Programmable enforcement framework of information flow policies

Minh Ngo and Fabio Massacci

University of Trento, Italy
{surname}@disi.unitn.it

Abstract. We propose a programmable framework that can be easily instantiated to enforce a large variety of information flow properties. Our framework is based on the idea of secure multi-execution in which multiple instances of the controlled program are executed in parallel. The information flow property of choice can be obtained by simply implementing programs that control parallel executions. We present the architecture of the enforcement mechanism and its instantiations for non-interference (NI) (from Devriese and Piessens), non-deducibility (ND) (from Sutherland) and some properties proposed by Mantel, such as removal of inputs (RI) and deletion of inputs (DI), and demonstrate formally soundness and precision of enforcement for these properties.

Keywords: Non-Interference, Non-Deducibility, Possibilistic Information Flow Properties, Programming Language, Secure Multi Execution

1 Introduction

Computing systems often process data classified as *sensitive*, or, secret. To ensure security, treatment of these data has to comply with designated information flow policies that regulate whether the publicly visible behavior of a system can be influenced by secret data.

Non-interference (NI) [7] totally prevents leakage of secrets to public channels by requiring that the confidential information does not interfere with all events at the public levels. With or without the confidential information, observations at the public levels are still the same. By weakening or strengthening the definition of NI, security researchers have proposed different information flow properties. For example, declassification policies accept the behaviors in which some selected secret data can be released [14]. Sutherland defines [15] *non-deducibility* (ND), a stronger property than NI [6]. It assumes that an attacker knows the design of the observed program, and has partial access to the public program interfaces, and tries to infer the occurrence and non-occurrence of sequences of high input events. ND prevents the attacker from deducing which confidential event sequences have occurred or not.

Existing mechanisms for information flow policies enforcement and secure information release are based on several techniques: e.g., type systems [13], symbolic execution [2], multi-execution [5, 11], faceted values [1], etc. Yet these all

Table 1. EMs for the selected information flow policies

Policy	Components		
	MAP	REDUCE	T_M/T_R
Termination (in)sensitive non-interference [5]	Fig. 3a	Fig. 3b	Fig. 3
Termination (in)sensitive non-deducibility [15]	Fig. 3a	Fig. 3b	Fig. 7
Removal of inputs [9]	Fig. 8a	Fig. 3b	Fig. 8
Deletion of inputs [9]	Fig. 9a	Fig. 3b	Fig. 9

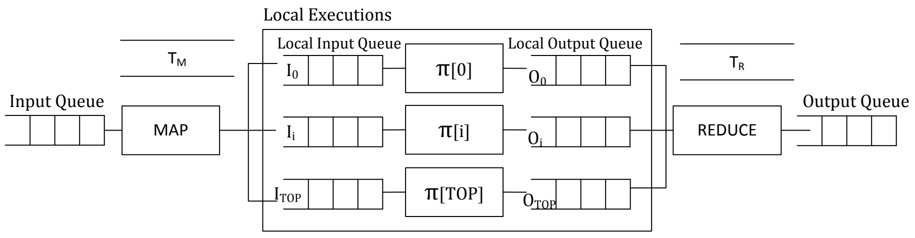

Fig. 1. Architecture of enforcement mechanisms

fall short in the same aspect: *these approaches work only for a single information flow policy*, typically NI or NI equipped with declassification. Modification of these mechanisms to enforce another information flow policy (for example, ND) is not straight-forward. Moreover, no run-time enforcement mechanism is proposed for ND.

1.1 The contribution of this paper

We propose a *programmable enforcement mechanism* (EM) that can be easily configured to enforce NI, ND and other information flow policies. Configurations of the EMs are summarized in Tab. 1. Our proposal is the first run-time EM that covers ND. SME by Devriese and Piessens [5] is a special case. Our EM relies on the secure multi-execution technique (SME) [5] in which multiple instances of the controlled program are executed in parallel and their input and output behaviors are controlled. To this construction we add two programmable components that map each input to the multiple instances and reduce output of the instances to a single output. We demonstrate soundness and precision of the proposed mechanisms using the operational semantics.

The rest of the paper is organized as follows. §2 gives an overview of the idea behind our approach and the architecture of the enforcement framework. Selected information flow policies and implementations of their EMs are introduced respectively in §3 and §4. Semantics of controlled programs and framework is introduced in §5. The soundness and precision of the EMs constructed are presented in §6. We discuss related work and conclude in §7.

2 Overview

Fig. 1 depicts the general architecture of the EM for an information flow property on a program π. It is composed by the MAP and REDUCE components, a stack EX of local executions ($\pi[0], \dots, \pi[TOP]$, where TOP is the index of the top of the stack), global input and output queues, and the tables T_M and T_R. Instructions used to compose controlled programs, MAP, and REDUCE programs are in respectively Fig. 2a, Fig. 2b, and Fig. 2c.

