HAL open science

Non-Interference through Annotated Multisemantics

Gurvan Cabon, Alan Schmitt

To cite this version:

Gurvan Cabon, Alan Schmitt. Non-Interference through Annotated Multisemantics. 28ièmes Journées Francophones des Langages Applicatifs, Jan 2017, Gourette, France. hal-01503094

HAL Id: hal-01503094

https://hal.science/hal-01503094v1

Submitted on 6 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Non-Interference through Annotated Multisemantics

G. Cabon[1] & A. Schmitt[2]

1: inria
gurvan.cabon@inria.fr
2: inria
alan.schmitt@inria.fr

Abstract

Non-interference can be defined as a program property that give guaranties on the independence of specific (public) outputs of a program from specific (secret) inputs. The notion of non-interference does not depend on one particular execution of the program (unlike illegal memory access for example), but on its global behavior.

To develop a certified system verifying information flows, such as non-interference, we propose to only rely on the execution of the program, and thus investigate such properties using directly the derivation tree of an execution.

Considering a single execution is clearly not sufficient to determine if a program has the non-interference property. Surprisingly, studying every execution independently is also not sufficient. This is why we propose a formal approach that builds, from a given semantics, a multisemantics that allows to reason on several executions at once. Adding annotations in this multisemantics lets us capture the dependencies between inputs and outputs of a program.

To motivate and demonstrate our approach, we provide a concrete example where it is clear that reasoning on all the executions at once is required, and we show that our approach works on this example.

This is a work in progress, partially formalized in Coq. Ultimately, our goal is to automatically build the multisemantics from the semantics, and to prove that the method correctly approximate non-interference, i.e., if a pair of input and output are independent according to the annotations, then changing the input does not result in a different output.

1. Introduction and motivation

Suppose we have a programming language in which variables can be private or public, and where the programs can take variables as parameters. We say a program is non-interferent if, for any pair of execution that differs only on the private parameters, the values of the public variables are the same. In other words, changing the value of the private variables does not influence the public variables. Or in yet other word, the public variables do not depend on the private variables: there is no leak of private information. In all this work, we only consider finite program executions. This property is crucial in terms of privacy and security. As a simple first example, the naive program in Figure 1, where `public` is a public parameter and `secret` is a private variable, is clearly interferent (or not non-interferent). Changing the value of `secret` changes the value of `public`. We call this a *direct flow* of information because the value of `secret` is directly assigned into `public`.

```
public := secret
```

Figure 1: Example of naive interference

But catching non-interference is not always that easy, because it does not simply consist of the transitive closure of direct flows. It may also depend on the context in which a particular instruction is executed. For example in Figure 2 we have a program that has an *indirect flow*. The value of `secret` is not directly stored into `public` but the condition in the if statements ensures that in each case `secret` receives the value of `public`.

```
if secret
  then public := true
  else public := false
```

Figure 2: Example of indirect flow

Another source of interference is the fact that *not* executing a part of the code can give information (masking). For example Figure 3 shows a program with a mask. In the case where `secret` is false, the variable `public` is not modified. It gives us the information that the first branch of the if statement has not been executed and that `secret` is $false$. Nevertheless the value of `secret` always ends up in the variable `public`.

```
public := false
if secret
  then public := true
  else skip
```

Figure 3: Example of indirect flow with a mask

This last example shows that we cannot look at a single execution of the program to determine non-interference because in the one where `secret` is set to $false$ we do not even modify `public` after the initialization.

The situation is even more dire. In the example shown in Figure 4, we can see that there exists no execution where the flow can be inferred. In the first execution (in the center), `public` depends on `y`, which is not touched by the execution. In the second execution (on the right), `public` still depends on `y`, which itself depends on `x`, which is not touched by the execution. Hence in both cases there seems to be no dependency on `secret`. Yet, we have `public` = `secret` at the end of both execution, so the secret is leaked. Looking at every execution independently is not sufficient.

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$secret = true$

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$public = true$

$secret = false$

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$public = false$

Figure 4: Running Example (executed code, non-executed code)

To recover the inference of information flow only by looking at executions, we propose to look at

every executions at once, and to combine the information gathered by several executions. In the case of Figure 4, we can see that `x` depends on `secret` in the first execution at the end of the first `if`. Hence, in the second execution, `x` must also depend on `secret`, as the fact that not modifying it is an information flow. We can similarly deduce that `y` depends on `x` in both executions, hence `public` transitively depends on `secret`.

To formally define this approach, we first describe how to simultaneously execute the same program with different inputs, then show how annotations let us soundly capture the dependencies between variables and inputs for this program. We then formalize this work in the Coq proof assistant letting us state the correctness of our method.

Related Work Studies about non-interference take their roots in 1977 with E. Cohen [6] and D. E. Denning & P. J. Denning [7]; and then formalized in 1982 by J. A. Goguen & J. Meseguer [8] as following:

> One groups 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.

There are actually many formalization of non-interference, and in particular non-interference can depend on the termination or not of the program executions as *termination-insensitive non-interference* [1], *termination-aware non-interference* [3], *timing- and termination-sensitive non-interference* [10].

