MAP-REDUCE Enforcement Framework of Information Flow Policies

Minh Ngo, Fabio Massacci, and Olga Gadyatskaya

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

Abstract. We propose a flexible framework that can be easily customized to enforce a large variety of information flow properties. Our framework combines the ideas of secure multi-execution and map-reduce computations. The information flow property of choice can be obtained by simply changes to a map (or reduce) program that control parallel executions.
We present the architecture of the enforcement mechanism and its customizations for non-interference (NI) (from Devriese and Piessens) 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: Runtime enforcement, information flow, secure multi-execution

1 Introduction

Information flow properties define the acceptable behaviours of computer programs with respect to allowed and forbidden flows of information. The most well-known information flow property is *non-interference* (NI), which roughly requires that the input data classified as confidential (also called secret, or high) should not influence the public (low) outputs [8, 7].

By weakening or strengthening the definition of NI, security researchers have proposed different information flow properties [15–17, 23]. For instance, the definition of NI in [8] assumes that if there is no high input, then there is no high output. Yet, this assumption does not always hold. In [17], the *generalized non-inference* (GNF) property is defined for systems that generate high outputs even if there are no high inputs.

To the best of our knowledge, there is no proposal in the literature with a unified approach to the enforcement of multiple information flow properties. The existing enforcement mechanisms (e.g. [2, 5, 7, 14, 22]) can be configured to accommodate different information flow policies that identify what is confidential and what is public, and what are the authorized flows in the security lattice [7, 20], and, sometimes, they can as well enforce declassification policies [1] (e.g. [1]).

[1] These policies are required when one needs to disclose information that depends on confidential data in some way, see e.g. [21] for details.

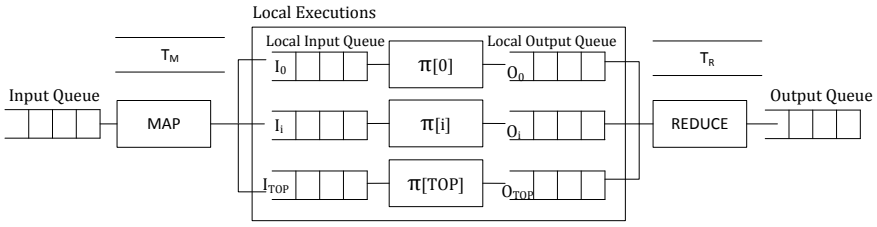

Fig. 1: Architecture of enforcement mechanisms

Yet, the adaptation of an existing enforcement mechanism (for example, for NI) to enforce another property (for instance, GNF) is not straight-forward.

We aim to fill this gap by providing an enforcement framework that can be extended by different information flow properties. The framework is inspired by the MAP-REDUCE approach from Google [12]; and generalizes the secure multi-execution (SME) technique proposed by Devriese and Piessens in [7] so that it can enforce other information flow properties, e.g. properties from [15]. The main idea is to execute multiple "local" instances of the original program, feeding different inputs to each instance of the program. The local inputs are produced from the original program inputs by the MAP component, depending on the security levels and the input channels. Upon receiving the necessary data (for instance, after each individual program instance is terminated), the REDUCE component collects the local outputs and generates the common output, thus ensuring that the overall execution is secure. MAP and REDUCE are customizable and by changing their programs the user can easily change the enforced property. Two simple tables (T_M and T_R) tell MAP and REDUCE what they should do when receiving respectively input and output requests from local executions on a channel. Table 1 summarizes the configurations for some sample properties.

Tab. 1: Enforcement mechanisms for the selected information flow properties

Property	Section	Components		
		MAP	REDUCE	T_M/T_R
Removal of inputs [15]	§5.1	Fig.9c	Fig.9d	Fig.9a,9b
Deletion of inputs [15]	§5.2	Fig.12c	Fig.9d	Fig.12a,12b
Termination (in)sensitive non-interference [7]	§5.3	Fig.13	Fig.9d	Fig.9a,9b

The rest of the paper is organized as follows. §2 overviews the idea of our approach and the architecture of the enforcement framework; §3 introduces the semantics of controlled programs. §4 presents the formalization of the framework; §5 describes the enforcement mechanisms for the chosen information flow properties. Soundness and precision of the constructed enforcement mechanisms are postulated in §6. We discuss future extensions of the framework in §7 and its limitations in §8. Then we discuss the related work in §9 and conclude in §10.

2 Overview

Fig. 1 depicts the general architecture of the enforcement mechanism for an information flow property on a program π. It is composed by a stack EX of local executions ($\pi[0], \ldots, \pi[TOP]$, where TOP is the index of the top of the stack), global input and output queues, the MAP and REDUCE components, and the tables T_M and T_R.

Local executions (instances of the original program that are executed in parallel and are unaware of each other) are separated from the environment

input and output actions by the enforcement mechanism. A local execution has its own input and output queues. 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 (received from the user or other input channels), and the global output queue containing the output items filtered by the enforcement mechanism 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* (or just *signal* for short). When different local executions request values from the same channel, there will be only one actual input action performed by the enforcement mechanism. After the value is read, MAP will distribute it to local executions, replacing the actual value by the default (fake) one, if necessary. Similarly, when a local execution generates an output item, the output item will be handled by REDUCE.

MAP and REDUCE can also autonomously send and, respectively, collect items from local queues. For example, upon receiving an input item from the environment, MAP can send it to *all* local executions that satisfy a predicate. The parallel broadcast and parallel collection to and from local processors are the characteristic features of MAP-REDUCE programs [12]; this explains our choice for the name of the enforcement mechanism.

The actions of MAP (respectively REDUCE) on an input (output) request from a local execution depend on the configuration information in the table T_M (T_R). These components of the enforcement mechanism are customized depending on the desired information flow property. The framework components configured to implement the chosen information flow properties are listed in Tab. 1 (for each selected property the table contains pointers to the actual component configurations).

