PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

A Programmable Enforcement Framework for Security Policies

Minh Ngo

Advisor:

Prof. **Fabio Massacci**

Università degli Studi di Trento

Thesis Committee:

Prof. **Heiko Mantel**

Technische Universität Darmstadt

Prof. **Frank Piessens**

Katholieke Universiteit Leuven

Prof. **Luca Viganò**

King's College London

April 2016

Abstract

This thesis proposes the MAP-REDUCE *framework, a programmable framework, that can be used to construct enforcement mechanisms of different security policies. The framework is based on the idea of secure multi-execution in which multiple copies of the controlled program are executed. In order to construct an enforcement mechanism of a policy, users have to write a* MAP *program and a* REDUCE *program to control inputs and outputs of executions of these copies.*

This thesis illustrates the framework by presenting enforcement mechanisms of non-interference (from Devriese and Piessens), non-deducibility (from Sutherland) and deletion of inputs (a policy proposed by Mantel). It demonstrates formally soundness and precision of these enforcement mechanisms.

This thesis also presents the investigation on sufficient condition of policies that can be enforced by the framework. The investigation is on reactive programs that are input total and have to finish processing an input item before handling another one. For reactive programs that always terminate on any input, non-empty testable hypersafety policies can be enforced. For reactive programs that might diverge, non-empty downward closed w.r.t. termination policies can be enforced.

Keywords
[programming language; secure multi-execution; non-interference; information flow policies; hypersafety policies; reactive programs]

Acknowledgements

First and foremost I would like to thank Fabio Massacci for being such a great supervisor and colleague over these past four years.

I am grateful to Frank Piessens and Dimiter Milushev. I am very glad that I had the chance to work with both of you.

I would like to thank Heiko Mantel, Frank Piessens and Luca Viganò for serving in the committee.

I would like to thank the Department of Information Engineering and Computer Science of the University of Trento for offering me the chance of working and studying here.

Thank you to my friends and colleagues in Trento and Leuven. Without you, working in Trento and Leuven would not be what it was.

I am grateful to my parents, brothers, and sisters for their support and encouragements during these past four years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and objectives

Non-interference [22] prevents leakage of confidential data to attackers by requiring that the confidential data do not interfere with events observable to attackers. With or without the confidential data, observations to attackers should be the same. By weakening or strengthening some assumptions of non-interference, researchers proposed other information flow policies, such as non-deducibility [45], declassification polcies [42], etc.

Information flow policies can be enforced by different techniques [41, 30, 20, 38, 47]. Among them, the *secure multi-execution* technique [20, 38, 51, 50, 25, 4, 47] has an advantage that is it does not change behaviour of programs that already comply with the policy. This technique executes multiple instances of the controlled program and carefully controls their input and output behaviors.

There are several enforcement mechanisms based on secure multi-execution technique [20, 38, 51, 50, 25, 4, 47]. However, they work only for a single information flow policy, typically non-interference or non-interference with declassification. Modification of these mechanisms to enforce another information flow policy is not straight-forward.

Research Question 1. *Is there a* general purpose*, i.e. programmable, enforcement framework that is based on the secure multi-execution technique and can be used to construct enforcement mechanisms of different policies?*

Given a general purpose enforcement framework for information flow policies based on the secure multi-execution technique. Enforcement mechanisms from this framework have to satisfy different properties. Two important properties are *soundness* and *precision*. An enforcement mechanism is sound if the application of the enforcement mechanism on any program complies with the policy. An enforcement mechanism is precise if it does not change the behaviour of programs that already obey the policy.

Additional interesting issues are the classes of policies that can be enforced by enforcement mechanisms from such a framework. Constructed enforcement mechanisms from the framework can explore alternative executions and observe inputs and output of these executions. However, they cannot know whether an execution terminates or not.

Purely static enforcement mechanisms (i.e. mechanisms that accept or reject a program after some finite amount of analysis) can enforce decidable properties of programs [23]. A mechanism of execution monitoring can only observe executions of the controlled program and terminate them if it detects a violation. Roughly speaking, execution monitors can enforce computable safety properties [43, 48]. Edit automata are mechanisms that are more powerful than execution monitors and can enforce *infinite renewal properties* [33]. There is no investigation on classes of policies that can be enforced by enforcement mechanisms based on the secure multi-execution technique.

Research Question 2. *Which policies can be enforced soundly and precisely by using a general purpose enforcement framework based on the secure multi-execution technique?*

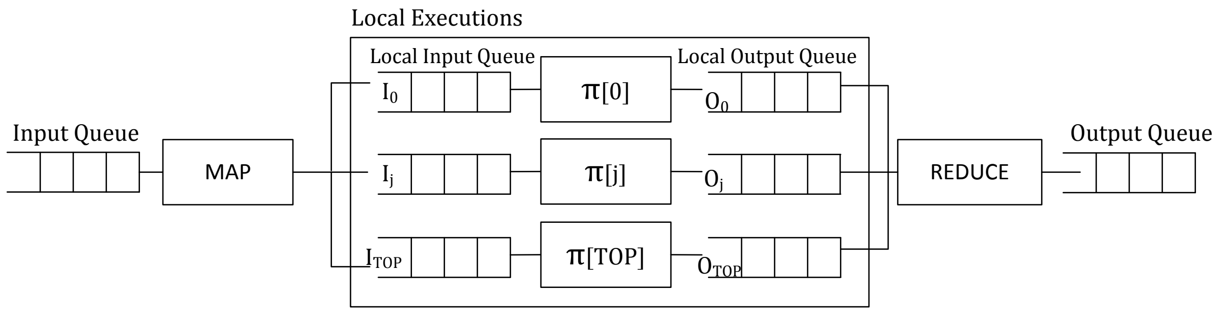


Figure 1.1: Architecture of enforcement mechanisms

1.2 Contribution

This thesis presents the MAP-REDUCE framework, a *programmable* framework, that can be used to construct enforcement mechanisms of different security policies. The enforcement mechanisms rely on secure multi-execution technique. To construct an enforcement mechanism of an information flow policy users have to write a program of MAP to control inputs consumed by the enforcement mechanisms, and a program of REDUCE to control outputs generated by the enforcement mechanism. Programs of MAP and REDUCE are written by using instructions whose semantics is formally specified.

Figure 1.1 depicts the general architecture of the enforcement mechanism of a policy on a controlled program π. It is composed by global input and output queues, MAP and REDUCE components, and an array EX of local executions ($\pi[0], \ldots,$ $\pi[TOP]$, where TOP is the index of the last local execution in the array).

The global input queue contains the input items from the external environment, and the global output queue contains the output items filtered by the enforcement mechanism to the environment.

Local executions $\pi[j]$ are instances of the original program π. They are unaware of each other, and are separated from the environment input and output actions by the enforcement mechanism. 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. 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.

MAP is responsible for performing input operations and sending input items to local executions. It can also make clones of local executions or wake local executions up. REDUCE can collect output items generated by local execution and send output items to global output queue. It can remove local executions from the array of local executions or wake local executions up.

This thesis illustrates the framework by presenting enforcement mechanisms for three information flow policies: non-interference [20], deletion of inputs [34]), and non-deducibility [45]. Soundness and precision of constructed enforcement mechanisms are formally demonstrated.

This thesis presents the investigation on which policies can be enforced soundly and precisely by using this framework. Controlled programs are reactive programs that are input total and have to finish processing an input item before handling another one. The investigation focuses on enforcement mechanisms whose MAP consumes inputs, sends different inputs to different copies of the controlled program, and REDUCE collects outputs from all local executions and decides good outputs to outside. For reactive programs that always terminates on inputs, any non-empty testable hypersafety policy [37] can be enforced. The thesis introduces downward closed w.r.t. termination policies and shows that for reactive programs that might diverge on inputs, any non-empty downward closed w.r.t. termination policies can be enforced.

Table 1.1: Enforcement mechanisms of policies

Policy	Components	
	MAP	REDUCE
Termination (in)sensitive non-interference [20]	Figure 4.2a	Figure 4.2b
Deletion of inputs [34]	Figure 4.5a	Figure 4.5b
Termination (in)sensitive non-deducibility [45]	Figure 4.8a	Figure 4.8b
Non-empty testable hypersafety policies [37]	Figure 5.4	Figure 5.5
Non-empty downward closed w.r.t. termination policies	Figure 6.2	Figure 6.3

A summary of MAP and REDUCE programs for enforcing non-interference, deletion of inputs, non-deducibility, testable hypersafety policies, and downward closed w.r.t. termination policies can be found in Table 1.1.