More recently, in 2003, A. Sabelfeld & A. C. Myers [13] give an overview of the information-flow techniques and show the many sources of potential interference. Later G. Le Guernic [11] published his PhD thesis in which he proposes and proves a precise dynamic analysis for non-interference. A. Sabelfeld and A. Russo [12, 14] also proved several properties comparing static and dynamic approaches of non-interference. G. Barthe, P.R. D'Argenio & T. Rezk [2] reduce the problem of non-interference of a program into a safety property of a transformation of the program. S. Hunt & D. Sands [9] presented a family of semantically sound type system for non-interference. In 2010, M.R. Clarkson & F.B. Schneider studied the hyperproperties, properties (like non-interference for example) that do not depend on only one program execution [5].

Outline In Section 2, we give the foundation of our work: the WHILE language we work on and its Pretty-Big-Step semantics. In Section 3, we describe how the multisemantics works and we give the property we want it to have with respect to the Pretty-Big-Step semantics. In Section 4, we extend the multisemantics with annotations and show that this approach correctly captures the interference of our running example. We conclude with the future works and a look back at the work presented in this paper in Section 5.

2. Pretty-big-step (PBS)

Before describing how our multisemantics works, we need to introduce a language and its Pretty-Big-Step semantics. We use the toy language WHILE :

$\langle \textit{expression} \rangle$ `e` ::= `Const n` | `Var x` | `Op e e` | `MEM n`

$\langle \textit{statement} \rangle$ `s` ::= `Skip` | `Seq s s` | `If e s s` | `While e s` | `Assign x e` | `OBS e n`

We consider a set of values *Val* and a set of variables *Var*. Our language is first of all composed of expressions : constants, variables, an operator of arity 2 and an expression MEM able to read a value

$$\frac{}{\sigma, \underline{t} \to \sigma'}\ \textsc{Axiom} \qquad \frac{\sigma_1, \underline{t'} \to \sigma'_1}{\sigma, \underline{t} \to \sigma'_1}\ \textsc{Rule1} \qquad \frac{\sigma_1, \underline{t1} \to \sigma'_1 \qquad \sigma_2, \underline{t2} \to \sigma'_2}{\sigma, \underline{t} \to \sigma'_2}\ \textsc{Rule2}$$

Figure 5: The 3 types of rules in Pretty-Big-Step

from an external memory. This memory is fixed at the beginning of the execution and the program cannot write in it. One should see this memory as the arguments or the inputs of the programs. Then, we also have statement in the WHILE language: a skip statement, the sequence, a conditional jump, a while loop, a statement to assign an expression to a variable and a statement OBS, the dual of MEM, able to output an expression in some channels that can be observed by an external observer. When speaking of either an expression or a statement, we will refer to it as a *term* and usually denoted by t. In our semantics, a memory $M \in \mathit{Mem}$ will be a triplet, generally noted (F, E, C), where $F \in \mathit{fixMem}$ is the fixed memory giving for each memory cell a value, $E \in \mathit{Env}$ the environment that associates a value to each variable and $C \in \mathit{Chan}$ the set of channels represented by a function from indexes to lists of values.

$$\mathit{Cell} := \mathbb{N}$$

$$\mathit{fixMem} := \mathit{Cell} \mapsto \mathit{Val}$$

$$\mathit{Env} := \mathit{Var} \mapsto \mathit{Val}$$

$$\mathit{Index} := \mathbb{N}$$

$$\mathit{Chan} := \mathit{Index} \mapsto \mathit{Val}\ \mathit{list}$$

$$\mathit{Mem} := \mathit{fixMem} \times \mathit{Env} \times \mathit{Chan}$$

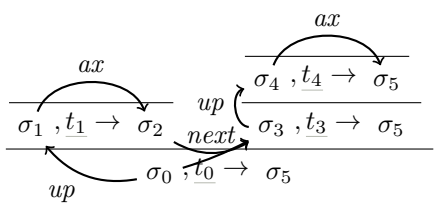
The Pretty-Big-Step semantics, introduced by Charguéraud [4], is inspired by the Big-Step semantics with some constraints on the premises of the rules. In Pretty-Big-Step, there are at most two inductive premises for each rule. Therefore, we can categorize the rules in 3 groups shown in Figure 5: the rules with no inductive premise (axioms), the rules with one inductive premise (rules 1) and the rules with two inductive premises (rules 2).

where $\sigma, \sigma', \sigma_1, \sigma'_1, \sigma_2, \sigma'_2 \in \mathit{State}$ are states, either a memory or a pair of a memory and a value or a triplet of a memory and two values.

$$\mathit{State} := \mathit{Mem} \cup (\mathit{Mem} \times \mathit{Val}) \cup (\mathit{Mem} \times \mathit{Val} \times \mathit{Val})$$

In the rule $\sigma, \underline{t} \to \sigma'$ we call the state σ the semantic context because it represents the memory and it gives information on potential values that have already been evaluated and we call σ' the result of the rule because it shows how is the memory after computation and it may give some values with it.