The configuration of input and output actions of local executions is based on two privileges: *ask* (a) and *tell* (t). 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 the interrupt signal from a local execution. If a local execution has the *tell* privilege on the input channel c, then this local execution can get the real value from the channel c when MAP broadcasts the input item to local executions, otherwise it will get a default value. If a local execution has the *ask* privilege on the output channel c, then REDUCE will actually ask the execution for the real value that it wished to send to c. Otherwise, REDUCE will just replace it with a default value. If a local execution has the *tell* privilege on the output channel c, it can invoke REDUCE to send the values generated by itself to c.

Notice that an execution may have only one privilege. For example, an execution with the *ask* but not the *tell* privilege in T_R will provide the real value to REDUCE, but will not be able to invoke REDUCE to put the value in the external output. It will have to wait for somebody else with the *tell* privilege on the channel to produce an output.

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

Fig. 3: Semantics of the input and output instructions of controlled programs

3 Semantics of Controlled Programs

Our model programming language is close to the one used in the SME paper [7]. Valid values in this language are boolean values (**T** and **F**) or non-negative integers. A program π is an instruction composed from the terms described in Fig. 2. In this figure π, e, x, and c are meta-variables for instructions, expressions, variables, and input/output channels respectively.

$\pi ::=$	*instructions* :
$\quad \mid x := e$	*assignment*
$\quad \mid \pi; \pi$	*sequence*
$\quad \mid \textbf{if}\ e\ \textbf{then}\ \pi\ \textbf{else}\ \pi$	*if*
$\quad \mid \textbf{while}\ e\ \textbf{do}\ \pi$	*while*
$\quad \mid \textbf{skip}$	*skip*
$\quad \mid \textbf{input}\ x\ \textbf{from}\ c$	*input*
$\quad \mid \textbf{output}\ e\ \textbf{to}\ c$	*output*

Fig. 2: Language instructions

We model an input (output) item as a vector and define input (output) of program instances as queues. 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). An *input vector* $\vec{v}$ is a mapping from input channels to their values, $\vec{v} : C_{in} \to \Sigma \cup \{\perp\}$, where Σ is the set of all non-negative integer and boolean values, and the value $\perp$ is the special undefined value. An *output vector* $\vec{v}$ is a mapping from output channels to their values, $\vec{v} : C_{out} \to \Sigma \cup \{\perp\}$.

Given a vector $\vec{v}$ and a channel c, the *value of the channel* is denoted by $\vec{v}[c]$. The symbol $\vec{\perp}$ denotes a vector mapping all channels to $\perp$. 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 $\vec{v}[c] \neq \perp$.

Let queue Q be a sequence of elements $q_1 \ldots q_n$. We denote the addition of a new element to the queue Q as $Q.q$, or $q_1 \ldots q_n.q$; the removal of the first element from the queue Q is denoted by $q_2 \ldots q_n$. By ϵ we denote an empty queue.

To define an execution configuration, we use a set of labelled pairs. A labelled pair is composed by a label and an object and is written in the form of label: *object*. The label is attached to the *object* to differentiate this object from the 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 instruction to be executed, m is the memory (a function mapping variables to values), I (O, respectively) is the queue of input (output) vectors.

The operational semantics of the input and output instructions of the model language, as the most interesting, is described in Fig. 3. The conclusion part of each semantic rule is written as $\Delta, \Gamma \Rightarrow \Delta, \Gamma'$, where Δ denotes the elements of the execution configuration that are unchanged upon the transition. The semantics of the comma "," in the expression Δ, Γ is the disjoint union of Δ and Γ. We abuse the notation of the memory function $m(.)$ and use it to evaluate expressions to values. When an output command sends a value of e to the channel c, an output vector $\vec{v} = \vec{\perp}[c \mapsto m(e)]$ 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 $\vec{v}[c'] = \perp$ for all $c' \neq c$ and $\vec{v}[c] = m(e)$. For the lack of space we do not provide

$$\text{LINP1}\ \frac{EX[i].st = \mathbf{E} \qquad \pi = \mathbf{input}\ x\ \mathbf{from}\ c \qquad dequeue(I, c) = (val, I') \qquad val \neq \bot}{\Delta, EX[i].\mathsf{prg}{:}\pi, EX[i].\mathsf{mem}{:}m, EX[i].\mathsf{in}{:}I \Rightarrow \Delta, EX[i].\mathsf{prg}{:}\mathbf{skip}, EX[i].\mathsf{mem}{:}m[x \mapsto val], EX[i].\mathsf{in}{:}I'}$$

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

Fig. 4: Semantics of the input instruction of $\pi[i]$

the other rules; the interested reader can find the semantics of the controlled programs and the framework components in the full version of this paper [18].

An *execution* of the program π is a finite sequence of configuration transitions $\gamma_0 \rightarrowtriangle \gamma_1 \rightarrowtriangle \ldots \rightarrowtriangle \gamma_k$, where $\gamma_0 = \{\mathsf{prg}:\pi, \mathsf{mem}:m_0, \mathsf{in}:I, \mathsf{out}:\epsilon\}$ is the initial configuration, m_0 is the function mapping every variable to the initial value, and k is the number of transitions. The transition sequence can be also written as $\gamma_0 \rightarrowtriangle^* \gamma$ if the exact number of transitions does not matter. The program *terminates* if there exists a configuration $\gamma_f = \{\mathsf{prg}:\mathbf{skip}, \mathsf{mem}:m, \mathsf{in}:\epsilon, \mathsf{out}:O\}$ such that $\gamma_0 \rightarrowtriangle^* \gamma_f$. We denote the whole derivation sequence by $(\pi, I) \Downarrow O$ using the big step notation.