1.3 Structure of the thesis

This thesis is organized as follows. Chapter 2 presents a review of existing enforcement mechanisms constructed with the secure multi-execution technique, and enforceable policies. Chapter 3 describes semantics of controlled programs and the framework. Chapter 4 illustrates the framework by presenting enforcement mechanisms of non-interference [20], deletion of inputs [34] and non-deducibility [45]. Chapter 5 shows the investigation on which policies can be enforced by the framework on reactive programs that are input total, have to finish processing an input item before handling another one, and process inputs in finite time. Chapter 6 presents downward closed w.r.t. termination policies defined on reactive programs that might diverge on inputs. Chapter 7 concludes the thesis.

Chapter 2

State of the Art

This chapter presents existing enforcement mechanisms of information flow policies. The focus of the presentation is on enforcement mechanisms constructed with secure multi-execution technique. Next, it presents policies that can be statically or dynamically enforced.

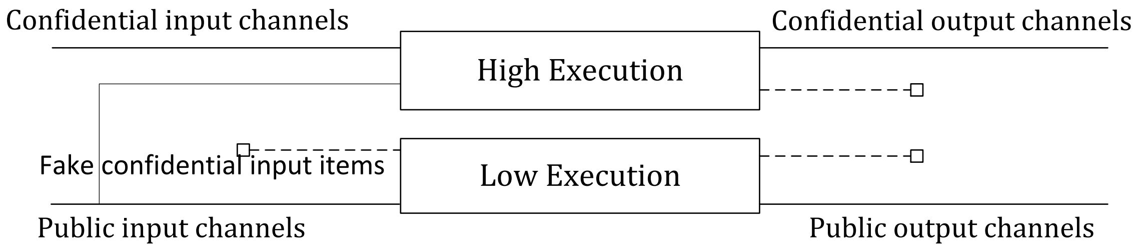
2.1 Enforcement mechanisms of information flow policies

Many of static analyses are based on type systems. An example of a type system for non-interference can be found in [49] where intuitively, variables at confidential level in a well-typed program do not interfere with variables at public level. A discussion of static analysis on enforcing information flow policies can be found in the survey of Sabelfeld and Myers [41] and in Le Guernic's PhD thesis [30]. Static analysis may reject secure programs [41].

For dynamic analysis, an extensive survey up until 2007 is in Le Guernic's PhD thesis [30]. Taint tracking only tracks *explicit flows* and thus is unsound. A generic framework that captures the essense of explicit flows can be found in [44]. Austin and Flanagan [2] proposed an enforcement mechanism in which assignments to public variables in *confidential contexts* are forbidden. In their subsequent work [3], they developed another technique that may be more permissive. When an assignment to a public variable in a confidential context occurs, it is allowed. If later in the execution, there is a branch on this variable, or the value of this variable is to be output, the execution is stopped. Recent works focus on JavaScript. Russo et al. [40] proposed a monitor that track information flow in dynamic tree structures. Russo and Sabelfeld [39] proposed a monitor that prevents insecure information flow via assignments to public variables. They also address internal timing attacks, a type of attacks where attackers do not need access to a clock. Similar to static analysis techniques, dynamic analysis techniques may rejects some secure programs.

Hybrid monitors [31, 29, 8] are based on the combination of dynamic analysis and static analysis and can accept more secure programs. In the monitor proposed by Le Guernic et al. [31], when a branch instruction is executed, variables that can be influenced by confidential data in the unexecuted branch are syntactically analyzed. This information is used to prevent *implicit indirect flows*. In [29], Le Guernic presented a more permissive static analysis and gave constraints that characterize a set of static analysis techniques that can be used. In [8], Besson et al. proposed a generic framework that is parametrised by static analysis techniques. They compared existing hybrid monitors by relative precision and showed that their instantiated monitor is more precise. Hybrid monitors may still reject some secure programs.

The secure multi-execution technique can be used to enforce precisely information flow policies [20, 38, 51]. To the best of our knowledge, the first secure multi-execution based enforcement mechanisms are in [26, 15]. Devriese and Piessens independently formalized the idea in [20] and pro-


Confidential input channels
Confidential output channels
High Execution
Fake confidential input items
Low Execution
Public input channels
Public output channels

Only the high execution (resp. the low execution) can consume input items from confidential input channels (resp. public input channels). The high execution has to reuse input items from public channels consumed by the low execution. When the low execution needs a high input item, it will be fetched a fake input item. Only the high execution (resp. the low execution) can send its outputs to confidential output channels (resp. public output channel).

Figure 2.1: Secure Multi-Execution

posed an enforcement mechanism called secure multi-execution (SME). We next describe SME proposed by Devriese and Piessens since their work has attracted many researchers to the technique.

The basic idea of SME with two security levels is depicted in Figure 2.1. The enforcement mechanism contains two local executions which are copies of the controlled program. The low execution can ask and receive public input items from input channels. Whenever the low execution needs a high input item, it is fetched with a fake value. When the low execution terminates, the high execution starts executing. The high execution can ask and receive confidential input items from input channels, and it has to reuse public input items asked by the low execution. The high (resp. low) execution can send output items to confidential (resp. public) output channels. Output items of the high execution (resp. the low execution) to public (resp. confidential) output channels are ignored.

SME-based enforcement mechanisms. Khatiwala et al. [26] proposed a technique called *data sandboxing*. The controlled program is split into two partitions: *public zone* and *private zone*. The private zone contains instructions used to process confidential data. The public zone contains the remained ones. This technique is applicable only when the source of the controlled program is available. Capizzi et al. [15] proposed the *shadow execution* technique. The privileges of copies of the controlled program on inputs and outputs are as in SME.

In SME, the scheduler for local executions is fixed, that is the low execution has to be executed first. Kashyap et al. [25] explored different scheduler strategies (for example, all local executions are executed in parallel, or local executions are executed in an interleave manner, etc) and their security guarantees on termination and timing covert channels [35].

In SME, for non-interferent programs, the order of the output events are not preserved. This limitation was addressed by Rafnsson and Sabelfeld [38], and Zanarini et al. [51]. In the work of Rafnsson and Sabefeld, instead of being ignored, the public outputs of the high execution are matched with the public outputs of the low execution. If there is a deviation, the controlled program is not non-interferent. To preserve the order of events, Zanarini et al. [51] let the controlled program and the application of SME on the controlled program are executed in parallel and their outputs are checked at each steps. If there is a mismatch, there is a presence of information leakage.

In SME, the set of input and output channels are fixed and each channel is assigned to a fixed security level (for example, high or low). By extending the enforcement mechanism in [51], Zanarini and Jaskelioff proposed the enforcement mechanism that can be used in the situation where the set of channels can be changed and the security level of a channel can be reassigned [50].

Declassification policies [42] are addressed in [38, 47, 4]. In the work of Rafnsson and Sabelfeld [38], for policies in which the occurrence of confidential input events can be declassified [42], when there is a confidential input value sent to the high execution, a fake high input value is sent to the low execution. Thus, the low execution knows the occurrence of confidential events but not their values. For policies which specify what can be declassified, values computed by expressions marked with declassification annotations in the high execution are available for being used by the low execution. In the work of Vanhoef et al., declassify annotations are only directives indicating that a particular value is computed by the *release function* [47]. In their work, the stateless *project function* that projects confidential events to events visible to attackers is idempotent. Bolosteanu and Garg [14] proposed an enforcement mechanism called *asymmetric SME* in which the low execution is a variant of the original program. Asymmetric SME can enforce declassification policies where the projection functions are not idempotent, and can enforce policies whose states depend on program outputs. Austin and Flanagan [4] proposed enforcement mechanisms for *robust declassification* [52], which requires that active attackers (who can introduce new code) are no more powerful than passive attackers (who can only observe).

Austin and Flanagan [4] proposed the *multiple facet technique* that can simulates SME by using *faceted values.* A faceted value is a triple consisting of a principal, a value that appears to observers who can view private data of the principal, and a value that appears to observers who cannot view private data of the principal. SME with two security levels (confidential and public) can be simulated by the multiple facet technique with a single principal. The author reported that when the number of principals increases, the multiple facet technique is the more efficient approach.

Barthe et al. [5] proposed a way to achieve the security guarantees of SME by program transformation. Given a controlled program, a new program is created by sequentially composing copies of the original programs. They also proposed a variant of the transformation for concurrent programming languages. One advantage of program transformation is that it does not require modifications to runtime environment.

