 5 Conclusion

In this paper, we refine the notion of noninterference, concerning all possible executions of a program, to a notion of noninterfering execution. All possible initial states of a program are partitioned in equivalence classes. The equivalence relation is based on the value of the public inputs of the program. Two initial states are equivalent if and only if they have the same values for public inputs. An execution, started in the initial state  $s$ , is said to be noninterfering if any execution, started in an initial state belonging to the same equivalence class than  $s$ , returns the same values for the public outputs of the program.

Refining the notion of noninterference to the level of execution offers two main advantages. The first one is that it is now possible to *safely* run noninterfering executions of a program which is not noninterfering. The second benefit is a better precision in the analysis of some programs. A static information flow analysis has to take into consideration all the potential paths of all the executions of the program. Using the method presented in this paper to ensure the respect of confidentiality, only the potential paths of executions low-equivalent to the current one are taken into consideration. This feature results in a better precision towards possible execution paths. For example, in the following program,  $h$  is a secret input,  $l$  a public input,  $tmp$  a temporary variable which is not an output, and  $x$  is the only public output.

```

1  if ( (cos l)^2 < 0.1 ) then { tmp := h } else { skip } end
2  if ( (tan l) < 3 ) then { x := tmp } else { skip } end

```

It is likely that a static analysis would conclude that the program is not noninterfering because of a bad flow from  $h$  to  $x$ . However, the program *is* noninterfering. As “ $(\cos x)^2 + (\sin x)^2 = 1$ ” and “ $\tan x = \frac{\sin x}{\cos x}$ ”, there is no  $l$  such that  $(\cos l)^2 < 0.1$  and  $(\tan l) < 3$ . It follows that there is no execution of the program which evaluates both assignments. Consequently, there is never a flow from  $h$  to  $x$ . The mechanism proposed in this paper would allow all executions of this program. The reason is that, for any low-equivalent class of executions, there is exactly *one* possible path. And so, only the current execution path is taken into consideration when determining if a given execution is noninterfering or not.

Concerning the capacity of the attacker, this work considers an attacker which is only able to get information from the low outputs of the program at the end of the computation. Another limitation concerns termination of programs. The mechanism developed here does not prevent information leakage from the study of the termination behavior of programs (neither does it take care of timing covert channels either). The system proposed in this paper could prevent those flaws using a technique similar to the one found in [5]. In short, the authors of this paper track the security level of variables appearing in while-loop conditions and other statements influencing the termination. This is efficient but restrictive since it forbids any loop conditioned by a secret. That is the reason why those types of covert channels are not taken into consideration at first.

We propose a special semantics and a predicate on the final state of an execution which, together, are able to detect noninterfering executions. This semantics mixes dynamic mechanism and static analysis techniques. When the semantics encounters a branching statement, the branch designated by the value of the condition is evaluated and the other branch is analyzed. The aim of the analysis is to extract the information flow created by the fact that the given branch is not executed. The result of the analysis and the evaluation of the other branch are merged together to build the resulting information flows corresponding to the evaluation of the branching statement.

The next step of this work consists in altering programs behavior in order to ensure an appropriate behavior of programs towards confidentiality. However, the first semantics presented does not necessarily return the same

result about noninterference for two executions whose initial states belong to the same equivalence class. This prevents the use of this semantics for the programs behavior alteration in order to ensure confidentiality. In Section 3 we describe succinctly a first attempt at improving the semantics. The resulting semantics is proved sufficient to ensure the respect of confidentiality by all altered execution of any program. However, this semantics does not terminates for programs containing a while-statement conditioned by a “secret” data.

Future work will involve the development of a semantics having a precision enabling the insertion of dynamic tests, but having better termination properties. This semantics will use an analysis of non-executed branches based on the model of flow logic [15] in a way similar to [5] since this model seems to have a good precision. In particular, it does not require that a variable keeps the same security level in all the statements. The precision of this model will be improved by taking into account the knowledge (i.e. the value store) gathered by the semantics up to the starting point of the analysis.

**Acknowledgment.** Discussions with David Schmidt and Anindya Banerjee during the development of this work have been helpful; as well as their comments on this paper.

## References

- [1] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proc. ACM International Conf. on Functional Programming*, pages 83–91, 1996.
- [2] A. Banerjee and D. 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.
- [3] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, May 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.
- [4] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, Nov. 2001.
- [5] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *J. Computing Languages*, 28(1):3–28, 2002.
- [6] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.
- [7] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.
- [8] M. Gandhe, G. Venkatesh, and A. 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.
- [9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.
- [10] 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.

- [11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. Principles of Programming Languages*, pages 228–241, 1999.
- [12] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, 1999.
- [13] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. Security and Privacy*, pages 186–197, 1998.
- [14] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow, 2001. Soft. release. <http://www.cs.cornell.edu/jif>.
- [15] H. R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 223–244, 2002.
- [16] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conf. on Functional Programming*, pages 46–57, 2000.
- [17] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.
- [18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
- [19] V. Simonet. Fine-grained information flow analysis for a  $\lambda$ -calculus with sum types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 223–237, 2002.
- [20] G. E. Suh, J. W. Lee, D. Zhang, and S. 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.
- [21] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 607–621, 1997.
- [22] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
- [23] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl, 3rd Edition*. O’Reilly, July 2000.
- [24] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.
- [25] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. IEEE Symp. Security and Privacy*, pages 23–31, 1987.
- [26] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.
- [27] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proc. Workshop Formal Aspects in Security and Trust*, 2004.