4 Semantics of the Enforcement Mechanism

We now specify the semantics of the enforcement mechanism components: local executions, the programs of MAP and REDUCE. The general approach is that execution of parallel programs is modeled by the interleaving of concurrent atomic instructions [13] so each transition rule either by a local execution, by MAP, or by REDUCE is a step of the enforcement mechanism as a whole.

Local executions. Each local execution is associated with a unique identifier i, that is its number on the stack EX. A local execution can be in one of the two states: **E** (Executing) or **S** (Sleeping). Initially, the state of all local executions is **E**. A local execution moves from **E** to **S** when it has sent an interrupt signal to require an input item that is not ready in its local input queue, or to signal that it has generated an output item. A local execution moves from **S** to **E** when it is awoken by the MAP component (the input item it required is ready) or by the REDUCE component (its output item is consumed). The semantics for the local execution instructions is the direct adaptation of the controlled programs semantics with catering for the switch to and from the **S** state when doing inputs.

We provide the rules for input instruction in Fig. 4, where $dequeue(Q, c)$ returns the first value from the channel c in the queue Q and the rest of the queue. When the input instruction is executed and the input item required is in the local input queue, this item will be consumed (rule LINP1). Otherwise, the local execution emits an input interrupt signal c and moves to the sleep state (rule LINP2).

An *execution configuration of a local execution* $\pi[i]$ is a set $\mathsf{LECS}_i \triangleq \{EX[i].\mathsf{stt}{:}st, EX[i].\mathsf{int}{:}signal, EX[i].\mathsf{prg}{:}\pi, EX[i].\mathsf{mem}{:}m, EX[i].\mathsf{in}{:}I, EX[i].\mathsf{out}{:}O\}$, where EX is the global stack of local execution, i denotes the i-th execution, st is the state of the local execution, $signal$ is the interrupt signal sent by the local execution, π is an instruction to be executed, m is the memory, and I and O are queues of input and output vectors respectively.

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

$$\text{CLNE}\ \frac{\pi_M = \mathbf{clone}(PRED[\,], PRIV_{T_M}, PRIV_{T_R}) \qquad S = \{i \in \{0, \ldots, TOP\} : PRED[i]\} \\ \mathsf{LECS} = \bigcup_i \mathsf{LECS}_i \qquad \mathsf{LECS}' = \mathsf{LECS} \cup \bigcup_{i \in S} fork(\mathsf{LECS}_i, TOP + assignIndex(i)) \\ TOP' = TOP + |S| \qquad (T'_M, T'_R) = assign(T_M, T_R, TOP, TOP', PRIV_{T_M}, PRIV_{T_R})}{\Delta, \mathsf{t_m}{:}T_M, \mathsf{t_r}{:}T_R, \mathsf{top}{:}TOP, \mathsf{map{:}prg{:}}\pi_M, \mathsf{LECS} \Rightarrow \Delta, \mathsf{t_m}{:}T'_M, \mathsf{t_r}{:}T'_R, \mathsf{top}{:}TOP', \mathsf{map{:}prg{:}}\mathbf{skip}, \mathsf{LECS}'}$$

Fig. 6: Semantics of the MAP instructions **map** and **clone**

MAP. A MAP program is normally composed of three steps: the input retrieval step, the value distribution step and the wake up step. In the first step, an input item is fetched by performing an actual input action from the specified channel, or by using the default value (val_{def}). In the second step, a real input item or the default item is sent to local executions. These two steps depend on the configuration in T_M. In the third step, local executions are awoken if a certain condition is satisfied, e.g., these local executions were waiting for input items and they have received the input items they required.

In addition to the instructions in Fig. 2 (except for the output instruction replaced by the map instruction), π_M may also contain the instructions described in Fig. 5, where $PRED[\,] \triangleq \lambda x.Pred(x)$ is a meta-variable for predicates. Evaluation of the predicate $PRED[\,]$ on the configuration of the local execution $\pi[i]$ is denoted as $PRED[i]$.

$\pi_M ::= \ldots$	*instructions* :
$\mid \mathbf{map}(e, c, PRED[\,])$	*map*
$\mid \mathbf{wake}(PRED[\,])$	*wake*
$\mid \mathbf{clone}(PRED[\,], PRIV_{T_M}, PRIV_{T_R})$	*clone*

Fig. 5: The MAP instructions

The execution of map, wake, or clone instruction is applied simultaneously to all local executions $\pi[i]$ such that $PRED[i]$ is true as follows. The execution of the map instruction sends the value of the expression e to the input queues of all local executions. The value sent is considered as a value from the channel c. The execution of the wake instruction wakes all local executions $\pi[i]$ up and removes the interrupt signals generated by those local executions (if there were some). The execution of the clone instruction clones the configuration of each local execution $\pi[i]$. The new executions will be appended to the local executions stack. The state of the new executions is **S**. The privileges of the new local executions are copied from the lists of privileges $PRIV_{T_M}$ and $PRIV_{T_R}$. $PRIV_{T_M}$ ($PRIV_{T_R}$, respectively) is an input (output, respectively) privilege configuration template which varies depending on the enforced property. We give an example of such templates in §5.2, where the enforced property requires cloning.

A *configuration of the* MAP *component* is a set $\{\mathsf{map{:}prg{:}}\pi_M, \mathsf{map{:}mem{:}}m\}$, where π_M is the instruction to be executed, and m is the memory.