Implementations of SME are reported in [26, 15, 20, 24, 9, 18, 4, 19, 47]. Khatiwala et al. [26] reported the implementation of a prototype. Capizzi et al. [15] implemented their proposed techniques on Firefox, Adobe Reader, etc. Devriese and Piessens [20] implemented a prototype using a Javascript engine. Jaskelioff and Russo [24] proposed a monadic library to implement SME in Haskell. Austin and Flanagan [3] evaluated their idea of faceted values by an implementation on Firefox. They reported that when the number of principals increases, their approach becomes more efficient. Bielova et al. [9] reported an implementation of SME with Featherfox [13], a model of browsers implemented in Ocaml. In implementations of both Bielova et al. [9] and Capizzi et al. [15], two local copies of the browser are executed. De Groef et al. [18, 19] implemented SME with FireFox and presented the first fully functional web browser. Instead of having two instances of the browser executed as in [9, 15], two instances of scripts are executed. They reported that the performance and memory cost is substantial but not prohibitive. The implementation reported in [47] is based on the implementation on [18, 19].

A summary of the SME-based enforcement mechanisms is presented in Table 2.1. The last five columns of the table correspond to five features of enforcement mechanisms: (1) Scheduler: enforcement mechanisms investigate the influence of the order of executing local executions on security guarantees; (2) Event Order: enforcement mechanisms preserves order of output events; (3) Dynamic Channel: enforcement mechanisms can cope with the situation where the set of channels is not fixed and security levels of channels can be changed dynamically; (4) Declassification: enforcement mechanisms can enforce declassification policies; and (5) Implementation: enforcement mechanisms are implemented. When we mark an enforcement mechanism with the X mark, we mean that this enforcement mechanism has a particular feature. The work in [18] is extended in [19] and hence it is not presented in the table.

2.2 Enforceable security policies

Hamlen et al. [23] investigated the class of statically enforceable policies. Generally, a policy is statically enforceable if there is a machine that takes an arbitrary program as an input and returns true in finite time if the program satisfies the policy, otherwise, it returns false in finite time. Hamlen et al. proved that the class of statically enforceable security policies is the class of decidable properties of programs. They also proposed a class of policies that can be enforced by *program rewriters*. A program rewriter modifies, in finite time, untrusted programs before their executions. However, they did not give a precise characterization of this class.

Schneider [43] demonstrated that for a policy to be enforced by *execution monitors*, it has to be a safety property [27]. Schneider also introduced monitors called *security automata* that can be used as recognizers for safety properties. These automata can observe executions of programs, and terminating the executions if violations are detected.

The condition for a policy to be enforced was further refined by Viswanathan [48]. Viswanathan observed that a finite execution must be checked whether it is good or bad in finite time. If this condition is violated, then an enforcement mechanism cannot know whether the finite execution it has observed so far is good or not and thus cannot do an appropriate action.

Hamlen et al. pointed out that not all policies that satisfy conditions proposed by Schneider and Viswanathan can be enforced [23]. For example, it is impossible for enforcement mechanisms to enforce a policy if it forbids all interventions available to enforcement mechanisms. Basin et al. revised enforceable safety policies [7, 6]. They differentiated actions that

Table 2.1: SME-based enforcement mechanisms

Paper	Year	Scheduler	Event Order	Dynamic Channel	Declassification	Implementation
Khatiwala et al. [26]	2006					×
Capizzi et al. [15]	2008					×
Devriese and Piessens [20]	2010					×
Jaskelioff and Russo [24]	2011					×
Kashyap et al. [25]	2011	×				
Bielova et al. [9]	2011					×
Austin and Flanagan [4]	2012				×	×
Rafnsson and Sabelfeld [38]	2013		×		×	
Zanarini et al. [51]	2013		×			
De Groef et al. [19]	2014					×
Mathy et al. [47]	2014				×	×
Zanarini and Jaskelioff [50]	2014			×		
Bolosteanu and Garg [14]	2016				×	

are only observable by enforcement mechanisms (that is the enforcement mechanism can see them but not prevent) and actions that are controllable by enforcement mechanisms. They gave necessary and sufficient conditions for a security policy to be enforceable based on their generalized notion of safety properties.

Ligatti et al. investigated which policies can be enforced by edit automata [32, 33]. In addition to terminating executions as security automata, edit automata can insert events or remove events. Ligatti et al. proved that edit automata can enforce a class of policies called *infinite renewal properties* which include some liveness properties [1]. They argued that the set of computable safety properties is included in the set of infinite renewal properties.

Given an edit automaton, Bielova and Massacci investigated the question whether the automaton really enforced the security policy that you wanted [10, 11]. They introduced a fine grained classification of edit automata and related security properties. They also investigated the influence of enforcement mechanisms on bad executions (or how far bad executions are changed by enforcement mechanisms) [12]. They proposed *iterative properties* and enforcement mechanisms that can handle bad executions in a more predictable way.

Fong [21] investigated policies that can be enforced by execution monitors that are limited by the information that they can track. He introduced *shallow history automata* that track only a shallow access history and use this information to make decision on granting access. Despite the limitation, many policies are still enforceable. Talhi et al. [46] extended works of Schneider [43], Ligatti [32, 33], and Fong [21], and introduced a class of automata called *bounded history automata*. They gave a classification of policies that are enforceable under memory limitation constraints.

2.3 Summary

This chapter presented several enforcement mechanisms of information flow policies which are constructed based on the secure multi-execution technique. One problem with those enforcement mechanisms is that they enforce only single information flow policy, usually non-interference or non-interference with declassification. This chapter also presented the investigation on enforceable policies. The existing investigation does not cover runtime enforcement mechanisms that can explore different executions.

Chapter 3

MAP-REDUCE Framework

This chapter presents the small step operational semantics of controlled programs, local executions, MAP *programs, and* REDUCE *programs.*

3.1 Overview

Instructions used to compose controlled programs, MAP, and REDUCE programs are presented in respectively Figure 3.1a, Figure 3.1b, and Figure 3.1c. Basic instructions which are assignment, condition, loop, skip, input and output instructions might be used to compose controlled programs, MAP programs, and REDUCE programs.

MAP is responsible for sending inputs to local executions, creating local executions, or changing states of local executions. Thus, in addition to basic instructions, instructions in programs of MAP might be map, wake, and clone instructions. MAP can send input values to local executions by map instructions, wake local executions up by wake instructions, and make clones of local executions by clone instructions.

REDUCE needs to collect outputs from local executions, remove local executions, or modify states of local executions. Therefore, for REDUCE, additional instructions are retrieve, wake, and kill instructions. Retrieve

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

(a) Controlled program instructions

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

(b) MAP instructions

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

(c) REDUCE instructions

π, e, x, c, π_M, π_R, and $PRED$ are meta-variables for respectively instructions, expressions, variables, (input/output) channels, instructions of MAP, instructions of REDUCE, and predicates. A (controlled, MAP, or REDUCE) program is a sequence of instructions. Executions of instructions with $PRED$ as a parameter influence local executions that satisfies $PRED$.

Figure 3.1: Language instructions

instructions allow REDUCE to collect output items generated by local executions. Kill instructions allow REDUCE to remove local executions from the array of local executions. REDUCE can clean local output queues of local executions by clean instructions.

3.2 Semantics of controlled programs

Our model programming language is derived from [20]. We consider only deterministic programs. A program π is an instruction composed from the basic instructions described in Figure 3.1a. Since a program is a sequence of instructions (i.e. a complex instruction itself), we will use program and instruction interchangeably. An example of controlled programs is presented in Example 3.2.1.