The restriction to such rules allows to define them as a set of *transfer functions*, depending on their kind, as depicted in below. Axioms have a single transfer function *ax* relating a semantic context to a result, rules 1 have a single transfer function *up* relating a semantic context to a new semantic context, and rules 2 have two transfer functions: *up* as in rules 1, and *next* relating a result and a semantic context to a new semantic context.



$$\frac{}{\sigma, \underline{c} \to \sigma, c}\ \textsc{Cst} \qquad \frac{E[\underline{\mathtt{x}}] = v}{(F, E, C), \underline{\mathtt{x}} \to (F, E, C), v}\ \textsc{Var} \qquad \frac{\sigma, \underline{e_1} \to \sigma' \qquad \sigma',\ \underline{\mathtt{op}_1\ e_2} \to \sigma''}{\sigma, \underline{e_1\ \mathtt{op}\ e_2} \to \sigma''}\ \textsc{Op}$$

$$\frac{\sigma, \underline{e_2} \to \sigma' \qquad \sigma',\ \underline{\mathtt{op}_2} \to \sigma''}{\sigma,\ \underline{\mathtt{op}_1\ e_2} \to \sigma''}\ \textsc{Op1} \qquad \frac{v = v_1\ \mathtt{op}\ v_2}{M, v1, v2,\ \underline{\mathtt{op}_2} \to M, v}\ \textsc{Op2}$$

$$\frac{F[n] = v}{(F, E, C), \underline{\mathtt{MEM}\ n} \to (F, E, C), v}\ \textsc{Mem}$$

Figure 6: Expression rules of the Pretty-Big-Step semantics

In Pretty-Big-Step, the intermediate steps of evaluation are made explicit through *extended terms*, defined as follows.

$\langle$*extended_expression*$\rangle$ `e_ext` ::= `Basic e` | `op1 e` | `op2`

$\langle$*extended_statement*$\rangle$ `s_ext` ::= `Basic s` | `Seq1 s` | `If1 s s` | `While1 e s` | `While2 e s` | `Assign1 x` | `OBS1 n`

These statements and expressions cannot be used by a programmer, but they are used in the Pretty-Big-Step semantics rules shown in Figures 6 and 7. The operator `op` is here evaluated as the addition without losing generality.

To simplify the reading of the rules, we use some notations.

$\underline{c}$ for `Const` c

$\underline{\mathtt{x}}$ for `Var` x

$\underline{e_1\ \mathtt{op}\ e_2}$ for `Op` e_1 e_2

$\underline{s_1; s_2}$ for `Seq` s_1 s_2

$\underline{;_1\ s_2}$ for `Seq1` s_2

$\underline{\mathtt{x}\ \mathtt{=}\ e}$ for `Assign` x e

$\underline{\mathtt{x}\ \mathtt{=}_1}$ for `Assign1` x

$\underline{\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2}$ for `If` e s_1 s_2

$\underline{\mathtt{while}\ e\ \mathtt{do}\ s}$ for `While` e s

$f[\mathtt{x} \mapsto v]$ denotes the function $y \mapsto \begin{cases} \text{v} & \text{if x = y} \\ \text{f(y)} & \text{otherwise} \end{cases}$

$x :: A$ for the set containing all the elements of A plus the element x.

Besides the many advantages given in Charguéraud's paper about Pretty-Big-Step, we rely on the strict structure of the rules to make the creation of our multisemantics easier (and, as future work, to automatize this creation).

$$\frac{}{\sigma, \underline{\texttt{skip}} \to \sigma}\ \textsc{Skip} \qquad \frac{\sigma, \underline{s_1} \to \sigma' \qquad \sigma', \underline{;_1\, s_2} \to \sigma''}{\sigma, \underline{s_1; s_2} \to \sigma''}\ \textsc{Seq} \qquad \frac{\sigma, s \to \sigma'}{\sigma, \underline{;_1\, s} \to \sigma'}\ \textsc{Seq1}$$

$$\frac{\sigma, \underline{e} \to \sigma' \qquad \sigma', \underline{\texttt{if1}(s_1, s_2)} \to \sigma''}{\sigma, \underline{\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \to \sigma''}\ \textsc{If} \qquad \frac{M, \underline{s_1} \to \sigma}{(M, \texttt{true}), \underline{\texttt{if1}(s_1, s_2)} \to \sigma}\ \textsc{IfTrue}$$

$$\frac{M, \underline{s_2} \to \sigma}{(M, \texttt{false}), \underline{\texttt{if1}(s_1, s_2)} \to \sigma}\ \textsc{IfFalse} \qquad \frac{\sigma, \underline{e} \to \sigma' \qquad \sigma', \underline{\texttt{while1}(e, s)} \to \sigma''}{\sigma, \underline{\texttt{while}\ e\ \texttt{do}\ s} \to \sigma''}\ \textsc{While}$$

$$\frac{M, \underline{s} \to \sigma \qquad \sigma, \underline{\texttt{while2}(e, s)} \to \sigma'}{(M, \texttt{true}), \underline{\texttt{while1}(e, s)} \to \sigma'}\ \textsc{WhileTrue1} \qquad \frac{\sigma, \underline{\texttt{while}\ e\ \texttt{do}\ s} \to \sigma'}{\sigma, \underline{\texttt{while2}(e, s)} \to \sigma'}\ \textsc{WhileTrue2}$$

$$\frac{}{(M, \texttt{false}), \underline{\texttt{while1}(e, s)} \to M}\ \textsc{WhileFalse} \qquad \frac{\sigma, \underline{e} \to \sigma' \qquad \sigma', \underline{\texttt{x} =_1} \to \sigma''}{\sigma, \underline{\texttt{x} = e} \to \sigma''}\ \textsc{Asg}$$

$$\frac{E' = E[\texttt{x} \mapsto v]}{((F, E, C), v), \underline{\texttt{x} =_1} \to (F, E', C)}\ \textsc{Asg1} \qquad \frac{\sigma, \underline{e} \to \sigma' \qquad \sigma', \underline{\texttt{OBS}_1(n)} \to \sigma''}{\sigma, \underline{\texttt{OBS}(e, n)} \to \sigma''}\ \textsc{Obs}$$

$$\frac{C' = C[n \mapsto v :: C[n]]}{((F, E, C), v), \underline{\texttt{OBS}_1(n)} \to (F, E, C')}\ \textsc{Obs1}$$