The semantics of instructions of assignment, sequence, if, while, and skip of MAP is quite similar to the semantics of corresponding instructions of controlled programs. The output instruction is not used in π_M. The semantics of the two most interesting instructions map and clone is described in Fig. 6. For the map, wake, and clone instructions, if there is no i such that $PRED[i]$ holds, then the execution of these instructions makes all local executions to move from their current configurations to themselves.

$$\text{OUTR}\ \frac{\pi_R = \mathbf{output}\ e\ \mathbf{to}\ c \qquad \mathsf{red.mem} = m \qquad \vec{v} = \vec{\perp}[c \mapsto m(e)]}{\Delta, \mathsf{red{:}prg{:}}\pi_R, \mathsf{out{:}}O \Rightarrow \Delta, \mathsf{red{:}prg{:}}\mathbf{skip}, \mathsf{out{:}}O.\vec{v}}$$

$$\text{WAKR}\ \frac{\begin{array}{c}\pi_R = \mathbf{wake}(PRED[\,]) \qquad S = \{i \in \{0,\ldots,TOP\} : PRED[i]\} \\ \mathsf{LECS} = \bigcup_{i \in S} \{EX[i].\mathsf{int{:}}signal, EX[i].\mathsf{stt{:}}\mathbf{S}\} \qquad \mathsf{LECS'} = \bigcup_{i \in S} \{EX[i].\mathsf{int{:}}\perp, EX[i].\mathsf{stt{:}}\mathbf{E}\}\end{array}}{\Delta, \mathsf{red{:}prg{:}}\pi_R, \mathsf{LECS} \Rightarrow \Delta, \mathsf{red{:}prg{:}}\mathbf{skip}, \mathsf{LECS'}}$$

Fig. 8: Semantics of the REDUCE instructions **output** and **wake**

The bijective function $assignIndex : S \to \{1,\ldots,|S|\}$ assigns and returns a unique index of the element i in the set S (the index starts from 1). The function $fork(\mathsf{LECS}_i, j)$ makes a copy of the local execution $\pi[i]$; the new execution can be referred as $EX[j]$. The function $assign(T_M, T_R, TOP, TOP', PRIV_{T_M}, PRIV_{T_R})$ modifies tables T_M and T_R by adding new columns for the newly cloned processes and the corresponding values for the privileges from $PRIV_{T_M}$ and $PRIV_{T_R}$ for the input and output channels for these processes.

REDUCE. The REDUCE component controls the output actually generated by the enforcement mechanism. A REDUCE program π_R can ask an item from a local execution, send an item to the external output, clean local output queues of local executions and wake local executions up.

Except for the input instruction, that is replaced by the retrieve instruction, in addition to the instructions in Fig. 3 and the wake instruction, the REDUCE program may contain instructions described in Fig. 7. The execution of the retrieve instruction reads the value from the output queue of $\pi[i]$ and stores it into x. The execution of the clean instruction is applied to all local executions $\pi[i]$ such that $PRED[i]$ is true. This instruction removes the first vector $\vec{v}$ of the output queue O of $\pi[i]$, where the value of $\vec{v}[c]$ is different from $\perp$.

A *configuration of the* REDUCE *component* is a set $\{\mathsf{red{:}prg{:}}\pi_R, \mathsf{red{:}mem{:}}m\}$, where π_R is the instruction to be executed, and m is the memory.

$$\begin{array}{lll} \pi_R ::= \ldots & & instructions: \\ \quad |\mathbf{retrieve}\ x\ \mathbf{from}\ (i,c) & & retrieve \\ \quad |\mathbf{clean}(c, PRED[\,]) & & clean \end{array}$$

Fig. 7: The REDUCE instructions

The semantics of the output and wake instructions, as the most interesting ones, is described in Fig. 8.

The enforcement mechanism. A *configuration of an enforcement mechanism* is a set $\{\mathsf{t_m}{:}T_M, \mathsf{t_r}{:}T_R, \mathsf{top}{:}TOP, \mathsf{map}{:}M, \mathsf{red}{:}R, \mathsf{in}{:}I, \mathsf{out}{:}O, \bigcup_i \mathsf{LECS}_i\}$, 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, M and R are configurations of respectively MAP and REDUCE components, I and O are respectively the input and output queues of the enforcement mechanism, and LECS_i is the configuration of the i-th local execution.

The program of MAP (REDUCE respectively) is activated only when the previous execution of MAP (REDUCE) terminated, there is an interrupt signal c from the local execution $\pi[i]$, the state of this local execution is sleeping ($\mathbf{S}$), and the instruction to be executed is an input (output) instruction. The activation of the MAP program or the REDUCE program on a signal on channel c from $\pi[i]$ will remove the signal from $\pi[i]$.

We denote the enforcement mechanism on π by $\mathsf{EM}(\pi)$. 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 enforcement mechanism terminates when all local executions, MAP and REDUCE programs are terminated, and the global input queue is consumed completely.

The enforcement mechanism *terminates* if there exists a configuration $\gamma_f = \{\mathsf{t_m}{:}T_M, \mathsf{t_r}{:}T_R, \mathsf{top}{:}TOP, \mathsf{map}{:}M, \mathsf{red}{:}R, \mathsf{in}{:}\epsilon, \mathsf{out}{:}O, \bigcup_i \mathsf{LECS}_i\}$ such that $\gamma_0 \rightarrow^* \gamma_f$, where $EX[i].\mathsf{prg}{:}\mathbf{skip}$ for all i, $\mathsf{map.prg}{:}\mathbf{skip}$, and $\mathsf{red.prg}{:}\mathbf{skip}$. We denote this whole derivation sequence by $(\mathsf{EM}(\pi), I) \Downarrow O$ using the big step notation.

5 Configurations for the Selected Properties

In [15], Mantel proposes a uniform framework to define possibilistic information flow properties and he proves that existing possibilistic information flow properties can be expressed as a predefined basic security predicate (BSP) or conjunction of these BSPs. A BSP is generally defined in the framework of Mantel based on removal of some high inputs and events.

In the next sections, we will demonstrate configurations of our framework for enforcement of two BSPs, RI and DI, and the SME-style NI. It might not be obvious whether these properties are actually different in our model. We resolve possible doubts of the attentive reader in [18].