Example 3.2.1. *Figure 3.2a present a JavaScript script that facilitates the job application process. The execution of this program gets the selected position of an applicant from a low channel (Figure 3.2a, line 1) and his desired salary from a high channel (Figure 3.2a, line 2). Then, if the selected position is CEO and the desired salary is not too high, the execution will get the bonus from a confidential channel which is* `https://aCompany/getBonus`*. After that, the program displays the desired salary and the bonus to user (Figure 3.2a-line 15), and send these data to* `http://attacker` *(Figure 3.2a-lines 17-19), a low channel that attackers can directly observe.*

The program is rewritten in our model language and the rewritten version is described in Figure 3.2b. The public input channel for getting the selected position is `cL1`*, the confidential input channels for getting the desired salary and the bonus are respectively* `cH1` *and* `cH2`*. Confidential channel* `cH3` *is corresponding to the one used to show desired salary and bonus to user. Public channel* `cL2` *is corresponding to* `http://attacker`*.*

Let **I** (resp. **O**) be an enumerable set of input values (resp. output

```
1 var selectedPosition = document.getElementbyId('selectedPosition').value;
2 var desiredSalary = document.getElementbyId('desiredSalary').value;
3
4 var bonus = 0;
5 var xmlHttpsBonus = new XMLHttpRequest();
6 xmlHttpsBonus.open("POST","https://aCompany/getBonus",false);
7
8 var isSalaryHighNotTooHigh = checkSalary(desiredSalary)
9 if (selectedPosition == 'CEO') and isSalaryHighNotTooHigh {
10     xmlHttpsBonus.send();
11     var xmlDocs=xmlHttpsBonus.responseXML;
12     bonus = xmlDocs.getElementsByTagName("bonus")[0].
13                  childNodes[0].nodeValue;  }
14
15 document.getElementbyId('info').value = desiredSalary + bonus;
16
17 var xmlhttpAttacker = new XMLHttpRequest();
18 xmlhttpAttacker.open("POST","http://attacker/");
19 xmlhttpAttacker.send(desiredSalary + bonus);
```

(a) In JavaScript

```
input l1 from cL1                //corresponding to line 1
input h1 from cH1                //corresponding to line 2
h2 := 0                          //corresponding to line 4
b := check(h1)                   //corresponding to line 8
if l1 and b then                 //corresponding to line 9
    input h2 from cH2            //corresponding to lines 10-13
output h1 + h2 to cH3            //corresponding to line 15
output h1 + h2 to cL2            //corresponding to lines 17-19
```

(b) In our language

Figure 3.2: Running example program - Job application

values). We model an input (output) item as a vector and define input (output) of programs 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).

Definition 3.2.1. *An* input vector $\vec{v}$ *is a mapping from input channels to their values,* $\vec{v} : C_{in} \rightarrow \boldsymbol{I} \cup \{\bot\}$*, where the value* $\bot$ *is the special undefined value. An* output vector $\vec{v}$ *is a mapping from output channels to their values,* $\vec{v} : C_{out} \rightarrow \boldsymbol{O} \cup \{\bot\}$*.*

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

Let (finite) queue Q be a sequence of elements $q_1.q_2 \dots q_n$. The addition of new element q to Q is denoted by $Q.q$ and the result of the addition is queue $q_1.q_2 \dots q_n.q$. After removing the first element from Q, we get $q_2 \dots q_n$. 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 in the form $\mathsf{label}{:}object$. The label is attached to the *object* in order to differentiate this object from others, so each label occurs only once. For example, $\mathsf{map{:}prg}\pi_M$ is the instruction to be executed of MAP. A summary of labels and their semantics used in this report is in Table 3.1.

Definition 3.2.2. *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 that is a function mapping variables to values,* I *is the input (a queue of input vectors) to be consumed, and* O *is the generated output (a queue of output vectors).*

The operational semantics of controlled programs is described in Figure 3.3. 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

$$\text{ASSG}\ \frac{\pi = x := e \qquad m(e) = val}{\Delta, \mathsf{prg}{:}\pi, \mathsf{mem}{:}m \rightarrowtriangle \Delta, \mathsf{prg}{:}\mathbf{skip}, \mathsf{mem}{:}m[x \mapsto val]}$$

$$\text{COMP}\ \frac{\mathsf{prg}{:}\pi_1, \mathsf{mem}{:}m, \mathsf{in}{:}I, \mathsf{out}{:}O \rightarrowtriangle \mathsf{prg}{:}\pi_1', \mathsf{mem}{:}m', \mathsf{in}{:}I', \mathsf{out}{:}O'}{\mathsf{prg}{:}\pi_1;\pi_2, \mathsf{mem}{:}m, \mathsf{in}{:}I, \mathsf{out}{:}O \rightarrowtriangle \mathsf{prg}{:}\pi_1';\pi_2, \mathsf{mem}{:}m', \mathsf{in}{:}I', \mathsf{out}{:}O'}$$

$$\text{IF-T}\ \frac{\pi = \mathbf{if}\ e\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2 \qquad m(e) = \mathbf{T}}{\Delta, \mathsf{prg}{:}\pi \rightarrowtriangle \Delta, \mathsf{prg}{:}\pi_1}$$

$$\text{IF-F}\ \frac{\pi = \mathbf{if}\ e\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2 \qquad m(e) = \mathbf{F}}{\Delta, \mathsf{prg}{:}\pi \rightarrowtriangle \Delta, \mathsf{prg}{:}\pi_2}$$

$$\text{WHIL-T}\ \frac{\pi = \mathbf{while}\ e\ \mathbf{do}\ \pi_{loop} \qquad m(e) = \mathbf{T}}{\Delta, \mathsf{prg}{:}\pi \rightarrowtriangle \Delta, \mathsf{prg}{:}\pi_{loop};\pi}$$

$$\text{WHIL-F}\ \frac{\pi = \mathbf{while}\ e\ \mathbf{do}\ \pi_{loop} \qquad m(e) = \mathbf{F}}{\Delta, \mathsf{prg}{:}\pi \rightarrowtriangle \Delta, \mathsf{prg}{:}\mathbf{skip}}$$

$$\text{SKIP}\ \frac{}{\Delta, \mathsf{prg}{:}\mathbf{skip};\pi \rightarrowtriangle \Delta, \mathsf{prg}{:}\pi}$$

$$\text{INP}\ \frac{\pi = \mathbf{input}\ x\ \mathbf{from}\ c \qquad I = \vec{v}.I' \qquad \vec{v}[c] \neq \bot}{\Delta, \mathsf{prg}{:}\pi, \mathsf{mem}{:}m, \mathsf{in}{:}I \rightarrowtriangle \Delta, \mathsf{prg}{:}\mathbf{skip}, \mathsf{mem}{:}m[x \mapsto \vec{v}[c]], \mathsf{in}{:}I'}$$

$$\text{OUTP}\ \frac{\pi = \mathbf{output}\ e\ \mathbf{to}\ c \qquad \vec{v} = \vec{\bot}[c \mapsto m(e)]}{\Delta, \mathsf{prg}{:}\pi, \mathsf{out}{:}O \rightarrowtriangle \Delta, \mathsf{prg}{:}\mathbf{skip}, \mathsf{out}{:}O.\vec{v}}$$

Figure 3.3: Semantics of instructions of controlled programs

Table 3.1: Labels and their semantics

Label	**Semantics**
prg	Controlled program or program executed by a component (a local execution, MAP, or REDUCE)
mem	Memory of the controlled program or of a component (a local execution, MAP, or REDUCE)
in	Input of the controlled program, the enforcement mechanism, or a local execution
out	Output of the controlled program, the enforcement mechanism, or a local execution
top	Index of the last execution in the array EX
stt	State of a local execution (Sleeping or Executing)
int	Interrupt signal sent by a local execution
map	Configuration of the MAP component
red	Configuration of the REDUCE component

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 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 $\vec{v}[c] = m(e)$.

The initial configuration of controlled program π with input I is $\{\mathsf{prg}{:}\pi, \mathsf{mem}{:}m_0, \mathsf{in}{:}I, \mathsf{out}{:}\epsilon\}$, where m_0 is the function mapping variable to initial values. An *execution* of program π on input I is a finite sequence of configuration transitions $\gamma_0 \rightharpoonup \gamma_1 \rightharpoonup \ldots \rightharpoonup \gamma_k$, where γ_0 is the initial configuration of π with input I, and γ_k is the configuration after k transitions. The transition sequence can be also written as $\gamma_0 \rightharpoonup^k \gamma_k$, or $\gamma_0 \rightharpoonup^* \gamma_k$ if the exact number of transitions does not matter.

Definition 3.2.3. *An execution of program π on input I* terminates *if there exists a configuration $\gamma_f = \{\mathsf{prg}{:}skip, \mathsf{mem}{:}m, \mathsf{in}{:}\epsilon, \mathsf{out}{:}O\}$ such that*

Input:

Channel \ Time	0	1
`cH1`	$\perp$	M_1
`cH2`	$\perp$	$\perp$
`cL1`	**T**	$\perp$

Output:

Channel \ Time	0	1	2	3
`cH3`	$\perp$	$\perp$	$M_1 + 0$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$M_1 + 0$

Each column in the tables corresponds to an input/output operation. The execution consumes **T** and M_1 from channel `cL1` and channel `cH1` at respectively time 0 and time 1. At time 2 and 3, it generates output values M_1 on channel `cH3` and M_1 on channel `cL2`.