Figure 7: Statement rules of the Pretty-Big-Step semantics

3. Multisemantics

Now we have our Pretty-Big-Step semantics, we can build the multisemantics. A rule will have this structure :

$$\frac{P_1 \qquad \dots \qquad P_n}{\underline{t} \Downarrow \mu}$$

where $P_1, \dots, P_n$ are premises, s is a statement and μ is a relation between states. Informally μ relates semantic contexts to results, if two states σ_1 and σ_2 are in relation by μ then with the Pretty-Big-Step semantics, s transforms the semantic context σ_1 into the result σ_2. μ represents the set of all the executions we are considering in the multiderivation.

We give the rules of the multisemantics in Figures 8 and 9. Each rule of the multisemantics (except `MltMergeExpr` and `MltMergeStmt`) precisely follows the corresponding rule in the Pretty-Big-Step semantics. For intelligibility concerns we omit some useless (in terms of understanding the rules) identifiers using the character "_".

In order to evaluate a constant $\underline{c}$ with a relation μ, the rule `MltCst` just has to ensure that every pair of states in relation by μ is composed of a first state σ and a second state σ' of the form (σ, c). The rule `MltVar` does the same thing but makes sure that the value v in the result of each pair in relation by μ is actually the value of `x` in the environment E of the semantic context. The rule `MltOp` first evaluates the first expression e_1, lets `MltOp1` take care of the evaluation of the second expression e_2 and adds a condition ensuring that for every pair of states (σ, σ'') in relation by μ, there exists an intermediate state σ' which is the result associated to σ in the relation μ' (the relation used to

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma' = (\sigma, c)}{\underline{c} \Downarrow \mu}\ \textsc{MltCst}$$

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ v\ E, \sigma' = (\sigma, v) \wedge \sigma = (_, E, _) \wedge E[\mathtt{x}] = v}{\underline{\mathtt{x}} \Downarrow \mu}\ \textsc{MltVar}$$

$$\frac{\underline{e_1} \Downarrow \mu' \qquad \underline{\mathtt{op}_1\ e_2} \Downarrow \mu'' \qquad \forall\ \sigma\ \sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{\underline{e_1\ \mathtt{op}\ e_2} \Downarrow \mu}\ \textsc{MltOp}$$

$$\frac{\underline{e_2} \Downarrow \mu' \qquad \underline{\mathtt{op}_2} \Downarrow \mu'' \qquad \forall\ \sigma\ \sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma' \mu''\ \sigma''}{\underline{\mathtt{op}_1\ e_2} \Downarrow \mu}\ \textsc{MltOp1}$$

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ M\ v_1\ v_2, \sigma = (M, v_1, v_2) \wedge \sigma' = M, (v_1\ \mathtt{op}\ v_2)}{\underline{\mathtt{op}_2} \Downarrow \mu}\ \textsc{MltOp2}$$

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ F\ v, \sigma = (F, _, _) \wedge F[n] = v \wedge \sigma' = (\sigma, v)}{\underline{\mathtt{MEM}\ n} \Downarrow \mu}\ \textsc{MltMem}$$

$$\frac{\underline{e} \Downarrow \mu_1 \qquad \underline{e} \Downarrow \mu_2 \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma\mu_1\sigma' \vee \sigma\mu_2\sigma'}{\underline{e} \Downarrow \mu}\ \textsc{MltMergeExpr}$$

Figure 8: Expression rules of the multisemantics semantics

evaluate e_1) and the semantic context associated to σ'' in the relation μ'' (the relation used to delay the evaluation of e_2). `MltOp1` evaluates the second expression e_2, postpones the operation between the two values and gives the same insurance concerning μ. The operation is finally computed in the rule `MltOp2` where, similar to the first two rules, the premise only guarantees that the states in relation are coherent to the operation (here the addition). The MEM expression is handled by the rule `MltMem` by doing the same thing as `MltVar` but looking in the fixed memory F. The last rule `MltMergeExpr` is explained later, after the explanation of the rules for the statements.

The rule `MltSkip` checks that every pair of states in relation by μ is of the form (σ, σ). The rule `MltSeq` first evaluates the left statement s_1 and let `MltSeq1` define what to do with s_2, which is simply evaluate it in the new state. As always, when two inductive premises are used, there is the condition ensuring that for every pair of states in relation by μ we can legitimately do both inductive steps. `MltIf` is analogous to the previous rules, but `MltIfTrue` introduces a new behaviour: here, for every pair (σ, σ') of states in relation by μ, σ must carry the value $true$ ($\sigma = (M, true)$) and then M and σ' are in relation by μ_1, the relation used to evaluate s_1. The rule `MltFalse` takes care of the case when every state σ in relation to a state σ' by μ carries the value $false$ ($\sigma = (M, false)$) and then M and σ' are in relation by μ_2, the relation used to evaluate s_2. The rules `MltWhile`, `MltWhileTrue1`, `MltWhileTrue2` and `MltWhileFalse` do a similar thing to the rules `MltIf`, `MltIfTrue` and `MltIfFalse` with the difference that in the case of `MltFalse`, we have no inductive premise and the only needed guarantee is that the memory does not change. When evaluating an assignment statement the rule `MltAsg` evaluates the expression e and then the rule `MltAsg1` handles the requirement that each pair in relation by μ is a pair of two identical states, except that the result is an updated version of the semantic context according to the corresponding variable and value. In a similar way, `MltObs` and `MltObs1` update the channels by adding the value carried by every semantic context of μ in the channel n.