Let $COND[\,] \triangleq \lambda\vec{v}.Cond()$ be a predicate and $COND[\vec{v}]$ be the result of the evaluation of $COND[\,]$ on $\vec{v}$. We define the restriction operator on the queue Q with $COND[\,]$, $Q|_{COND[\,]}$, that returns all $\vec{v}$ in Q such that $COND[\vec{v}]$ is true. We will use the notation $Q|_l$, the restriction on security level l, if $Cond(l) \triangleq \lambda\vec{v}.\exists c : \vec{v}[c] \neq \bot \wedge LVL[c] = l$; and the notation $Q|_c$, the restriction on channel c, if $Cond(c) \triangleq \lambda\vec{v}.\vec{v}[c] \neq \bot$.

5.1 Removal of Inputs

The *removal of inputs* (RI) property [15] 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 if all high input items are replaced by the default values or removed, the input queue can be sanitized so that the program will terminate when executing on this input and the generated output will be equivalent at the low level to the original output.

Definition 1. *A program π satisfies the property of* removal of inputs *iff for any potential value chosen as a default value,*

$$\forall I : (\pi, I) \Downarrow O \implies \exists I' : I'|_L = I|_L \wedge I'|_H = (\vec{df})^* \wedge \\ \wedge\, \forall c \in C_{in}, \parallel I'|_c \parallel \leq \parallel I|_c \parallel \wedge (\pi, I') \Downarrow O' \wedge O'|_L = O|_L,$$

where $\vec{df}$ is a vector containing the default value, and $\parallel Q \parallel$ is the length of Q.

The enforcement mechanism of the RI property on the program π only needs two parallel programs: the high ($\pi[0]$) and the low ($\pi[1]$). We specify the full configuration of the local executions in Fig. 9. The high execution can receive (real) input values from L and H channels, while the low execution can receive only (real) input values from L channels. The high execution can write output values only to H channels, the low execution can write values only to L channels. If the interrupt signal is from $\pi[1]$, or the interrupt signal is from $\pi[0]$ and the level of channel c is H, then the input action will be performed. Otherwise, the local execution keeps sleeping.

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

(a) T_M for RI

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

(b) T_R for RI

```
1: if a ∈ T_M[i][c] then
2:     input x from c
3:     map(x, c, canTell(c))
4:     map(val_def, c, ¬canTell(c))
5:     wake(isReady(c))
6: else
7:     skip
```

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

```
1: x := val_def
2: if a ∈ T_R[i][c] then
3:     retrieve x from (i, c)
4: if t ∈ T_R[i][c] then
5:     output x to c
6: clean(c, identical(i))
7: wake(identical(i))
```

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

Fig. 9: Configuration of the enforcement mechanism for RI

The MAP program is described in Fig. 9c. The function $canTell(c)$ indicates whether the local execution $\pi[x]$ can receive real values from MAP: $canTell(c) \triangleq \lambda x.t \in T_M[x][c]$. If a local execution that is sleeping and waiting for an input item from a channel has received the input item required, this local execution is ready to be awoken: $isReady(c) \triangleq \lambda x.EX[x].\mathsf{stt} = \mathbf{S} \wedge EX[x].\mathsf{prg} = \textbf{input } y \textbf{ from } c; \pi \wedge$ $EX[x].\mathsf{in} = I \wedge dequeue(I, c) = (val, I') \wedge val \neq \bot$.

When there is an interrupt signal c from $\pi[i]$ on an output instruction, the REDUCE program provided in Fig. 9d is activated. If the local execution $\pi[i]$ can send items to the c channel, the output action is performed. Otherwise, there is no output action. After that the output queue of $\pi[i]$ is cleaned and only $\pi[i]$ is waken. Since the execution of the wake instruction wakes up only $\pi[i]$, the function $identical()$ is defined as $identical(i) \triangleq \lambda x.x = i$.

Example. We illustrate the enforcement mechanism of RI with the program in Fig. 10. The program has two high input channels `cH1`, `cH2`, and one high output channel `cH3`. It is not secure: with the execution of instructions at lines 3, 4, 7, and 8, the secret values from `cH1` (line 1) and `cH2` (line 8) can influence the value sent to the low output channel `cL3` (line 10). In addition, the sequences of high input items are affected by the low input (line 7 and 8); for example, if the value of `l1` is $\mathbf{T}$, an input item from `cH2` will be consumed. We consider the execution of the program with the input sequence ($\texttt{cH1} = \mathbf{T}$) ($\texttt{cL1} = \mathbf{F}$) ($\texttt{cL2} = m$) ($\texttt{cH2} = M$).

```
1 input h1 from cH1
2 input l1 from cL1
3 if !h1 then
4     l1 := !l1
5 input l2 from cL2
6 h2 := 0
7 if l1 then
8     input h2 from cH2
9 output l2 + h2 to cH3
10 output l2 + h2 to cL3
```

Fig. 10: Running Example Program

The high execution in our framework executes the instructions at lines 1, 2, 3, 5, 6, 7, 9, and 10. The output generated at line 10 is ignored by REDUCE. The low execution executes the instructions from line 1 to 10. MAP reads an

input from cH3 for the input instruction at line 8. The output generated by the output instruction at line 9 is ignored by REDUCE.

We describe the global input, output queues, and local input, output queues in Fig. 11. The values sent to cH3 and cL3 are respectively m and $*{+}m$. Each column in the tables 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.