Figure 3.4: Execution of the example program

$\gamma_0 \rightharpoonup\!\!\triangleright^* \gamma_f$, *where* γ_0 *is the initial configuration of* π *with input* I. *We denote this whole derivation sequence by* $(\pi, I) \Downarrow O$ *using the big step notation.*

Example 3.2.2. *Let* $\boldsymbol{T}$ *(resp.* $\boldsymbol{F}$*) denote the boolean value* true *(resp.* false*). We consider the execution of the program presented in Figure 3.2 with input* $($`cL1` $= \boldsymbol{T})($`cH1` $= M_1)$*, which means that the input contains two vectors, the first one contains* $\boldsymbol{T}$ *from* `cL1`*, and the last one contains* M_1 *from* `cH1`*. We assume that check*$(M_1) = \boldsymbol{F}$*.*

The execution of the program with the input terminates and generates output $($`cH3` $= M_1 + 0)($`cL2` $= M_1 + 0)$*. The value sent to* `cH3` *and* `cL2` *is* $M_1 + 0$*.*

We describe input and output queues in Figure 3.4, where 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.*

The execution consumes $\boldsymbol{T}$ *and* M_1 *from channel* `cL1` *and channel* `cH1` *at respectively time* 0 *and time* 1*. At time* 2 *and* 3*, it generates respectively output value* M_1 *on channel* `cH3` *and* M_1 *on channel* `cL2`*.*

3.3 Semantics of enforcement mechanisms

Definition 3.3.1. *A* configuration of an enforcement mechanism *of a policy is a set* $\{\mathsf{top}:TOP, \mathsf{map.prg}:\pi_M, \mathsf{map.mem}:m_M, \mathsf{red.prg}:\pi_R, \mathsf{red.mem}: m_R, \mathsf{in}:I, \mathsf{out}:O, \bigcup_j \mathsf{LECS}_j\}$*, where TOP is the index of the last execution in the array of local executions EX, π_M (resp. π_R) is the instruction to be executed of* MAP *(resp.* REDUCE*), m_M (resp. m_R) is the memory of* MAP *(resp.* REDUCE*), I is the input that can be consumed, O is the generated output, and LECS_j is the configuration of the j-th local execution.*

We denote the enforcement mechanism of policy $\mathcal{P}$ on program π by $\mathsf{EM}_{\mathcal{P}}(\pi)$. In the *initial configuration of the enforcement mechanism with input I*, the input that can be consumed is I, the output generated is empty, variables in programs of MAP and REDUCE are mapped to initial values, skip is the only instruction to be executed of MAP and REDUCE. In addition, all local inputs and local outputs of local executions are empty, there is no interrupt signal generated by local executions, all variables of all local executions are mapped to initial values, instructions to be executed of all local executions and the controlled program are the same.

Definition 3.3.2. *The execution of the enforcement mechanism on input I* terminates *if there exists a configuration γ_f such that $\gamma_0 \Rightarrow^* \gamma_f$, where γ_0 is the initial configuration of the enforcement mechanism with input I; and in γ_f the input that can be consumed is empty, skip is the only instruction to be executed of all local executions,* MAP *and* REDUCE*. We denote this whole derivation sequence by $(\mathsf{EM}_{\mathcal{P}}(\pi), I) \Downarrow O$ using the big step notation.*

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 [28] so each transition rule either by a local execution, by MAP, or by REDUCE is a step of the enforcement mechanism as a whole.

3.3.1 Local executions

Each local execution is associated with a unique identifier j, that is its index on the array EX. A local execution can be in one of the two states: **E** (Executing) or **S** (Sleeping). Initially, states of all local executions depend on policies to be enforced (e.g. in the enforcement mechanisms of sample policies in Chapter 4, initial states of all local executions are executing states). 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, or to signal that it has generated an output item. A local execution moves from **S** to **E** when it is awaken by the MAP component (e.g. the input item it required is ready) or by the REDUCE component (e.g. its output item is consumed).

Definition 3.3.3. *An execution configuration of local execution $EX[j]$ is a set $\mathsf{LECS}_j \triangleq \{EX[j].\mathsf{stt}: st, EX[j].\mathsf{int}: sig, EX[j].\mathsf{prg}: \pi, EX[j].\mathsf{mem}: m, EX[j].\mathsf{in}{:}I, EX[j].\mathsf{out}{:}O\}$, where st is the state of the local execution, sig 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 local input and local output respectively.*

We define dequeue operator $dequeue(Q, c)$ on queue Q and channel c that returns (val, Q'), where the value of val is $\vec{v}[c]$, and $\vec{v}$ is the first vector in Q such that $\vec{v}[c] \neq \bot$, and Q' is obtained by removing $\vec{v}$ from Q; otherwise (there is no vector $\vec{v}$ in Q such that $\vec{v}[c] \neq \bot$), $val = \bot$ and $Q' = Q$.

The semantics of local executions for assignment, composition, if, while, skip instructions is essentially identical to the one of the controlled pro-

$$\text{LINP1}\ \dfrac{\begin{array}{lrr} EX[j].\mathsf{prg}{:}\pi = \textbf{input}\ x\ \textbf{from}\ c & & \\ EX[j].\mathsf{stt} = \mathbf{E} & & \\ EX[j].\mathsf{in} = I & \text{dequeue}(I, c) = (val, I') & val \neq \bot \end{array}}{\begin{array}{c} \Delta, EX[j].\mathsf{prg}{:}\pi, EX[j].\mathsf{mem}{:}m, EX[j].\mathsf{in}{:}I \\ \Rightarrow \Delta, EX[j].\mathsf{prg}{:}\textbf{skip}, EX[j].\mathsf{mem}{:}m[x \mapsto val], EX[j].\mathsf{in}{:}I' \end{array}}$$

$$\text{LINP2}\ \dfrac{\begin{array}{lr} EX[j].\mathsf{prg}{:}\pi = \textbf{input}\ x\ \textbf{from}\ c & \\ EX[j].\mathsf{stt} = \mathbf{E} & \\ EX[j].\mathsf{in} = I & \text{dequeue}(I, c) = (\bot, I') \end{array}}{\Delta, EX[j].\mathsf{stt}{:}\mathbf{E}, EX[j].\mathsf{int}{:}\bot \Rightarrow \Delta, EX[j].\mathsf{stt}{:}\mathbf{S}, EX[j].\mathsf{int}{:}\odot(c)}$$

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

Figure 3.5: Semantics of input and output instructions of $\pi[j]$

grams. The only difference is the explicit condition that the local state must be **E**. We do not present these rules in the paper. We provide the rules for input and output instructions in Figure 3.5. 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 $\odot(c)$ and moves to the sleep state (rule LINP2). The output interrupt signal $\odot(c)$ is generated when the output instruction is executed (rule LOUTP).

Example 3.3.1. *We consider the execution of a local execution of the program presented in Figure 3.2. Its local input is* $(\texttt{cH1} = M_1)(\texttt{cL1} = \boldsymbol{T})$*, which means that the input contains two vectors, the first one contains* M_1 *from* `cH1`*, and the last one contains* $\boldsymbol{T}$ *from* `cL1`*. We assume that* $check(M_1) = \boldsymbol{F}$*.*

The local execution terminates and generates output $(\texttt{cH3} = M_1{+}0)(\texttt{cL2} = M_1 + 0)$*. The value sent to* `cH3` *and* `cL2` *is* $M_1 + 0$*. We describe input and output corresponding to the execution in Figure 3.6, where each column in the tables corresponds to an input/output operation.*

The execution consumes $\boldsymbol{T}$ *and* M_1 *from channel* `cL1` *and channel* `cH1` *at respectively time* 0 *and time* 1*. At time* 2 *and* 3*, it generates respectively output value* M_1 *on channel* `cH3` *and* M_1 *on channel* `cL2`*. Notice that the first input item in the input does not contain an input value from channel* `cL1` *and the local execution has to find the first input item that contains such an input value.*

3.3.2 MAP

MAP controls the input actually consumed by the enforcement mechanism. MAP can perform input operations, broadcast input items to local input queues of local executions, clone local executions and wake local executions

Input:

Channel \ Time	0	1
cH1	M_1	$\perp$
cH2	$\perp$	$\perp$
cL1	$\perp$	$\mathbf{T}$

Output:

Channel \ Time	0	1	2	3
cH3	$\perp$	$\perp$	$M_1 + 0$	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$M_1 + 0$

Each column in the tables corresponds to an input/output operation. The execution consumes $\mathbf{T}$ and M_1 from channel `cL1` and channel `cH1` at respectively time 0 and time 1. At time 2 and 3, it generates output values M_1 on channel `cH3` and M_1 on channel `cL2`.

Figure 3.6: Execution of a local execution

up.