$$\frac{\forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma = \sigma'}{\texttt{skip} \Downarrow \mu}\ \textsc{MltSkip}$$

$$\frac{s_1 \Downarrow \mu' \qquad ;_1\, s_2 \Downarrow \mu'' \qquad \forall S\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{s_1; s_2 \Downarrow \mu}\ \textsc{MltSeq} \qquad \frac{s \Downarrow \mu}{;_1\, s \Downarrow \mu}\ \textsc{MltSeq1}$$

$$\frac{e \Downarrow \mu' \qquad \texttt{if1}(s_1, s_2) \Downarrow \mu'' \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \mu}\ \textsc{MltIf}$$

$$\frac{s_1 \Downarrow \mu_1 \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ M, (\sigma = (M, \texttt{true}) \wedge M\ \mu_1\ \sigma')}{\texttt{if1}(s_1, s_2) \Downarrow \mu}\ \textsc{MltIfTrue}$$

$$\frac{s_2 \Downarrow \mu_2 \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ M, (\sigma = (M, \texttt{false}) \wedge M\ \mu_2\ \sigma')}{\texttt{if1}(s_1, s_2) \Downarrow \mu}\ \textsc{MltIfFalse}$$

$$\frac{e \Downarrow \mu' \qquad \texttt{while1}(e, s) \Downarrow \mu'' \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{\texttt{while}\ e\ \texttt{do}\ s \downarrow \mu}\ \textsc{MltWhile}$$

$$\frac{\begin{array}{c} s \Downarrow \mu_1 \\ \texttt{while2}(e, s) \Downarrow \mu_1' \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ M\ \sigma', (\sigma = (M, \texttt{true}) \wedge \sigma\ \mu_1\ \sigma' \wedge \sigma' \mu_1'\ \sigma'') \end{array}}{\texttt{while1}(e, s) \Downarrow \mu}\ \textsc{MltWhileTrue1}$$

$$\frac{\texttt{while}\ e\ \texttt{do}\ s \Downarrow \mu}{\texttt{while2}(e, s) \Downarrow \mu}\ \textsc{MltWhileTrue2}$$

$$\frac{\forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ M, (\sigma = (M, \texttt{false}) \wedge \sigma'' = M)}{\texttt{while1}(e, s) \Downarrow \mu}\ \textsc{MltWhileFalse}$$

$$\frac{e \Downarrow \mu' \qquad \texttt{x} =_1 \Downarrow \mu'' \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{\texttt{x} = e \Downarrow \mu}\ \textsc{MltAsg}$$

$$\frac{\forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ E\ E'\ F\ C\ v, E' = E[\texttt{x} \mapsto v] \wedge \sigma = ((F, E, C), v) \wedge \sigma' = (F, E', C)}{\texttt{x} =_1 \Downarrow \mu}\ \textsc{MltAsg1}$$

$$\frac{e \Downarrow \mu' \qquad \texttt{OBS}_1(n) \Downarrow \mu'' \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''}{\texttt{OBS}(e, n) \Downarrow \mu}\ \textsc{MltObs}$$

$$\frac{\forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ E\ F\ C\ C'\ v, C' = C[\texttt{n} \mapsto v :: C[\texttt{n}]] \wedge \sigma = ((F, E, C), v) \wedge \sigma' = (F, E, C')}{\texttt{OBS}_1(n) \Downarrow \mu}\ \textsc{MltObs1}$$

$$\frac{s \Downarrow \mu_1 \qquad s \Downarrow \mu_2 \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma\mu_1\sigma' \vee \sigma\mu_2\sigma'}{s \Downarrow \mu}\ \textsc{MltMergeStmt}$$

Figure 9: Statement rules of the multisemantics semantics

```
if Mem 1
  then Obs (Mem 2) 1
  else Obs (Mem 3) 1
```

Figure 10: A program needing the merge rule

```
n := Mem 1
i := 0
While (i<=n) do
   i := i + 1
```

Figure 11: Counter example to the reciprocal of the first theorem

Finally we have special rules `MltMergeExpr` and `MltMergeStmt` to manage expressions and statements for which different Pretty-Big-Step rules can be applied, depending on the states. Let us consider a multiderivation of the program in Figure 10 with a μ relating at least one state containing $true$ in the first memory cell with another state, and one state containing $false$ in the first memory cell with another state. The derivation tree starts with the rule `MltIf`: one premise is the evaluation of the guard and the other one is the evaluation of $\texttt{if1}(s_1, s_2)$ with a relation μ' where the first states of the pairs carry two different values. Neither the rule `MltIfTrue`, nor the rule `MltIfFalse` can be applied since all the pairs of state in relation μ' do not contain the same value in the first state. But we can use the rule `MltMergeStmt` to separate those pairs of state into two μ_1 and μ_2 in which all the values carried are respectively the sames. And then, we can continue the derivation tree with the rules `MltIfTrue` and `MltIfFalse`.