Input to MAP:

	0	1	2	3
cH1	**T**	$\perp$	$\perp$	$\perp$
cH2	$\perp$	$\perp$	$\perp$	M
cL1	$\perp$	**F**	$\perp$	$\perp$
cL2	$\perp$	$\perp$	m	$\perp$

Output by REDUCE:

	0	1	2	3	4	5
cH3	$\perp$	$\perp$	$\perp$	$\perp$	m	$\perp$
cL3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$*{+}m$

Local Executions:

The high execution $\pi[0]$:

The local input:

cH1	**T**	$\perp$	$\perp$	$\perp$
cH2	$\perp$	$\perp$	$\perp$	M
cL1	$\perp$	**F**	$\perp$	$\perp$
cL2	$\perp$	$\perp$	m	$\perp$

The local output:

cH3	$\perp$	$\perp$	$\perp$	$\perp$	m	$\perp$
cL3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	m

The low execution $\pi[1]$:

The local input:

cH1	**F**	$\perp$	$\perp$	$\perp$
cH2	$\perp$	$\perp$	$\perp$	$*$
cL1	$\perp$	**F**	$\perp$	$\perp$
cL2	$\perp$	$\perp$	m	$\perp$

The local output:

cH3	$\perp$	$\perp$	$\perp$	$\perp$	$*{+}m$	$\perp$
cL3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$*{+}m$

Fig. 11: Example of input and output queues for RI

5.2 Deletion of Inputs

The property of deletion of inputs (DI) [15] requires that if we perturb a possible trace t (where $t = \beta.e.\alpha$ and there is no high input event in α) by deleting the high input event e, then the result can be corrected into a possible trace t' ($t' = \beta'.\alpha'$). The parts β and β' are equivalent on the low input events and the high input events. In other words, the low input events and the high input events in β and β' must be the same. The parts α and α' are also equivalent on the low events and the high input events. Since there is no high input events in α, there is also no high input events in α'.

In our notation, if we have an input queue $I = I_1.\vec{v}.I_2$, where $\vec{v}$ contains a value from a high channel and in I_2 there are either no high input items or only high input items with default values, then this input queue can be changed by replacing $\vec{v}$ by the default vector. The obtained input queue 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 can be consumed completely by a clone of the original program and the output should still be equivalent at the low level to the original output generated with the input I.

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

(a) T_M for DI

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

(b) T_R for DI

```
1:  if LVL[c] == H and i == 0 then
2:      clone(identical(i), PRIV_{T_M}, PRIV_{T_R})
3:  if a ∈ T_M[i][c] then
4:      input x from c
5:      map(x, c, canTell(c))
6:      map(val_def, c, ¬canTell(c))
7:      wake(isReady(c))
8:  else
9:      if t ∉ T_M[i][c] then
10:         map(val_def, c, identical(i))
11:         wake(identical(i))
12:     else
13:         skip
```

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

Fig. 12: Configuration of the enforcement mechanism for DI. REDUCE is in Fig. 9d.

Definition 2. *A program π satisfies the property of* deletion of inputs *DI iff for any potential value chosen as a default value,*

$$\forall I : I = I_1.\vec{v}.I_2 \wedge LVL[c] = H \wedge I_2|_H = (\vec{df})^* \wedge (\pi, I) \Downarrow O \implies$$
$$\exists I' : I' = I'_1.I'_2 \wedge I'|_L = I|_L \wedge I'_2|_H = (\vec{df})^* \wedge (\pi, I') \Downarrow O' \wedge O'|_L = O|_L,$$

where $\vec{v}[c] \neq \bot$ *and* $\vec{df}$ *is a vector containing the default value.*

DI is enforced with the idea that whenever the high execution requests a high input item, this execution will be cloned and the clone cannot receive real values from high channels. The enforcement mechanism for DI (presented in Fig. 12, REDUCE is presented in Fig. 9d) 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 only use the default values. When the high execution is cloned the new execution is inserted into the stack of local executions. The configuration of the clones for input (respectively, output) is presented in Fig. 12a (respectively, 12b) in the column $\pi[i] > 1$; this is the privilege configuration template $PRIV_{T_M}$ ($PRIV_{T_R}$, respectively). In addition, 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.

5.3 Non-Interference

```
1: if a ∈ T_M[i][c] then
2:     input x from c
3:     map(x, c, canTell(c))
4:     map(val_def, c, ¬canTell(c))
5:     wake(isReady(c))
6: else
7:     if t ∉ T_M[i][c] then
8:         map(val_def, c, identical(i))
9:         wake(identical(i))
10:    else
11:        skip
```

Fig. 13: MAP for SME for input from c requested from $\pi[i]$

The enforcement mechanism configured in this section mimics the SME-style enforcement of non-interference [7] from Devriese and Piessens, and therefore inherits also the limitations of SME formal guarantees.

Informally, a program satisfies the termination-insensitive non-interference (TINI) property if given two arbitrary inputs that are equivalent at the low level and the executions of the program on these two inputs are terminated, then the outputs generated are indistinguishable to the users at the low level. In other words, the high input items in these two inputs have no effect on what observable is to users at the low level. Termination-sensitive non-interference (TSNI) additionally requires that the secret input items do not influence the termination of the program [2].

Definition 3. *A program* π *satisfies the property of* termination-insensitive non-interference*, denoted as* $\pi \models$ *TINI, with respect to the given semantics iff*

$$\forall I, I' : I'|_L = I|_L \implies O'|_L = O|_L,$$

where $(\pi, I) \Downarrow O$ *and* $(\pi, I') \Downarrow O'$.

Definition 4. *A program* π *satisfies the property of* termination-sensitive non-interference*, denoted as* $\pi \models$ *TSNI, with respect to the given semantics iff*

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

To implement the SME approach [7], we use the following configuration. The high execution $\pi[0]$ can only ask high input items, while for low input items it needs to wait for the values ask by the low execution $\pi[1]$. The low execution $\pi[1]$ can ask and consume only low items. If the low execution requires a high input item, the default value will be used.