Local executions are instances of the original program, are executed in parallel and are unaware of each other. They are separated from the environment input and output actions by the EM. The local input (resp. output) queue of a local execution contains the input (resp. output) items that can be freely consumed (resp. generated) by this local execution. MAP and REDUCE are responsible for respectively the global input queue containing the input items from the external environment, and the global output queue containing the output items filtered by the EM to the environment. When a local execution needs an input item that is not yet ready in its local input queue it will request the help of MAP by emitting an *interrupt signal*. When a local execution generates an output item it emits a signal to request the help of REDUCE.

$$\begin{array}{lll}
\pi ::= & & \textit{program instructions}: \\
& |x := e & \textit{assignment} \\
& |\pi ; \pi & \textit{sequence} \\
& |\textbf{if } e \textbf{ then } \pi \textbf{ else } \pi & \textit{if} \\
& |\textbf{while } e \textbf{ do } \pi & \textit{while} \\
& |\textbf{skip} & \textit{skip} \\
& |\textbf{input } x \textbf{ from } c & \textit{input} \\
& |\textbf{output } e \textbf{ to } c & \textit{output}
\end{array}$$

(a) Basic instructions

$$\begin{array}{lll}
\pi_M ::= \pi & & \textit{instructions}: \\
& |\textbf{map}(e, c, PRED[\,]) & \textit{map} \\
& |\textbf{wake}(PRED[\,]) & \textit{wake} \\
& |\textbf{clone}(PRED[\,], PRIV_{T_M}, PRIV_{T_R}) & \textit{clone}
\end{array}$$

(b) MAP instructions

$$\begin{array}{lll}
\pi_R ::= \pi & & \textit{instructions}: \\
& |\textbf{retrieve } x \textbf{ from } (i, c) & \textit{retrieve} \\
& |\textbf{wake}(PRED[\,]) & \textit{wake} \\
& |\textbf{clean}(c, PRED[\,]) & \textit{clean}
\end{array}$$

(c) REDUCE instructions

π, e, x, and c are meta-variables for respectively instructions, expressions, variables, and input/output channels. A (controlled, MAP, or REDUCE) program is a sequence of instructions.

Fig. 2. Language instructions

MAP and REDUCE can autonomously send and, respectively, collect items from local queues. The actions of MAP (resp. REDUCE) on an input (resp. output) request from a local execution depend on the configuration information in the table T_M (resp. T_R). This configuration is based on two privileges: *ask* (a) and *tell* (t). These components of the EM are customized depending on the desired information flow policy.

All local executions with the *tell* privilege on the input channel c can get the real value from the channel c when MAP broadcasts the input item to local executions, otherwise they will get a default value. If a local execution has the *tell* privilege on the output channel c, REDUCE can tell its value to the environment. Otherwise, REDUCE will just replace it with a default value.

If a local execution has the *ask* privilege on the input channel c, then MAP can fetch the input item from the environment upon receiving a signal from a local execution. A local execution with the *ask* privilege on the output channel c can ask REDUCE to start processing outputs from the local executions.

An execution with only the *ask* but not the *tell* privilege in T_R will activate REDUCE to retrieve output items, but REDUCE will not put the value in the external output (i.e. will not tell it to anyone). The execution will have to wait for somebody else with the *tell* privilege on the channel to produce an output.

3 Information flow policies

In this section we briefly present some policies.

Non-Interference. Let $(\pi, I) \Downarrow O$ denote a terminating execution of π that consumes input sequence I and generates output sequence O. Given a security level l (where l is in $\{L, H\}$), $I|_l$ (resp. $O|_l$) returns the projection of the sequence I (resp. O) containing only items at level l. For NI, for two arbitrary input sequences I and I' that are low-equivalent ($I'|_L = I|_L$), the generated outputs O and O' are also low-equivalent ($O'|_L = O|_L$). NI comes in termination-sensitive (TSNI) or termination-insensitive (TINI) flavors.

Definition 1 (TINI). *A program π is* TINI *iff*

$$\forall I, I' : I'|_L = I|_L \wedge (\pi, I) \Downarrow O \wedge (\pi, I') \Downarrow O' \implies O'|_L = O|_L$$

The formal definition of TSNI can be derived from TINI by moving $(\pi, I') \Downarrow O'$ after the implication.

Non-Deducibility. Sutherland defines ND by using two views: the first view corresponds to secret events, and the second view corresponds to observations of attackers at the low level [15]. There are no flows from from the former to the latter if the two views can always be combined. In this way an attacker cannot know whether a particular high input took place, because it can be always replaced by another valid input and still yield a valid execution.

Termination-insensitive ND (TIND) is defined in Def. 2. TIND requires that for any two inputs I and I^*, such that the program terminates with these inputs, there exists another input I^{**}, which is low-equivalent with I ($I|_L = I^{**}|_L$), high-equivalent to I^* ($I^*|_H = I^{**}|_H$), and if the program terminates with I^{**}, the generated output visible to attackers at the low level (L) is not changed. Termination-sensitive ND (TSND) assumes that attackers can observe terminations of executions and the existence of the default view where we replaced input values with default values. If the default values could not be accepted by an execution then it would be possible to deduce that the high information is actually different from the default value.

Definition 2 ((Input-Output) TIND). *A program π is* TIND *iff*

$$\begin{aligned}\forall I, I^* : (\pi, I) \Downarrow O \wedge (\pi, I^*) \Downarrow O^* \implies (\exists I^{**} : I|_L = I^{**}|_L \wedge I^*|_H = I^{**}|_H \wedge \\ \wedge ((\pi, I^{**}) \Downarrow O^{**} \implies O|_L = O^{**}|_L))\end{aligned}$$

The formal definition of TSND can be derived from TIND by requiring that $(\pi, I^{**}) \Downarrow O^{**}$ holds and the execution where all input values have been replaced by default values is always present and terminates.

Removal of Inputs. RI [9] requires that if a possible trace is perturbed by removing all high input items, then the result can be corrected into a possible trace. In our notation, an input (resp output) is a queue of input (resp. output) vectors (see §5). If all high input items in an input I are replaced by default items ($\vec{df}$) or removed, the input can be modified to an input I' such that the program terminates when executing on I' and the generated output will be equivalent at the low level with the original output. $I|_c$ returns an input I' whose items are in I and from channel c.