One property we expect from this multisemantics is that for every term, finding a derivation in the multisemantics for a relation μ implies that we can find a derivation in the Pretty-Big-Step semantics for every pair of states in relation by μ. The reciprocal is not true: let us consider a infinite number of Pretty-Big-Step derivation of the program in Figure 11 for which the WHILE language has been extend with the lower or equal $\Leftarrow$ operator. In the semantic context of the nth derivation, the first memory cell is set to n. A multiderivation contain all those semantic contexts would be endless because for every n, the Pretty-Big-Step derivation in the $(n+1)^{th}$ state takes at least one step more than the derivation in the nth state. We can not derive a multisemantics derivation with all those states because it would require an infinite derivation, which is not possible in our case. We formalize this in the Coq proof assistant in Figure 12. We still prove that for a finite number of Pretty-Big-Step derivation, we can derive a multisemantics derivation by the second and third theorem.

`multisemantics_statement s mu` and `pretty_big_step_statement st1 s st2` are inductive predicate respectively describing the multisemantics and the Pretty-Big-Step semantics, and they are respectively equivalent to $s \Downarrow \mu$ and $st1, s \rightarrow st2$. `sum mu1 mu2` is the relation containing exactly the pairs of states present in $mu1$ or $mu2$. The proof of the first theorem is conducted by induction on the inductive predicate $\Downarrow$, and the second on by induction on the inductive predicate $\rightarrow$. The last theorem is just the use of the rule `MltMergeStmt`.

The three theorems are proved for expressions too.

4. Annotations

We can now introduce annotations to track information flows. To achieve this, the annotations record the memory cells from which each variable and each channel depends in a *dependency* generally noted D of type *Dep*. Additionally, we define the *context dependency* CD of type *ConDep* by a set of memory

```
Theorem Correct_statement :
  forall s mu,
    multisemantics_statement s mu
    ->
    (forall st1 st2, mu st1 st2 -> pretty_big_step_statement st1 s st2)
.

Theorem Correct_statement2 :
  forall s st1 st2,
    pretty_big_step_statement st1 s st2
    -> multisemantics_statement s (fun a b => a = st1 /\ b = st2)
.

Theorem Correct_statement3 :
  forall s mu1 mu2,
    multisemantics_statement s mu1
    -> multisemantics_statement s mu2
    -> multisemantics_statement s (sum mu1 mu2).
```

Figure 12: Correctness theorem in coq

cells : it represents the dependency of the context in which the current expression or statement is evaluated and is used to track indirect flows. For example, when entering the evaluation of one branch of an `if` statement we need to know the dependency given by the condition.

$$Dep := (Var \cup Index) \mapsto Cell\ set$$

$$ConDep := Cell\ set$$

The rules will look like the multisemantics rules but with the information before and after evaluating the term:

$$\frac{P_1 \quad \ldots \quad P_n}{(D, CD), \underline{t} \Downarrow \mu, (D', CD')}$$

where $P_1, \ldots, P_n$ are premises, t is a term, μ is a relation between states, D and D' are *dependencies* and CD and CD' are *context dependencies*. A *context dependency* CD on the left means that the context in which the term is evaluated depends on the cells of CD, but a *context dependency* CD' on the right means that the term itself depends on the cell of CD': it is the set of dependencies used to evaluate the term.

One example of where the multisemantics clearly shows its role is on the rule `AnnotMergeState` in which we merge the annotations obtained in each branch.

$$\frac{(D, CD), \underline{s} \Downarrow \mu_1, (D_1, CD_1) \qquad (D, CD), \underline{s} \Downarrow \mu_2, (D_2, CD_2) \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma\mu_1\sigma' \vee \sigma\mu_2\sigma'}{(D, CD), \underline{s} \Downarrow \mu, (D', CD_1 \cup CD_2)}\ \textsc{MltMergeStmt}$$

where D' is the function defined for all variables or channels by $D'(x) = D_1(x) \cup D_2(x)$.

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma' = (\sigma, c)}{(D, CD), \underline{c} \Downarrow \mu, (D, CD)}\ \textsc{AnnotMltCst}$$

$$\frac{\forall\ \sigma\ \sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ v\ E, \sigma' = (\sigma, v) \wedge \sigma = (_, E, _) \wedge E[\mathtt{x}] = v}{(D, CD), \underline{\mathtt{x}} \Downarrow \mu, (D, x :: CD)}\ \textsc{AnnotMltVar}$$

$$\frac{\forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \sigma = \sigma'}{(D, CD)\underline{\mathtt{skip}} \Downarrow \mu, (D, CD)}\ \textsc{AnnotMltSkip}$$

$$\frac{\begin{array}{c}(D, CD), \underline{e} \Downarrow \mu', (D_e, CD_e)\\ (D_e, CD_e), \underline{\mathtt{if1}(s_1, s_2)} \Downarrow \mu'', (D', CD') \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''\end{array}}{(D, CD), \underline{\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2} \Downarrow \mu, (C', CD')}\ \textsc{MltIf}$$

$$\frac{(C, CD), \underline{s_1} \Downarrow \mu_1, (C_1, CD_1) \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ M, (\sigma = (M, \mathtt{true}) \wedge M\ \mu_1\ \sigma')}{(C, CD), \underline{\mathtt{if1}(s_1, s_2)} \Downarrow \mu, (C_1, CD_1)}\ \textsc{MltIfTrue}$$