The configuration tables T_M and T_R and the program for REDUCE to enforce the SME-style NI are presented in Fig. 9. However, the program for MAP is different, as shown in Fig. 13. The functions $canTell(i)$, $isReady()$ and $identical(i)$ are defined in Sec. 5.1.

6 Formal Properties

We formalize the soundness and precision properties of an enforcement mechanism and prove the theorems on the security guarantees that the shown enforcement mechanisms ensure with respect to the corresponding properties. Table 1 summarizes the properties and enforcement mechanisms.

Definition 5. *An enforcement mechanism is* sound *with respect to a property P if for all programs π the enforcement mechanism executed on π satisfies P.*

Theorem 1. *Each enforcement mechanism in Tab. 1 is* sound *with respect to the corresponding property, except for TSNI.*

Proof sketch: the low input items consumed and the low output items generated by the enforcement mechanism are always produced by the low execution; the high output items are always generated by the high execution. The differences among different properties come from the constraints on high input items. By using the induction technique on the length of the derivation sequence of the enforcement mechanism, we can prove that the high input items consumed by the enforcement mechanism satisfy the constraints of the enforced property. □

The notion of precision for enforcement of a property is taken from [7, 9]. The intuition is that the enforcement mechanism does not change the visible behavior of a program that is already secure with respect to the chosen property (and in particular each I/O on specific channels). Devriese and Piessens separated by construction the input queues of each channel. Since in our formulation the channels are merged into a global stream, our definition of precision must make explicit that the partial order of input items on a channel is preserved. This observation applies also to the order of output items in output queues. In our framework, the local executions are executed in parallel with no specific order. Therefore, the total order of input items consumed by the enforcement mechanism can be different from the total order of input items in the input queue consumed by the controlled program that already obeys the desired property. However, the partial order of input items on a channel is preserved. This observation applies also to the order of output items in output queues.

Definition 6. *An enforcement mechanism is* precise *with respect to a property, if for any program π that satisfies the property, and for every input I, where $(\pi, I) \Downarrow O$, the actually consumed input I^* and the actual output O^* of the enforcement mechanism regardless of the order of executing local executions will be such that $I^*|_c = I|_c$ and $O^*|_c = O|_c$ for every channel c, and $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$.*

Theorem 2. *Each enforcement mechanism in Tab. 1 is* precise *with respect to the corresponding property, except for TINI.*

Proof sketch: let π be a program satisfying the enforced property and $(\pi, I) \Downarrow O$. Regardless of the order of executing local executions, if the low execution consumes the same low input items as in I and the high execution consumes high input items as in I, then the input consumed by the enforcement mechanism is I^*, where $I|_c = I^*|_c$ for all c (if not, then contradictions will occur). □

We prove Th. 1 and Th. 2 in the full version of this paper [18].

7 Further Properties

Our framework can capture other properties. Other BSPs from [15] can also be enforced. Removal of events (RE) requires that if there is no high input, there is no high output. To enforce RE, when receiving an output request for a high channel from the high execution, REDUCE needs to check whether there are any other high input items different from the default values and affecting the output generated by the high execution. Enforcement of strict removal of inputs (SRI) is similar to the enforcement of RI, but only the low execution can generate output items for both high and low channels. Strict deletion of inputs, deletion of events, and backward strict deletion can be enforced by using the clone instruction and the REDUCE check mentioned above.

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

$\pi[1]$ waits for high input events from $\pi[0]$

Fig. 14: SubDI

By modifying the privileges of local executions we can enforce new properties. A possible configuration is shown in Fig. 14, where the low execution needs to wait for high input items requested by the high execution even though the low execution can only consume default values. This option leads to a novel strict property, which we have called *substitution-deletion of inputs* (SubDI). The configuration of T_R also leads to discovery of new properties. For the lack of space we provide examples of SubDI and the properties by the modification of T_R in the full version of this paper [18].

Our framework can be extended to accommodate information flow policies represented as a complete lattice [6]. For a complete lattice with n elements, the enforcement mechanisms of RI and NI require n local executions, one for each element of the lattice. The enforcement mechanism of DI requires n local executions at initialization; it will spawn a new local execution every time a local execution at the level l (where l is not the level at the bottom of the lattice) requests an input item at the level l.

8 Limitations

Currently, the enforcement mechanism is not independent from the choice of the default values (val_{def}). We prove soundness and precision of enforcement with respect to all possible choices of the default values, and we assume that for each channel it is possible to determine a suitable ("non-leaking") default value.

Our mechanism in §5.3 inherits the limitations of SME [7]. SME can soundly enforce TINI, but not TSNI because the low execution may terminate while the high execution may not terminate, and thus the whole enforcement mechanism does not terminate. SME (and our enforcement mechanism for NI) can precisely enforce TSNI, but not TINI.

In [11] Kashyap, Wiedermann and Hardekopf evaluate the security guarantees of SME for the termination covert channel; they have proposed to mediate the security problems of SME related to this channel with more sophisticated schedulers. In our approach we do not schedule the order of local executions, therefore, we cannot immediately adapt their suggestions. However, our framework can be extended to control the order of executing local executions by specifying a new rule to control the start of local executions, and the predicate $isReady()$ used in the wake instruction.

We see one of the main limitations of our current proposal in the absence of a practical implementation. It is still an open question, whether the memory and performance overhead will be acceptable, especially for complex properties, such as DI. In addition, the fact that MAP and REDUCE are responsible for all respectively input and output operations may have influence on the performance of the enforcement mechanism. Devriese and Piessens in the original SME paper [7], as well as Bielova et al. in [4] and De Groef et al. in [9] report on complications while instrumenting SME for real browsers, which we will have to address. A working implementation is our next target.

9 Related Work

The information flow policies enforcement is a deeply investigated field. We will briefly recall the developed approaches for information flow policies enforcement and discuss the most relevant techniques in more details.