Definition 3 (RI). *A program π satisfies* RI *iff*

$$\begin{aligned}\forall I, \forall\ values\ of\ val_{def} : (\pi, I) \Downarrow O \implies \exists I' :& I'|_L = I|_L \wedge \forall c, \| I'|_c \| \leq \| I|_c \| \ \wedge \\ &\wedge\ I'|_H = (\vec{df})^* \ \wedge \\ &\wedge\ (\pi, I') \Downarrow O' \wedge O'|_L = O|_L\end{aligned}$$

where $\vec{df}$ contains the default value, and $\| Q \|$ returns the length of Q.

Deletion of Inputs. DI [9] requires that if we perturb a possible trace $t = \beta.e.\alpha$ (there is no high input event in α) by deleting the high input event e, the result can be corrected into a possible trace t' ($t' = \beta'.\alpha'$). Parts β and β' and α and α' are equivalent on the low input events and the high input events; α and α' are also equivalent on low output events. In our notation, if we have an input $I = I_1.\vec{v}.I_2$, where $\vec{v}$ contains a value from a high channel ($\vec{v}[c] \neq \perp$ and $LVL[c] = H$) and in I_2 there are either no high items or only high items with default values ($I_2|_H = (\vec{df})^*$), then this input can be changed by replacing $\vec{v}$ by a default vector ($\vec{df}$). The obtained input can be sanitized by removing existing default high input items in I_2 or adding other default high input items to I_2. The sanitized queue is consumed completely by the program and the output is still low-equivalent to the original output generated with input I ($O'|_L = O|_L$).

Definition 4 (DI). *A program π satisfies* DI *iff*

$$\begin{aligned}&\forall I, \forall\ values\ of\ val_{def} : I = I_1.\vec{v}.I_2 \wedge LVL[c] = H \wedge I_2|_H = (\vec{df})^* \wedge (\pi, I) \Downarrow O \\ &\implies \exists I' : I' = I_1.I_2' \wedge I'|_L = I|_L \wedge I_2'|_H = (\vec{df})^* \wedge (\pi, I') \Downarrow O' \wedge O'|_L = O|_L\end{aligned}$$

where $\vec{v}[c] \neq \perp$ and $\vec{df}$ contains a default value.

4 Implementing the policies

```
if $a \in T_M[i][c]$ then
    input $x$ from $c$
    map($x, c, canTell(c)$)
    map($val_{def}, c, \neg canTell(c)$)
    wake($isReady(c)$)
else
    if $t \notin T_M[i][c]$ then
        map($val_{def}, c, identical(i)$)
        wake($identical(i)$)
    else
        skip
```

(a) MAP for an input from c from $\pi[i]$

```
$x := val_{def}$
if $a \in T_R[i][c]$ then
    retrieve $x$ from $(i, c)$
if $t \in T_R[i][c]$ then
    output $x$ to $c$
clean($c, identical(i)$)
wake($identical(i)$)
```

(b) REDUCE for an output to c from $\pi[i]$

T_M	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	$-$
$LVL[c] = L$	t	at

T_R	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	$-$
$LVL[c] = L$	$-$	at

Fig. 3. Implementation of NI

Non-Interference. Implementation of NI is in Fig. 3. The EM of NI on a program π needs only two local executions: the high execution ($\pi[0]$) and the low execution ($\pi[1]$). When the low execution needs a high input item, MAP sends a fake value to it. Thus, the execution of the low is independent from high input items consumed by the EM. In addition, only the low execution can send output items to low output channels. Put differently, high input items do not influence consumed low inputs and generated low outputs.

When MAP is activated on signal c from $\pi[i]$ having the ask privilege on c, MAP performs an input action, sends the real value to all local copies having the tell privilege on c, and sends a fake value to others. When MAP is activated on a signal c from $\pi[i]$ that has no privilege on c, MAP sends a fake value to $\pi[i]$ and wakes it up. Function $canTell(c) \triangleq \lambda x.t \in T_M[x][c]$ indicates whether a local copy $\pi[x]$ has the tell privilege on c. A local copy is ready to be waken up if it has received the required input item, $isReady(c) \triangleq \lambda x.EX[x].\mathsf{stt} = \mathsf{S} \wedge EX[x].\mathsf{prg} = \textbf{input } y \textbf{ from } c; \pi \wedge EX[x].\mathsf{in} = I \wedge dequeue(I, c) = (val, I') \wedge val \neq \perp$, where $dequeue(I, c) = (val, I')$ means there is an item from c in I. Function $identical()$ is defined as $identical(i) \triangleq \lambda x.x = i$.

When REDUCE is activated on a signal c from $\pi[i]$, it checks whether $\pi[i]$ has the ask privilege on c ($a \in T_R[i][c]$). If so, REDUCE gets the output value from the local output queue of $\pi[i]$. Otherwise a fake output value is used. REDUCE only sends an output value to c if $\pi[i]$ has the tell privilege on c ($t \in T_R[i][c]$). After that, the output queue of $\pi[i]$ is cleaned and $\pi[i]$ is waken.

In [10] we give a full proof that SME as identified by [5] is captured by our mechanism. In [5] soundness and precision are proved w.r.t a specific scheduler, our proof works for any scheduler respecting the configuration.