In addition to the instructions in Figure 3.1a (except the output instruction that is replaced by the map instruction), the program π_M is also composed by instructions $\mathbf{map}(e, c, PRED[\ ])$, $\mathbf{clone}(PRED[\ ])$, and $\mathbf{wake}(PRED[\ ])$ described in Figure 3.1b, where e is a meta-variable for expressions, and $PRED[\ ] \triangleq \lambda x.Pred(x)$ is a meta-variable for predicates. The evaluation of the predicate $PRED[\ ]$ on the configuration of the local execution $\pi[i]$ is denoted as $PRED[i]$.

The execution of map, wake, or clone instruction is applied simultaneously to all local executions $\pi[j]$ such that $PRED[j]$ is true as follows. For map instruction, the value of the expression e is sent to the input queues of all local executions. The value sent is considered as a value from the channel c. For wake instruction, all local executions $\pi[j]$ are awaken and the interrupt signals generated by those local executions (if there were some) are removed. The execution of the clone instruction clones the configuration of each local execution $\pi[j]$. Configurations of new executions are appended to the array of local executions, and the index of the last local execution (TOP) is updated.

The semantics of instructions of assignment, sequence, if, while, and skip of MAP is similar to the semantics presented in Figure 3.3. The output

$$\text{INPM}\ \frac{\begin{array}{ll}\textsf{map.prg}{:}\pi_M = \textbf{input } x \textbf{ from } c & \\ I = \vec{v}.I' & \vec{v}[c] \neq \bot\end{array}}{\Delta, \textsf{map.prg}{:}\pi_M, \textsf{map.mem}{:}m_M, \textsf{in}{:}I \Rightarrow \Delta, \textsf{map.prg}{:}\textbf{skip}, \textsf{map.mem}{:}m_M[x \mapsto \vec{v}[c]], \textsf{in}{:}I'}$$

$$\text{MAP}\ \frac{\begin{array}{ll}\textsf{map.prg}{:}\pi_M = \textbf{map}(e, c, PRED[\,]) & \\ S = \{j \in \{0,\ldots,TOP\} : PRED[j]\} & \vec{v} = \vec{\bot}[c \mapsto m_M(e)] \\ \textsf{LECS} = \bigcup_{j\in S}\{EX[j].\textsf{in}{:}I\} & \textsf{LECS}' = \bigcup_{j\in S}\{EX[j].\textsf{in}{:}I.\vec{v}\}\end{array}}{\Delta, \textsf{map.prg}{:}\pi_M, \textsf{LECS} \Rightarrow \Delta, \textsf{map.prg}{:}\textbf{skip}, \textsf{LECS}'}$$

$$\text{WAKM}\ \frac{\begin{array}{ll}\textsf{map.prg}{:}\pi_M = \textbf{wake}(PRED[\,]) & \\ S = \{j \in \{0,\ldots,TOP\} : PRED[j]\} & \\ \textsf{LECS} = \bigcup_{j\in S}\{EX[j].\textsf{int}{:}sig, EX[j].\textsf{stt}{:}\mathbf{S}\} & \textsf{LECS}' = \bigcup_{j\in S}\{EX[j].\textsf{int}{:}\bot, EX[j].\textsf{stt}{:}\mathbf{E}\}\end{array}}{\Delta, \textsf{map.prg}{:}\pi_M, \textsf{LECS} \Rightarrow \Delta, \textsf{map.prg}{:}\textbf{skip}, \textsf{LECS}'}$$

$$\text{CLON}\ \frac{\begin{array}{l}\textsf{map.prg}{:}\pi_M = \textbf{clone}(PRED[\,]) \\ S = \{j \in \{0,\ldots,TOP\} : PRED[j]\} \\ \textsf{LECS} = \bigcup_{0\le j\le TOP} \textsf{LECS}_j \\ \textsf{LECS}' = \textsf{LECS} \cup \bigcup_{j\in S} \text{fork}(\textsf{LECS}_j, TOP + \text{assignInd}(j))\end{array}}{\Delta, \textsf{top}{:}TOP, \textsf{map.prg}{:}\pi_M, \textsf{LECS} \Rightarrow \Delta, \textsf{top}{:}TOP + |S|, \textsf{map.prg}{:}\textbf{skip}, \textsf{LECS}'}$$

Figure 3.7: Semantics of instructions of MAP

instruction is not used in π_M. The semantics of input, map, wake, and clone instructions is described in Figure 3.7. The execution of map instruction or wake instruction changes only the program of MAP (referred in the configuration of the enforcement mechanism by map.prg), and configuration of local executions that satisfy $PRED[\,]$. In addition to these changes, the execution of the clone instruction also changes the index TOP. For map, wake, and clone instructions, if there is no j such that $PRED[j]$ holds, then the execution of these instructions makes all local executions move from their current configurations to themselves.

The cardinality of a set S is $|S|$. Assume that S is a set of integer numbers; bijective and increasing function $assignInd : S \to \{1, \ldots, |S|\}$ assigns and returns a unique index of the element j in the set S. Function $fork(\mathsf{LECS}_j, k)$ makes a copy of the local execution $\pi[j]$; the new execution can be referred as $EX[k]$ (notice that k is smaller than or equal to the updated TOP).

When MAP is activated, its configuration is $\{\mathsf{map.prg} : \pi_M, \mathsf{map.mem} : m_M\}$, where π_M is the program of MAP tailored to the policy to be enforced (for example, for non-interference the program of MAP is described in Figure 4.2a), and m_M is the memory of MAP from the last activation. The execution of MAP *terminates* if skip is the only instruction in the MAP program.

3.3.3 REDUCE

REDUCE controls the output actually generated by the enforcement mechanism. REDUCE can retrieve output items from a local output queue of a local execution, send output items to the external output queue, clean local output queues of local executions, wake local executions up, and kill local executions.

Except for the input instruction that is replaced by the retrieve instruc-

$$\text{RETR}\ \frac{\begin{array}{l}\mathsf{red.prg}{:}\pi_R = \textbf{retrieve}\ x\ \textbf{from}\ (j,c)\\ EX[j].\mathsf{out} = O \qquad \text{dequeue}(O,c) = (val, O') \qquad val \neq \bot\end{array}}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{red.mem}{:}m_R \Rightarrow \Delta, \mathsf{red.prg}{:}\textbf{skip}, \mathsf{red.mem}{:}m_R[x \mapsto val]}$$

$$\text{OUTR}\ \frac{\begin{array}{l}\mathsf{red.prg}{:}\pi_R = \textbf{output}\ e\ \textbf{to}\ c\\ \mathsf{red.mem} = m_R \qquad \vec{v} = \vec{\bot}[c \mapsto m_R(e)]\end{array}}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{out}{:}O \Rightarrow \Delta, \mathsf{red.prg}{:}\textbf{skip}, \mathsf{out}{:}O.\vec{v}}$$

$$\text{WAKR}\ \frac{\begin{array}{l}\mathsf{red.prg}{:}\pi_R = \textbf{wake}(PRED[\,])\\ S = \{j \in \{0,\dots,TOP\} : PRED[j]\}\\ \mathsf{LECS} = \bigcup_{j\in S}\{EX[j].\mathsf{int}{:}sig, EX[j].\mathsf{stt}{:}\mathbf{S}\} \qquad \mathsf{LECS}' = \bigcup_{j\in S}\{EX[j].\mathsf{int}{:}\bot, EX[j].\mathsf{stt}{:}\mathbf{E}\}\end{array}}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{LECS} \Rightarrow \Delta, \mathsf{red.prg}{:}\textbf{skip}, \mathsf{LECS}'}$$

$$\text{CLN}\ \frac{\begin{array}{l}\mathsf{red.prg}{:}\pi_R = \textbf{clean}(c, PRED[\,])\\ S = \{j \in \{0,\dots,TOP\} : PRED[j]\}\\ \mathsf{LECS} = \bigcup_{j\in S}\{EX[j].\mathsf{out}{:}O\} \qquad \mathsf{LECS}' = \bigcup_{j\in S}\{EX[j].\mathsf{out}{:}\text{remove}(O,c)\}\end{array}}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{LECS} \Rightarrow \Delta, \mathsf{red.prg}{:}\textbf{skip}, \mathsf{LECS}'}$$

$$\text{KIL}\ \frac{\begin{array}{l}\mathsf{red.prg}{:}\pi_R = \textbf{kill}(PRED[\,])\\ S = \{j \in \{0,\dots,TOP\} : PRED[j]\}\\ \mathsf{LECS} = \bigcup_{0\le j\le TOP} \mathsf{LECS}_j \qquad \mathsf{LECS}' = \text{delete}(\mathsf{LECS}, S)\end{array}}{\Delta, \mathsf{red.prg}{:}\pi_R, \mathsf{top}{:}TOP, \mathsf{LECS} \Rightarrow \Delta, \mathsf{top}{:}TOP - |S|, \mathsf{red.prg}{:}\textbf{skip}, \mathsf{LECS}'}$$

Figure 3.8: Semantics of instructions of REDUCE

tion, in addition to the instructions in Figure 3.1a, the program of the REDUCE component may contain **retrieve** x **from** (j, c), **clean**$(c, PRED[\,])$, **wake**$(PRED[\,])$, or **kill**$(PRED[\,])$ instructions which are described in Figure 3.1c. The execution of the retrieve instruction reads the value from the output queue of $\pi[j]$. The execution of clean, wake, or kill instruction is applied to all local execution $\pi[j]$ such that $PRED[j]$ is true. The execution of clean instruction removes the first vector $\vec{v}$ of the local output queue of $\pi[j]$, where the value of $\vec{v}[c]$ is different from $\perp$. The execution of the kill instruction removes $\pi[j]$ from the array of local execution and updates TOP. The execution of the wake instruction is similar to the one of MAP.