Static analysis techniques for information flow security inspect the program code in order to check whether there is any unwanted information flow. We refer the interested reader to the survey by Sabelfeld and Myers [20] with an excellent overview of static language-based approaches for information flow security.

In contrast to the static verification techniques, dynamic analysis for information flow enforcement tracks propagation of confidential information when a program is executed; an extensive review on the dynamic approach can be found in [14]. The trade-offs between static and dynamic analysis approaches are evaluated by Russo and Sabelfeld in [19].

Our choice of the multi-execution approach, despite its performance overhead which is negligible with multicoress, was dictated by its advantages over the static and dynamic information flow analysis techniques. Static analysis can fall short in scenarios when the program can be composed dynamically (e.g. JavaScript); dynamic runtime monitoring can suffer from impossibility to account for the branch of execution that was not taken, and can leak control flow details [20] through the halting behaviour of the program.

Secure multi-execution [7] has inspired many researchers to push further investigation of this technique. Jaskelioff and Russo in [10] describe their adaptation of SME to Haskell and provide an SME implementation in a handy library. SME is applied to a reactive model of a browser in [4], and is implemented as a fully functional web browser FlowFox that embeds an SME-based runtime enforcement mechanism in [9]. FlowFox is a modification of Firefox, it introduces a noticeable memory and performance overhead, but works with most of the existing web sites. We plan to learn from [9] how to implement a fully working solution and how to evaluate the usability.

Barthe et al. [3] provides sound and precis enforcement of non interference through static program transformation instead of modifying the runtime environment. The transformation technique is based on the main SME idea: a program is transformed into the sequential composition of the same code, first at the low level and then at the high level. The high instance reuses the inputs of the low instance through global input buffers.

Instead of having multiple different executions, in [1] non-interference is achieved by using faceted values (pairs of two values containing low and high information). This allows to simulate multiple executions on different security levels while in fact running a single-process execution. The authors also introduce enforcement of declassification policies with their technique.

Capizzi et al. in [5] propose a shadow execution technique that is similar to SME. Shadow execution consists of replacing the original program with two copies. The private copy (the high execution) receives the confidential data, but is prevented from accessing the network. The public copy (the low execution) receives fake data, but can access the network; the results from the network are supplied also to the private copy. In this way the private copy can avail any network related functionality without leaking the confidential data.

10 Conclusion

We have presented the architecture of an extensible framework for enforcement of information flow properties. To the best of our knowledge, this is the first enforcement mechanism capable to accommodate more than one property. The main idea behind our approach is to run several local instances of a program in parallel, as in secure multi-execution [7], and carefully orchestrate processing of input and output operations of the enforced program through two components (MAP and REDUCE), and two tables (T_M and T_R).

To support our claims on the extensibility of the framework we have provided a set of configurations of the enforcement framework for enforcement of non-interference and several properties from the framework of Mantel [15]. The framework components programs for each of these properties are simple and easy to write. As for software, the correctness proof maybe complicated but writing programs that work should be easy.

Our next steps include the investigation the patterns of T_M, T_R, π_M, π_R and the property to be enforced; a proof-of-concept implementation (we have chosen to implement our framework for a web browser; however, our approach can be suitable to any platform), and extension of the framework with more properties and options for declassification.

Acknowledgements We thank the anonymous reviewers of FCS'2013 for their feedback and suggestions which greatly helped to improve the paper. This work is partly supported by the projects EU-IST-NOE-NESSOS and EU-IST-IP-ANIKETOS.

References

1. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. *SIGPLAN Not.*, 47(1):165–178, Jan. 2012.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Math. Structures in Computer Science*, 21(6):1207–1252, 2011.

3. G. Barthe, et al. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems*, volume 7273 of *LNCS*, pages 186–202, 2012.
4. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. of NSS 2011*, pages 97–104, 2011.
5. R. Capizzi, et al. Preventing information leaks through shadow executions. In *Proc. of ACSAC 2008*, pages 322–331, 2008.
6. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
7. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of IEEE S&P 2010*, pages 109–124, 2010.
8. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of IEEE S&P 1982*, pages 11–20, 1982.
9. W. D. Groef, et al. Flowfox: a web browser with flexible and precise information flow control. In *Proc. of CCS 2012*, pages 748–759, 2012.
10. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Perspectives of Systems Informatics*, volume 7162 of *LNCS*, pages 170–178, 2012.
11. 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*, pages 413–428, 2011.
12. R. Lämmel. Google's MapReduce programming model - revisited. *Sci. Comput. Program.*, 68:208–237, October 2007.
13. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.
14. G. Le Guernic. *Confidentiality enforcement using dynamic information flow analyses*. PhD thesis, Kansas State University, Manhattan, KS, USA, 2007.
15. H. Mantel. Possibilistic definitions of security - an assembly kit. In *Proc. of CSFW 2000*, pages 185–199. IEEE Computer Society, 2000.
16. D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. of IEEE S&P 1987*, pages 161–166, 1987.
17. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of IEEE S&P 1994*, pages 79 –93, May 1994.
18. M. Ngo, F. Massacci, and O. Gadyatskaya. MAP-REDUCE runtime enforcement of information flow policies. Available as the Arxiv report 1305.2136. `http://arxiv.org/abs/1305.2136`. Technical Report DISI-13-019, University of Trento, 2013.
19. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of CSF 2010*, pages 186 –199, july 2010.
20. A. Sabelfeld and A. Myers. Language-based information-flow security. *J. on Selected Areas in Communications*, 21(1):5 – 19, 2003.
21. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. on Comput. Secur.*, 17(5):517–548, Oct. 2009.
22. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.
23. A. Zakinthinos and E. Lee. A general theory of security properties. In *Proc. of IEEE S&P 1997*, pages 94 –102, May 1997.