We illustrate the execution of the EM on a sample program presented in Fig. 4. The execution of this program requires confidential information about salary and bonus (at lines 2 and 5). This program does not satisfy NI since the desired salary can be sent to public channels (`evil.com` at line 7).

The execution of local executions of the EM is described in Fig. 5 with the input sequence (`cL1` $= \mathbf{T}$) (`cH1` $= M$)(`cH2` $= m$) which means that the position chosen by the applicant is "CEO", his desired salary is M, and the bonus is m. The high and the low copies execute instructions from line 1 to 7. The value

```
1 input l1 from cL1 //Get the position selected by the applicant.
2 input h1 from cH1 //Get the desired salary entered by the applicant.
3 h2 = 0
4 if l1 then //If the selected position is CEO,
5   input h2 from cH2 //Get the bonus from https://goodCompany/getBonus.
6 output h1 + h2 to cH3 //Show the income to users.
7 output h1 + h2 to cL2 //Send the income to http://evil.com/.
```

The script gets the desired position chosen by a prospective applicant from a public channel; and retrieves the desired annual salary from a confidential channel. If the chosen position is CEO, the script fetches also the annual bonus from `goodCompany/getBonus`, a confidential channel. Then, it shows the desired salary and the bonus to the applicant via `cH2`, and sends everything to `evil.com`.

Fig. 4. Running Example Program

```
1 input l1 from cL1 //Use T asked by π[1].
2 input h1 from cH1 //Get M from cH1.
3 h2 = 0;
4 if l1 then
5   input h2 from cH2 //Get m from cH2.
6 output h1 + h2 to cH3 //Send M + m to cH3.
7 ~~output h1 + h2 to cL2~~ //The output is ignored.
```

(a) The high execution $\pi[0]$

```
1 input l1 from cL1 //Get T from cL1.
2 input h1 from cH1 //The default value is used.
3 h2 = 0;
4 if l1 then
5   input h2 from cH2 //The default value is used.
6 ~~output h1 + h2 to cH3~~//The output is ignored.
7 output h1+h2 to cL2 //Send ∗ to cL3.
```

(b) The low execution $\pi[1]$

Fig. 5. Executions of local copies for NI

Input to MAP:

	0	1	2
cL1	**T**	$\perp$	$\perp$
cH1	$\perp$	M	$\perp$
cH2	$\perp$	$\perp$	m

Output by REDUCE:

	0	1	2	3	4
cH3	$\perp$	$\perp$	$\perp$	$M+m$	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	$*$

Local Executions:

High execution $\pi[0]$:

Local input:

cL1	**T**	$\perp$	$\perp$
cH1	$\perp$	M	$\perp$
cH2	$\perp$	$\perp$	m

Local output:

cH3	$\perp$	$\perp$	$\perp$	$M+m$	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	●

Low execution $\pi[1]$:

Local input:

cL1	**T**	$\perp$	$\perp$
cH1	$\perp$	$*$	$\perp$
cH2	$\perp$	$\perp$	$*$

Local output:

cH3	$\perp$	$\perp$	$\perp$	●	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	$*$

● is an output value that is ignored. $*$ is a default value or is calculated based on default values.

Fig. 6. Input and output queues for NI

generated by the output instruction of the high copy (resp. the low copy) at line 7 (resp. line 6) is ignored. To facilitate the presentation we present the contents of the global and local input and output queues in Fig. 6. The global input queue is consumed completely by the execution of the EM. The values sent to `cH3` and `cL2` are respectively $M + m$ and $*$, where $*$ denotes values calculated based on default values. Each column in the table corresponds to an input/output operation. Input and output tables should be read from left to right; columns describe the input/output to each channel at time $t = 0$, $t = 1$, etc.

Non-Deducibility. The configuration of the mechanism of ND requires three local copies. The low execution ($\pi[2]$) can consume only low input items and generate low output item. The high execution ($\pi[0]$) can consume real values from all channels and can send high output items to the environment. The purpose of the shadow execution ($\pi[1]$) is to make sure that low inputs do not determine high inputs. Indeed the shadow execution is

T_M	$\pi[0]$	$\pi[1]$	$\pi[2]$
$LVL[c] = H$	t	at	$-$
$LVL[c] = L$	t	$-$	at

T_R	$\pi[0]$	$\pi[1]$	$\pi[2]$
$LVL[c] = H$	at	$-$	$-$
$LVL[c] = L$	$-$	$-$	at

Fig. 7. Impl. of ND

the only one that can ask for high inputs but only receives dummy low inputs. We used the word shadow as its output are ignored (only legitimate high output from the high is going to see the light). In other words, the low inputs and the high inputs consumed by the EM are independent from each other.

The programs of MAP and REDUCE are the same as the ones of NI. Privileges of the low execution are the same as those of the low execution of NI. The only difference is that the high execution can be told but cannot ask input values and can output its values to high output channels. The shadow execution is the only one that can ask for high input items. Fig. 7 shows configuration of T_M and T_R. Our EM is slightly stronger as it will generate the correct low output even if the high execution might not terminate.