The semantics of retrieve, output, wake clean, and kill instructions is described in Figure 3.8, where function remove(O, c) used in rule CLN removes the first vector $\vec{v}$ in O where $\vec{v}[c] \neq \perp$; assume that LECS is the set of configurations of all local executions (i.e. $\mathsf{LECS} = \bigcup_{0 \leq j \leq TOP} \mathsf{LECS}_j$), and S is a subset of $\{0, \ldots, TOP\}$, function delete(LECS, S) in rule KIL removes configurations of local execution $\pi[j]$ (where $j \in S$) from LECS, and reassign the indexes of the remained local executions such that the order of local executions are maintained (i.e. for local executions $\pi[j]$ and $\pi[k]$ such that j, k are not in S and $j < k$, after the application of the function, these two local executions are referred with new indexes j' and k' such that $j' < k' \leq TOP - |S|$).

When REDUCE is activated, its configuration is $\{\mathsf{red.prg}{:}\pi_R, \mathsf{red.mem}{:}\, m_R\}$, where π_R is the program of REDUCE tailored to the policy to be enforced (for example, for non-interference, the program of REDUCE is described in Figure 4.2b), and m_R is the memory of REDUCE from the last activation. Similar to the execution of MAP, the execution of REDUCE *terminates* if skip is the only instruction in the REDUCE program.

3.4 Summary

Controlled programs can be composed of basic instructions which are assignment, condition, loop, skip, input and output instructions. Local executions are copies of controlled programs. In addition to these instruction, instructions available for MAP are map, wake and clone. For REDUCE, additional instructions are retrieve, wake and kill. The small step operational semantics of controlled programs, local executions, MAP programs, and REDUCE programs was specified.

Chapter 4

Enforcement Mechanisms of Information Flow Policies

This chapter illustrates the framework by presenting constructed enforcement mechanisms of non-interference [20], non-deducibility [45] and deletion of inputs [34]. Soundness and precision of the constructed enforcement mechanisms are formally demonstrated.

4.1 Overview

To illustrate our framework, we next present enforcement mechanisms of non-interference [20], non-deducibility [45], and deletion of inputs [34]. We choose non-interference because it is enforced by secure multi-execution [20], the mechanism that has drawn a lot of attention of researchers [9, 18, 38, 51, 50, 19, 47, 36] to multi-execution technique. Non-deducibility and deletion of inputs are chosen to illustrate the idea that from an existing enforcement mechanism (e.g. the enforcement mechanism of non-interference), by some modification, we can enforce another policy. Deletion of inputs is also used to illustrate the clone instruction.

The chosen information flow policies are defined on a lattice with two security levels: low (L) and high (H). An input or output channel is at either the low level or the high level. Items on channels at the low level is visible to users at the low or high level, while items on channels at the high level is visible only to users at the high level. Function $lvl(c)$ returns the level of channel c.

States of local executions. In the enforcement mechanisms of the selected policies, initially all local executions are in executing states and all local executions are executed in parallel. When a local execution needs an input item that is not ready in its local input queue or when it generates an output item, it emits an interrupt signal and move to a sleeping state. A local execution will be waken up by REDUCE if its generated output item has been processed. A local execution will be waken up by MAP if it has already received the input item that it is waiting for. The condition for a local execution to be waken up is encoded in function *isReady*(c).

From the semantics of the enforcement mechanism (Section 3.3), function isReady(c) is defined as in Equation 4.1, where $EX[x].\mathsf{stt} = \mathbf{S} \wedge EX[x].\mathsf{prg} = \textbf{input } y \textbf{ from } c; \pi$ means that local execution $\pi[x]$ is sleeping and waiting for an input item from c, and $EX[x].\mathsf{in} = I \wedge \text{dequeue}(I, c) = (val, I') \wedge val \neq \bot$ means that $\pi[x]$ has already received the input item that it is waiting for.

$$\begin{aligned}\text{isReady}(c) \triangleq \lambda x.EX[x].\mathsf{stt} = \mathbf{S} \ \wedge \ EX[x].\mathsf{prg} = \textbf{input } y \textbf{ from } c; \pi \ \wedge \\ EX[x].\mathsf{in} = I \ \wedge \ \text{dequeue}(I, c) = (val, I') \ \wedge \ val \neq \bot\end{aligned} \tag{4.1}$$

Activation of MAP and REDUCE. The program of MAP is activated only when the previous execution of MAP has terminated, there is an interrupt signal $\odot(c)$ from local execution $\pi[j]$, the state of this local execution is sleeping ($\mathbf{S}$), and the instruction to be executed is the input instruction. The resulting activation of MAP removes the signal from configuration of

$\pi[j]$.

Function *pick*(S) returns an element from a non-empty set S. Such selection can be non-deterministic or in a round-robin way. The chosen selection does not influence our results. The predicate $WAITI[\,]$ indicates whether a local execution is waiting for an input item or not.

$$WAITI[\,] \triangleq \lambda x.EX[x].\mathsf{stt} = \mathbf{S} \ \wedge \ \exists c \in C_{in} : EX[x].\mathsf{int} = \odot(c) \ \wedge \\ EX[x].\mathsf{prg} = \textbf{input } y \textbf{ from } c; \pi \tag{4.2}$$

Similarly to MAP, REDUCE can be activated when the previous execution of REDUCE has terminated, and there is an interrupt signal $\odot(c)$ from the local execution $\pi[j]$, the state of this local execution is sleeping ($\mathbf{S}$), the instruction to be executed is an output instruction. The resulting activation of REDUCE removes the signal from the configuration of $\pi[j]$. Predicate $WAITO[\,]$ indicates whether a local execution is sleeping on an output instruction.

$$WAITO[\,] \triangleq \lambda x.EX[x].\mathsf{stt} = \mathbf{S} \ \wedge \ \exists c \in C_{out} : EX[x].\mathsf{int} = \odot(c) \ \wedge \\ EX[x].\mathsf{prg} = \textbf{output } e \textbf{ to } c; \pi \tag{4.3}$$

Remark 4.1. *Notice that as reported by Kashyap et al. [25], orders of executing local executions influence security guarantees on timing and termination channels. We believe that the orders of executing local executions reported in their work and the corresponding security guarantees can be incorporated into our constructed enforcement mechanisms.*

Common functions. The enforcement mechanisms of selected policies are based on three functions *canAsk*(j, c), *canBeTold*(c), and *canSend*(j, c), which specify privileges of local executions on input and output channels.

$$\text{MACT}_{\text{SAMPLE}}\ \frac{\begin{array}{ll}\mathsf{map.prg}{:}\mathbf{skip} & \\ S = \{j \in \{0,\dots,TOP\} : WAITI[j]\} & S \neq \emptyset \\ j = \text{pick}(S) & EX[j].\mathsf{prg} = \mathbf{input}\ x\ \mathbf{from}\ c;\pi\end{array}}{\Delta, EX[j].\mathsf{int}{:}\odot(c), \mathsf{map.prg}{:}\mathbf{skip} \Rightarrow \Delta, EX[j].\mathsf{int}{:}\bot, \mathsf{map.prg}{:}\pi_M(j,c)}$$

$$\text{RACT}_{\text{SAMPLE}}\ \frac{\begin{array}{ll}\mathsf{red.prg}{:}\mathbf{skip} & \\ S = \{j \in \{0,\dots,TOP\} : WAITO[j]\} & S \neq \emptyset \\ j = \text{pick}(S) & EX[j].\mathsf{prg} = \mathbf{output}\ e\ \mathbf{to}\ c;\pi\end{array}}{\Delta, EX[j].\mathsf{int}{:}\odot(c), \mathsf{red.prg}{:}\mathbf{skip} \Rightarrow \Delta, EX[j].\mathsf{int}{:}\bot, \mathsf{red.prg}{:}\pi_R(j,c)}$$

Figure 4.1: Activation of MAP and REDUCE in constructed enforcement mechanisms

- Function canAsk(j, c) indicates whether local execution $\pi[j]$ can ask MAP perform input operations on channel c.