$$\frac{(C, CD), \underline{s_2} \Downarrow \mu_2, (C_2, CD_2) \qquad \forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ M, (\sigma = (M, \mathtt{false}) \wedge M\ \mu_2\ \sigma')}{(C, CD), \underline{\mathtt{if1}(s_1, s_2)} \Downarrow \mu, (C_2, CD_2)}\ \textsc{MltIfFalse}$$

$$\frac{\begin{array}{c}(D, CD), \underline{e} \Downarrow \mu', (D', CD')\\ (D', CD'), \underline{\mathtt{x} =_1} \Downarrow \mu'', (D'', CD'') \qquad \forall \sigma\sigma'', \sigma\ \mu\ \sigma'' \Rightarrow \exists\ \sigma', \sigma\ \mu'\ \sigma' \wedge \sigma'\ \mu''\ \sigma''\end{array}}{(D, CD), \underline{\mathtt{x}\ =\ e} \Downarrow \mu, (D'', CD'')}\ \textsc{AnnotMltAsg}$$

$$\frac{\forall \sigma\sigma', \sigma\ \mu\ \sigma' \Rightarrow \exists\ E\ E'\ F\ C\ v, E' = E[\mathtt{x} \mapsto v] \wedge \sigma = ((F, E, C), v) \wedge \sigma' = (F, E', C)}{(D, CD), \underline{\mathtt{x} =_1} \Downarrow \mu(D[x \mapsto CD], CD)}\ \textsc{AnnotMltAsg1}$$

Figure 13: Some rules of the annotated multisemantics

Rules are easy to annotate, thus we give in Figure 13 only the few rules we use in the example below. Notice that in the rule `AnnotMltAsg1` we strongly update the dependencies of the variable because when assigning the value, there is no other dependencies than what is currently in the context dependency.

We have re-written the running example of Figure 4 in the WHILE language in Figure 14 with the value of secret stored in the first cell of the fixed memory and the public variable being the first channel. Let us go back to this example and see how our method captures the dependency.

Consider two executions, one which sets the first cell to `false` and one which sets it to `true`. Both environments E_e are empty at the beginning (formally, each variable is set to a special value *Unbound*). The dependencies and the context dependency are also empty: we write D_e the empty dependency, a function returning an empty set for every variable and index. We note F_v the fixed memory in which the first cell is set to the value `v` , E_v the empty environment except that `x` is set to `v` and `y` is set to `true`, and C_e the set of channels where each is empty.

When evaluating the first `if` statement, we have to evaluate the condition $\underline{\mathtt{MEM}\ 1}$ and then evaluate each block with a smaller relation (due to the rule `MltMergeStmt`) depending on the condition. The

```
x = true;
y = true;
if Mem 1
    then x = false
    else skip;
if x
    then y = false
    else skip;
Obs y 1
```

Figure 14: The running example in the WHILE language

reader can easily verify that the first part is evaluated into

$$\frac{\dots}{(D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_1, (D_e[x \mapsto \{1\}], \{1\})} \textsc{AnnotMltIfTrue}$$

and the second one into

$$\frac{\dots}{(D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_2, (D_e, \{1\})} \textsc{AnnotMltIfFalse}$$

where μ_1 is the relation only verifying $((F_{true}, E_{true}, C_e), \texttt{true})\ \mu_1\ (F_{true}, E_{false}, C_e)$ and μ_2 the relation only verifying $((F_{false}, E_{true}, C_e), \texttt{false})\ \mu_2\ (F_{false}, E_{true}, C_e)$.

It leads us to evaluate the statement $\underline{\texttt{if1}(\texttt{x = false},\texttt{skip})}$ as follow

$$\frac{\begin{array}{c}(D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_1, (D_e[x \mapsto \{1\}], \{1\}) \\ (D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_2, (D_e, \{1\}) \\ \forall \sigma\sigma', \sigma\ \mu_{if1}\ \sigma' \Rightarrow \exists\ M, (M\ \mu_1\ \sigma') \vee (M\ \mu_2\ \sigma')\end{array}}{(D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_{if1}, (D_e[x \mapsto \{1\}], \{1\})} \textsc{AnnotMltMergeStmt}$$

where μ_{if1} is the relation only verifying
$((F_{true}, E_{true}, C_e), \texttt{true})\ \mu_{if1}\ (F_{true}, E_{false}, C_e)$ and
$((F_{false}, E_{true}, C_e), \texttt{false})\ \mu_{if1}\ (E_{false}, M_{true}, C_e)$.

And the full `if` statement is evaluate by

$$\frac{\begin{array}{c}(D_e, \emptyset), \underline{\texttt{MEM 1}} \Downarrow \mu_{mem1}(D_e, \{1\}) \\ (D_e, \{1\}), \underline{\texttt{if1}(\texttt{x = false}, \texttt{skip})} \Downarrow \mu_{if1}, (D_e[x \mapsto \{1\}], \{1\}) \\ \forall \sigma\sigma'', \sigma\ \mu_{if}\ \sigma'' \Rightarrow \exists\ \sigma', (\sigma\ \mu_{mem1}\ \sigma') \wedge (\sigma'\ \mu_{if1}\ \sigma'')\end{array}}{(D_e, \emptyset), \underline{\texttt{if MEM 1 then x = false else skip}} \Downarrow \mu_{if}, (D_e[x \mapsto \{1\}], \{1\})} \textsc{AnnotMltIf}$$

where μ_{mem1} is the relation only verifying
$(F_{true}, E_{true}, C_e)\ \mu_{mem1}\ ((F_{true}, E_{false}, C_e), \texttt{true})$ and
$(F_{false}, E_{true}, C_e)\ \mu_{mem1}\ ((F_{false}, E_{true}, C_e), \texttt{false})$
and μ_{if} is the relation only verifying
$(F_{true}, E_{true}, C_e)\ \mu_{if}\ (F_{true}, E_{false}, C_e)$ and
$(F_{false}, E_{true}, C_e)\ \mu_{if}\ (F_{false}, E_{true}, C_e)$.