Removal of Inputs. The configuration of RI is in Fig. 8. The EM of RI is similar to the one of NI except the way of handling signals on high input channels from the low execution ($\pi[1]$). To ensure the existence of I' as in the definition, MAP is allowed to ask high input items for the low execution. To ensure that the behaviors visible to attackers do not change, the low execution receives only default high input items and only it can send outputs to low output channels.

```
if $a \in T_M[i][c]$ then
    input $x$ from $c$
    map($x, c, canTell(c)$)
    map($val_{def}, c, \neg canTell(c)$)
    wake($isReady(c)$)
else
    skip
```

(a) MAP for an input from c from $\pi[i]$

T_M	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	a
$LVL[c] = L$	t	at

Fig. 8. Implementation of RI

The configuration table T_M is similar to the one of NI except that the low execution has the ask privilege on high input channels. The MAP program is also similar to the one of RI except the cases of handling signals from the low execution on high input channels. In these cases, MAP performs an input action, sends the read value to the high, and send a default value to the low. Functions $canTell(c)$, $isReady(c)$ and $identical(i)$ are as in the ones in NI.

Deletion of Inputs. DI is enforced with the idea that whenever the high execution ($\pi[0]$) requests a high input item, this execution will be cloned. The clones have to reuse low input items asked by the low execution ($\pi[1]$), will not receive real values from high channels and cannot send output to the environment. As in NI, the low execution can only receive fake high input values.

Implementation of EM of DI is presented in Fig. 9.

```
if $LVL[c] == H$ and $i == 0$ then
    clone($identical(i), PRIV_{T_M}, PRIV_{T_R}$)
if $a \in T_M[i][c]$ then
    if $t \in T_M[i][c]$ then
        input $x$ from $c$
        map($x, c, canTell(c)$)
        map($val_{def}, c, \neg canTell(c)$)
        wake($isReady(c)$)
    else
        map($val_{def}, c, identical(i)$)
        wake($identical(i)$)
else
    skip
```

(a) MAP for DI for an input from c from $\pi[i]$

T_M	$\pi[0]$	$\pi[1]$	$\pi[i]>1$
$LVL[c]=H$	at	$-$	$-$
$LVL[c]=L$	t	at	t

T_R	$\pi[0]$	$\pi[1]$	$\pi[i]>1$
$LVL[c]=H$	at	$-$	$-$
$LVL[c]=L$	$-$	at	$-$

Fig. 9. Implementation of DI

$$\text{INP}\ \frac{\pi = \textbf{input}\ x\ \textbf{from}\ c \qquad I = \vec{v}.I' \qquad \vec{v}[c] \neq \bot}{\begin{array}{c}\mathit{\Delta}, \mathsf{prg}{:}\pi, \mathsf{mem}{:}m, \mathsf{in}{:}I \\ \Rightarrow \mathit{\Delta}, \mathsf{prg}{:}\textbf{skip}, \mathsf{mem}{:}m[x \mapsto \vec{v}[c]], \mathsf{in}{:}I'\end{array}} \qquad \text{OUTP}\ \frac{\pi = \textbf{output}\ e\ \textbf{to}\ c \qquad \vec{v} = \vec{\bot}[c \mapsto m(e)]}{\begin{array}{c}\mathit{\Delta}, \mathsf{prg}{:}\pi, \mathsf{out}{:}O \\ \Rightarrow \mathit{\Delta}, \mathsf{prg}{:}\textbf{skip}, \mathsf{out}{:}O.\vec{v}\end{array}}$$

Fig. 10. Semantics of input and output instructions of programs

The program of REDUCE is identical to the one in NI. The EM of DI requires more than two local executions. Only the high execution $\pi[0]$ can ask for and get the high input items, other local executions will use default values. Each time the high execution asks a high input item, it is cloned. In Fig. 9 the configuration of the clones for input and output is presented in respectively T_M and T_R in the columns with title $\pi[i] > 1$; These columns are the privilege templates for $PRIV_{T_M}$ and $PRIV_{T_R}$ in clone instruction in Fig. 9a. As in NI, only the low execution $\pi[1]$ can ask for low input items and generate low output items; other local executions will reuse the low input items retrieved by the low execution. Functions $canTell(c)$, $isReady(c)$ and $identical(i)$ are as in the ones in NI.

5 Semantics

Semantics of controlled programs. Our model language is close to the one used in the SME paper [5]. Valid values in this language are boolean values (**T** and **F**) or non-negative integers. A program π is an instruction described in Fig. 2a where π, e, x, and c are meta-variables for respectively instructions, expressions, variables, and input/output channels. Since a program is just a sequence of instructions (i.e. a complex instruction itself), we will use program and instruction interchangeably when referring to complex instructions. We model an input (output) item as a vector $\vec{v}$ and define input (output) of program instances as queues I, O so that $\vec{v}.I$ (resp. $\vec{v}.O$) adds the element $\vec{v}$ to the queue. We use vectors of channel to accommodate forms in which multiple fields are submitted simultaneously but are classified differently (e.g. credit card numbers vs. user names). Given a vector $\vec{v}$ and a channel c, the *value of the channel* is denoted by $\vec{v}[c]$. To simplify the formal presentation, in the sequel w.l.o.g. we assume that each input and output operation only affect one channel at a time. Thus, for each vector, there is only one channel c such that $\vec{v}[c] \neq \bot$.

To define an execution configuration, we use a set of labelled pairs. A labelled pair is composed by a label and an object and in the form label:*object*. The label is attached to the *object* in order to differentiate this object from others, so each label occurs only once. An *(execution) configuration* of a program is a set $\{\mathsf{prg}{:}\pi, \mathsf{mem}{:}m, \mathsf{in}{:}I, \mathsf{out}{:}O\}$, where π is the program to be executed, m is the memory (a function mapping variables to values), I (resp. O) is the queue of input (resp. output) vectors. The operational semantics of the input and output instructions of the model language is the natural one. Fig. 10 illustrates some examples. See also [5] for similar one and [10] for detail. The conclusion part of each semantic rule is written as $\mathit{\Delta}, \mathit{\Gamma} \Rightarrow \mathit{\Delta}, \mathit{\Gamma}'$, where $\mathit{\Delta}$ denotes the elements of the execution configuration that are unchanged upon the transition. We abuse the notation $m(.)$ and use it to evaluate expressions to values. When an output command sends a value to the channel c, an output vector $\vec{v} = \vec{\bot}[c \mapsto val]$ is inserted into the output queue, where $\vec{v}$ is the vector with all undefined channels, except c that is mapped to $m(e)$, so $\vec{v}[c'] = \bot$ for all $c' \neq c$ and $\vec{v}[c] = m(e)$.

Semantics of the Enforcement Mechanism. A *configuration of an EM* is a set $\{\mathsf{t_m}{:}T_M, \mathsf{t_r}{:}T_R, \mathsf{top}{:}TOP, \mathsf{map.prg}{:}\pi_M, \mathsf{map.mem}{:}m_M, \mathsf{red.prg}{:}\pi_R, \mathsf{red.mem}{:}m_R, \mathsf{in}{:} I, \mathsf{out}{:}O, \bigcup_i \mathsf{LECS}_i\}$, where T_M and T_R are configuration tables for respectively MAP and REDUCE, TOP is the index of the top of the stack of configurations of local executions EX, π_M and m_M (resp. π_R and m_R) are the program to be executed and the memory of MAP (resp. REDUCE), I and O are respectively the input and output queues of the EM, and LECS_i is the configuration of the i-th local execution.

For the initial configuration, all local input and output queues will be empty, all local executions will be in the executing state, and skip is the only instruction in MAP and REDUCE programs. The EM terminates when all local executions, MAP and REDUCE programs terminate, and the global input queue is consumed completely.

The semantics of EM is the interleaving of concurrent atomic instructions of the various programs so each transition rule either by a local execution, by MAP, or by REDUCE is a step of the EM as a whole.

Local Executions. Each local execution is identified by a unique identifier i, which is its number on stack EX. A local copy can be in one of two states: **E** (Executing) or **S** (Sleeping). A local copy moves from **E** to **S** when it needs an input item that is not available in its local queue or when it generates an output item. A local copy moves from **S** to **E** when the required input item is ready or its output item is consumed.

A configuration of i-th local copy is $\mathsf{LECS}_i \triangleq \{EX[i].\mathsf{stt} : st, EX[i].\mathsf{int} : s, EX[i].\mathsf{prg} : \pi, EX[i].\mathsf{mem} : m, EX[i].\mathsf{in} : I, EX[i].\mathsf{out} : O\}$, where st is its state, s is a signal, π, m, I, and O are as in configuration of controlled programs, EX is the global stack of local execution. The initial configuration of i-th local copy is $\{EX[i].\mathsf{stt}{:}\mathbf{E}, EX[i].\mathsf{int}{:}\bot, EX[i].\mathsf{prg}{:}\pi, EX[i].\mathsf{mem}{:}m_0, EX[i].\mathsf{in}{:}\epsilon, EX[i].\mathsf{out}{:}\epsilon\}$. A local copy terminates if there is only a skip instruction to be executed.

The semantics of assignment, composition, if, while, skip instructions is essentially identical to the one of the controlled programs. The only difference is the explicit condition that the local state must be **E**. When the input instruction of $\pi[i]$ is executed and the required input item is not in the local input queue ($dequeue(I, c) = (\bot, I')$), $\pi[i]$ emits a signal c and moves to a sleep state (rule LINP2 in Fig. 11). Otherwise, the first available item will be consumed. A signal c is generated when the output instruction is executed (rule LOUTP in Fig. 11).

MAP. In addition to the instructions in Fig. 2a (except the output instruction), the program π_M is also composed by instructions in Fig. 2b, where $PRED[\,] \triangleq$

$$\text{LINP2}\ \frac{EX[i].\mathsf{stt} = \mathbf{E} \qquad EX[i].\mathsf{prg}{:}\pi = \mathbf{input}\ x\ \mathbf{from}\ c \qquad dequeue(I, c) = (\bot, I')}{\Delta, EX[i].\mathsf{stt}{:}\mathbf{E}, EX[i].\mathsf{int}{:}\bot \Rightarrow \Delta, EX[i].\mathsf{stt}{:}\mathbf{S}, EX[i].\mathsf{int}{:}c}$$

$$\text{LOUTP}\ \frac{EX[i].\mathsf{stt} = \mathbf{E} \qquad \pi = \mathbf{output}\ e\ \mathbf{to}\ c \qquad EX[i].\mathsf{mem} = m \qquad \vec{v} = \vec{\bot}[c \mapsto m(e)]}{\begin{array}{c}\Delta, EX[i].\mathsf{stt}{:}\mathbf{E}, EX[i].\mathsf{int}{:}\bot, EX[i].\mathsf{prg}{:}\pi, EX[i].\mathsf{out}{:}O \\ \Rightarrow \Delta, EX[i].\mathsf{stt}{:}\mathbf{S}, EX[i].\mathsf{int}{:}c, EX[i].\mathsf{prg}{:}\mathbf{skip}, EX[i].\mathsf{out}{:}O.\vec{v}\end{array}}$$