We see that, without even going further, we already know that `x` depends on the first cell of the fixed memory. The second `if` statement has the same behaviour: at the end we know that y depends

```
Theorem Correctness :
  forall s,
  forall S1 S1',
  forall S2 S2',
  forall D D' C C' mu,
  forall cell,
  forall index,
  pretty_big_step_statement S1 s S1'                                  (*a*)
  /\ pretty_big_step_statement S2 s S2'                               (*b*)
  /\ annotated_multi_pretty_big_step_statement (D,C) s mu (D',C')     (*c*)
  /\ mu S1 S1' /\ mu S2 S2'                                           (*d*)
  /\ statesDifferOnlyOnCell cell S1 S2                                (*e*)
  /\ channelsDifferOnIndex index S1' S2'                              (*f*)
  -> In (cell) (D' index)                                             (*g*)
.
```

Figure 15: Theorem of correctness of the annotation

on the first cell of the fixed memory. Finally, when observing `y`, the dependency flows into the first channel. If we call our program *runningExample* we will have

$$\frac{}{(D_e, \emptyset), runningExample \Downarrow \mu_{RE}, (D, \{1\})}$$

where $D = D_e[x \mapsto \{1\}][y \mapsto \{1\}][1 \mapsto \{1\}]$ and μ_{RE} is the relation only verifying
$(F_{true}, E_e, C_e)\ \mu_{RE}\ (F_{true}, E_{false}, C_e[1 \mapsto [\texttt{true}]])$ and
$(F_{false}, E_e, C_e)\ \mu_{RE}\ (F_{false}, E_{true}[y \mapsto \texttt{false}], C_e[1 \mapsto [\texttt{false}]])$

We see that we effectively have $1 \in D(1) = \{1\}$.

To express the correctness of the annotation, we want that, for any term, if a channel depends on a cell, this cell should appear in the dependency associated to the channel. The Coq theorem we want to prove is shown in Figure 15. An analogous theorem can be established for expressions.

Suppose we have two Pretty-Big-Step executions of a statement s (lines a & b) with the semantic contexts S_1 and S_2 and the respective results S_1' and S_2' and an annotated multiexecution of the same statement annotated by (D, C) on the left and (D', C') on the right (line c) with a relation μ containing at least (S_1, S_1') and (S_2, S_2') (line d). If we add the constraints that the semantic contexts S_1 and S_2 have the same environment, the same set of channels, the same potential value carried and the same fixed memory except for the cell *cell* (line e) and that the channel indexed by *index* of the results S_1' and S_2' are different (line f) (without any constraint on the fixed memory, the environment, the other channels and the potential values carried), then the cell *cell* is effectively in the dependencies of the channel *index* (line g).

Another important property of our method is that it is more precise than syntactic methods, i.e., methods where annotations are collected on the syntax of a program. We suppose that our language has been extended with the operators $=<$ and $==$ which are respectively the lower or equal operator and the equal operator. We also use the infix operator $+$ instead of the prefix operator `Op`. In the example of Figure 16, the annotations will say that `x` depends only on cell 1 (because of `n`) and not on cell 2 because in every execution, the loop will end up overwriting the value of `x` by the constant $false$. We end up with a channel 1 depending on cell 1 but not on cell 2. In the other hand, any syntactic method (for example we could adapt one from Sabelfeld and Myers approach [13]) approximates the dependencies after the `if` statement saying that `x` always depends on cells 1 & 2, and then the channel 1 also depends on cells 1 & 2.

```
i = 0;
n = Mem 1;
x = true;
while i =< n
   if i == n
      then x = false
      else x = Mem 2;
   i = i + 1;
Obs x 1
```

Figure 16: An example where the annotations do not overapproximate

5. Conclusion and Future Work

We have presented a new semantics, the multisemantics, built from a Pretty-Big-Step semantics and able to evaluate a term of a WHILE language with different inputs all at once. We have built an annotation system for this semantics to track information flows. We expressed in Coq the correctness theorems we want to prove and we showed with examples the precision of this method compared to syntactic analyses.

Our next step is to prove the correctness of the annotations for arbitrary programs.

In the longer term, we want to build a framework to automatically derive a multisemantics from a semantics in Pretty-Big-Step. We also want to investigate the existence of local conditions on annotations, which would be checked for each rule independently, that would ensure the global correctness of the annotations, i.e., correctly capture flows.

References

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 333–348, 2008.

[2] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[3] Nataliia Bielova and Tamara Rezk. A taxonomy of information flow monitors. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 46–67, 2016.

[4] Arthur Charguéraud. Pretty-big-step semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 41–60, 2013.

[5] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[6] Ellis Cohen. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 133–139, New York, NY, USA, 1977. ACM.

[7] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[8] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982.

[9] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 79–90, 2006.

[10] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 413–428, 2011.

[11] Gurvan Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.

[12] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 186–199, 2010.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[14] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, pages 352–365, 2009.