Fig. 11. Semantics of input and output instructions of controlled $\pi[i]$

$$\text{MAP}\ \frac{\begin{array}{c}\pi_M = \mathbf{map}(e, c, PRED[\ ]) \qquad m = \mathsf{map.mem} \qquad S = \{i \in \{0, \ldots, TOP\} : PRED[i]\} \\ \mathsf{LECS} = \bigcup_{i \in S} \{EX[i].\mathsf{in}{:}I\} \qquad \vec{v} = \vec{\bot}[c \mapsto m(e)] \qquad \mathsf{LECS}' = \bigcup_{i \in S} \{EX[i].\mathsf{in}{:}I.\vec{v}\}\end{array}}{\Delta, \mathsf{map.prg}{:}\pi_M, \mathsf{LECS} \Rightarrow \Delta, \mathsf{map.prg}{:}\mathbf{skip}, \mathsf{LECS}'}$$

$$\text{RETR}\ \frac{\pi_R = \mathbf{retrieve}\ x\ \mathbf{from}\ (i, c) \qquad EX[i].\mathsf{out} = O \qquad dequeue(O, c) = (val, O') \qquad val \neq \bot}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{red.mem}{:}m \Rightarrow \Delta, \mathsf{red.prg}{:}\mathbf{skip}, \mathsf{red.mem}{:}m[x \mapsto val]}$$

Fig. 12. Semantics of map and retrieve instructions of MAP and REDUCE

$\lambda x.Pred(x)$ is a meta-variable for predicates. The evaluation of the predicate $PRED[\ ]$ on $\pi[i]$ is denoted as $PRED[i]$.

The execution of map, wake, or clone instruction is applied simultaneously to all local executions $\pi[i]$ such that $PRED[i]$ is true. For map, the value of expression e (which is considered from c) is sent to the input queues of all $\pi[i]$. The semantics of map instruction is described in Fig. 12. For wake, all local executions $\pi[i]$ are awaken and interrupt signals in their configurations are removed. For clone, the configuration of each $\pi[i]$ is cloned. The list $PRIV_{T_M}$ (resp. $PRIV_{T_R}$) is an input (resp. output) privilege template for clones which varies depending on the enforced property. We give an example of such templates in §4, where the enforced property requires cloning.

The initial configuration of MAP is $\{\mathsf{map.prg}{:}\pi_M, \mathsf{map.mem}{:}m_0\}$. The execution of MAP terminates if skip is the only instruction in the MAP program. MAP is activated when the previous execution of MAP has terminated, and there is a local execution asking for help for an input item.

REDUCE. Except the input instruction, in addition to the instructions in Fig. 2a, the program of REDUCE may contain instructions in Fig. 2c. The execution of retrieve instruction reads the value from the output queue of $\pi[i]$ and stores it into x. The execution of clean instruction is applied to all $\pi[i]$ such that $PRED[i]$ is true. This instruction removes the first output item to c from O of $\pi[i]$. The execution of the wake instruction is similar to the one of MAP. Configuration, activation and termination of REDUCE are similar to the ones of MAP. The semantics of retrieve instruction is shown in Fig. 12 where $dequeue(O, c)$ returns a first item to c in O.

6 Formal Properties

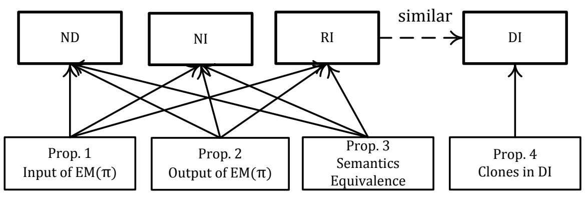

Fig. 13. Proof Strategy for Soundness

[The full versions of the proofs are available in [10].] The soundness property states that the EM correctly enforces the desired policy on an arbitrary program. Our notion of soundness is taken from [5, 4] and is close to the one used in [8]. It has some known limitations (see [11] for a different definition) but we retained it because it is widely used and understood. Soundness does not hold for EMs of termination-sensitive properties because one local copy might terminate but the others might not. Thus, the whole EM does not terminate.

Theorem 1 (Soundness of Enforcement). *For all programs π, each EM executed on π in Tab. 1 satisfies the corresponding policy, except for termination-sensitive policies.*

The proof strategy of soundness is sketched in Fig. 13. Prop. 1 states that the input handling in MAP is correct w.r.t. the specification: e.g., we prove that for NI, MAP only asks input items from the environment for high input requests from the high execution. Prop. 2 states that the output handling by REDUCE is correct w.r.t. to the specification: e.g., only the high execution sends items to high output channels. Prop. 3 states that the semantics of controlled programs and the semantics of local executions are equivalent (for I_1 and I_2, which coincide for all channels, the execution of the original program on I_1 and the execution of a local copy on I_2 yield the same output queues).

To prove the soundness theorem for NI and ND we perform case-based reasoning showing that outputs produced by EMs satisfies the respective definitions. This proof strategy is also used to prove the soundness theorem for RI. For DI, we need another proposition (Prop. 4) stating that the clones do not influence the consumed inputs and the generated outputs of the EM.

The notion of precision for enforcement of a property is taken from [5, 4]. The intuition is that the EM does not change the visible behavior of a program that is secure with respect to the property (and in particular the I/O behaviour on specific channels).

Definition 5. *An EM is* precise *w.r.t a property, if for any program π satisfying the property, and for every input I, where $(\pi, I) \Downarrow O$, the actually consumed input I^* and the actual output O^* of the EM, regardless of the order of executing local copies, are s.t. EM terminates and $I^*|_c = I|_c$ and $O^*|_c = O|_c$ for all channels c.*

Theorem 2 (Precision of Enforcement). *Each EM in Tab. 1 is precise w.r.t. the corresponding policy except for termination-insensitive policies.*

Fig. 14 shows the proof strategy for precision. We prove simple properties regarding the correct handling of interrupt signals (Prop. 5 and Prop. 6). We show that from the input of the high execution we can reconstruct the original global input (Prop. 7). The proof of the precision theorem of the EM of NI (resp. ND) follows directly from Lem. 1 (resp. Lem. 2). Lem. 1 shows that if a program π satisfies TSNI, terminates, and all local executions consume input correctly, then the consumed input of the mechanism is I^* where $I|_c = I^*|_c$ for all c. The proofs of the precision theorem of EMs of RI and DI are similar.

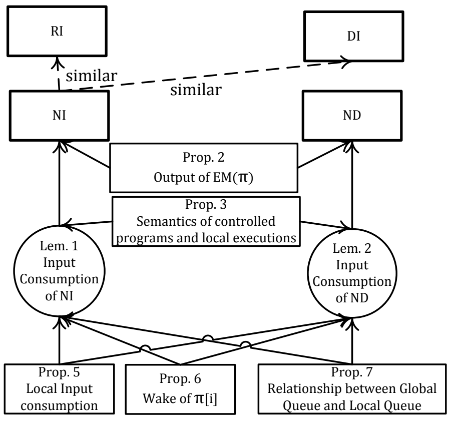

Fig. 14. Proof Strategy for Precision

Precision does not hold for mechanisms of termination-insensitive properties. For a program satisfying a termination-insensitive property, its execution on an input might terminate, while execution on the other inputs as in the definition of the property might not. Thus, there is a case that the high copy might terminate but other executions might not.

7 Related Works and Conclusions

Information flow policies can be enforced by many approaches [13, 12, 3]. Our choice of using the multi-execution approach, despite its performance overhead, was dictated by its advantages over the static and dynamic information flow analysis techniques. Furthermore, the multi-execution approach is also practical as demonstrated in [4], where SME, an instance of this approach, is implemented in FireFox. The implementation introduces a noticeable performance overhead but not prohibitive and the implementation works with most existing web sites.

SME [5] has inspired many researchers to push further investigation of this technique. The influence of the order of executing local copies on timing and termination channels is investigated in [8]. Stronger notions of precision are investigated in [11, 16]. Our current proposal does not address timing and termination channels, and does not offer the same precision guarantees. However, our proposal can be further extended by using the techniques proposed in [8, 11, 16]. The focus of our paper is to develop a programmable framework that is capable of handling different information flow properties.

SME-based EMs of declassification policies are proposed in [1, 11]. Our framework can be instantiated to enforce stateless declassification policies like the one in [11] where the existence of the high input items can be released. The configuration of this policy is similar to the one of RI except that the low does not have the ask privilege on high input channels. To enforce stateful declassification policies in which the physical locations of release are specified [14], one possible approach is to introduce declassify operators as in [1, 11]. However, by doing this we lose one advantage of SME which treats controlled programs as black boxes.

We presented a programmable framework that can enforce multiple information flow properties via running several copies of a program. The framework is instantiated for enforcing non-interference (NI) [5], non-deducibility (ND) [15], removal of inputs (RI) and deletion of inputs (DI) [9]. For these properties we formally proved soundness and precision of enforcement.

The framework uses the MAP and REDUCE components to interact with the environment: all input and output actions are mediated by these two components. Local executions consume different inputs (real input values or default ones) fed by MAP, depending on their privileges in the table T_M; for each channel the outputs are fetched by REDUCE from the dedicated execution (which has the corresponding privilege in the table T_R).

Acknowledgements This work is partly supported by the project EU-IST-NOE-NESSOS.

References

1. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. *SIGPLAN Not.*, 47(1):165–178, Jan. 2012.
2. M. Balliu, M. Dam, and G. L. Guernic. Encover: Symbolic exploration for information flow security. In *Proc. of CSF 2012*, pages 30–44, 2012.
3. G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Proc. of FMOODS/FORTE 2012*, 2012.
4. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proc. of CCS 2012*, 2012.
5. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of IEEE S&P 2010*, 2010.
6. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. of Comp. Sec.*, 3:5–33, 1994.
7. J. Goguen and J. Meseguer. Security policies and security models. In *Proc. of IEEE S&P'82*, 1982.
8. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE S&P*, 2011.
9. H. Mantel. Possibilistic definitions of security - an assembly kit. In *Proc. of CSFW 2000*, 2000.
10. M. Ngo, F. Massacci, and O. Gadyatskaya. MAP-REDUCE runtime enforcement of information flow policies. Availabe as ArXiv report `http://arxiv.org/abs/1305.2136`. Technical Report DISI-13-019, University of Trento, 2013.
11. W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *Proc. of CSF 2013*, 2013.
12. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of CSF 2010*, 2010.
13. A. Sabelfeld and A. Myers. Language-based information-flow security. *J. on Selected Areas in Comm.*, 21(1):5 – 19, 2003.

14. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. on Comput. Secur.*, 17(5):517–548, Oct. 2009.
15. D. Sutherland. A model of information. In *Proc. of NCSC'86*, 1986.
16. D. Zanarini, M. Jaskelioff, and A. Russo. Enforcement of confidentiality for reactive systems. In *Proc. of CSF 2013*, 2013.