- Function canBeTold(c) indicates whether local execution $\pi[x]$ can receive genuine values on channel c from MAP. The application of canBeTold(c) on local execution $\pi[j]$ is written as canBeTold$(c)[j]$.

- Function canSend(j, c) indicates whether local execution $\pi[j]$ can send its generated output values to channel c.

Table 4.1 summarizes ideas of these functions for the enforcement mechanisms.

Another function which is used by the constructed enforcement mechanisms is function *identical*(j) that returns true only when it is applied on j and is defined as below:

$$\text{identical}(j) \triangleq \lambda x.x = j \tag{4.4}$$

Table 4.1: Functions specifying privileges of local executions on channels used in sample enforcement mechanisms

Function	**Non-interference**	**Non-deducibility**	**Deletions of inputs**
canAsk(j, c)	Only the high execution (resp. the low execution) can ask for input items from high channels (resp. low channels).	Only the shadow execution (resp. the low execution) can ask for input items from high channels (resp. low channels).	Privileges for high and low execution are similar to the ones of non-interference. The clones cannot ask any input items.
canBeTold(c)	The high execution can be told all genuine items. The low executions can be told genuine items only from low input channels.	High and shadow executions can be told input items from high input channels. High and low executions can be told genuine input items from low input channels	Privileges for high and low execution are similar to the ones of non-interference. The clones can be told genuine input items from low input channels.
canSend(j, c)	Only the high execution (resp. low execution) can send output items to high channels (resp. low channels).	Similar to non-interference.	Similar to non-interference.

4.2 Enforcement mechanism of non-interference

Let $I|_L$ (resp. $O|_L$) return the projection of the input I (resp. output O) containing only items on channels at the low level. For non-interference, for two arbitrary inputs I and I' that are low-equivalent (i.e. $I'|_L = I|_L$), the generated outputs O and O' are also low-equivalent (i.e. $O'|_L = O|_L$). Non-interference comes in two flavors: termination-sensitive non-interference and termination-insensitive non-interference. Termination-sensitive non-interference is defined with assumption that attackers at the low level can observe the terminations of executions of programs. This assumption is not used in the definition of termination-insensitive non-interference.

Definition 4.2.1 (TINI). *A program π satisfies* termination-insensitive non-interference *iff*

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

Definition 4.2.2 (TSNI). *A program π satisfies* termination-sensitive non-interference *iff*

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

Enforcement mechanism. The enforcement mechanism of non-interference on a controlled program π needs only two local executions: the high execution ($\pi[0]$) and the low execution ($\pi[1]$). The high execution is responsible for asking input items from input channels at the high level and generating output items sent to output channels at the high level. The low execution is responsible for the corresponding ones at the low level. In the sequel, the term *high input items* is used for input items from input channels at the high level. Similarly, we have *high output items*, *low input items*, and *low output items*.

```
1: if $\text{canAsk}_{\text{NI}}(j, c)$ then
2:     input $x$ from $c$
3:     map($x, c, \text{canBeTold}_{\text{NI}}(c)$)
4:     map($\vec{df}[c], c, \neg\text{canBeTold}_{\text{NI}}(c)$)
5:     wake(isReady($c$))
6: else
7:     if $\neg\text{canBeTold}_{\text{NI}}(c)[j]$ then
8:         map($\vec{df}[c], c, \text{identical}(j)$)
9:         wake(identical($j$))
10:    else
11:        skip
12:    end if
13: end if
```

(a) MAP for NI on an input request from local execution $\pi[j]$ on channel c

```
1: if $\text{canSend}_{\text{NI}}(j, c)$ then
2:     retrieve $x$ from $(j, c)$
3:     output $x$ to $c$
4: end if
5: clean($c$, identical($j$))
6: wake(identical($j$))
```

(b) REDUCE for NI on an output request from local execution $\pi[j]$ on channel c

The low execution ($\pi[1]$) cannot ask MAP to perform input operations on high input channels. When it needs a high input item, MAP fetches it a fake value ($\vec{df}[c]$). Only the low execution (resp. the high execution) can send its own generated output items to low channels (resp. high channels).

Figure 4.2: Programs of MAP and REDUCE for the enforcement mechanism of non-interference

In order to enforce non-interference, we ensure that the execution of the low execution is independent from high input items consumed by the high execution. In addition, only the low execution can send output items to low output channels. Thus, high input items do not influence consumed low input items and generated low output items. These ideas are encoded in the programs of MAP and REDUCE that are presented in respectively Figure 4.2a, and Figure 4.2b, where $\vec{df}[c]$ returns the default value for channel c.

Because the low execution is responsible for observation visible at the low level, the low execution can ask MAP to perform input operations on low input channels. The high execution cannot do so since if it can, the consumed high input items influence the consumed low input items. For the low execution, if it can ask MAP to perform an input operation on high input channels, the enforcement mechanism is still non-interferent. However, for the controlled program that is already non-interferent, the inputs consumed by the enforcement mechanism and the inputs consumed by the controlled program might be different. In other words, the enforcement mechanism changes the behavior of programs that is already non-interferent. Thus, MAP only perform input operations on high input channels when it receives input request on these channels from the high execution.

$$\text{canAsk}_{\text{NI}}(j,c) \triangleq (j = 0 \ \wedge \ lvl(c) = H) \ \vee \ (j = 1 \ \wedge \ lvl(c) = L) \tag{4.5}$$

Since the high execution generates high output items, it is natural for the high execution being told genuine input items from high input channels. Because non-interference allows low input items to influence the observation of users at the high level, this execution can also be told genuine input items from low input channels. The low execution cannot receive genuine values from high input channels since if it can, high output items will influence the observation visible to users at the low level. Put differently, the enforcement mechanism fails to enforce non-interference. Thus, the low execution can receive only genuine input items from low input channels.

$$\text{canBeTold}_{\text{NI}}(c) \triangleq \lambda x.(lvl(c) = H \ \wedge \ x = 0) \ \vee \ (lvl(c) = L) \tag{4.6}$$

The high (resp. low) execution is allowed to send its generated output items to high (resp. low) output channels and it is not allowed to send anything to low (resp. high) output channels.

$$\text{canSend}_{\text{NI}}(j,c) \triangleq (j = 0 \ \wedge \ lvl(c) = H) \ \vee \ (j = 1 \ \wedge \ lvl(c) = L) \tag{4.7}$$

When MAP is activated on an input request from local execution $\pi[j]$ on channel c, its path of execution depends on the privileges of local execution

$\pi[j]$ on channel c. If local execution $\pi[j]$ can ask MAP to perform an input operation on channel c (i.e. $\text{canAsk}_{\text{NI}}(j,c) = \mathbf{T}$), MAP gets an input value from the global input queue by performing an input operation (Figure 4.2a-line 2), sends the asked value to all local executions that can be told genuine values from channel c (i.e. $\text{canBeTold}_{\text{NI}}(c)$) (Figure 4.2a-line 3), and sends a fake value to others (Figure 4.2a-line 4). After that, MAP wakes local executions that are sleeping and have already received the input items they are waiting for (Figure 4.2a-line 5). If local execution $\pi[j]$ cannot ask MAP to perform input operations on channel c, and this local execution cannot be told genuine values from c (i.e. $\text{canBeTold}_{\text{NI}}(c)[j] = \mathbf{F}$), MAP sends a fake value to this local execution (Figure 4.2a-line 8) and wakes it up (Figure 4.2a-line 9).

As shown in Figure 4.2b, when REDUCE is activated on an output request from local execution $\pi[j]$ on channel c, it checks whether this execution can send its generated output items to channel c or not. If so (i.e. $\text{canSend}_{\text{NI}}(j,c) = \mathbf{T}$), REDUCE retrieves the generated output item from the local output queue of local execution $\pi[j]$ (Figure 4.2b-line 2) and sends the retrieved output value to c (Figure 4.2b-line 3). Otherwise, no output operation is performed. After that, the output queue of $\pi[j]$ is cleaned (Figure 4.2b-line 5) and $\pi[j]$ is waken (Figure 4.2b-line 6).

Example 4.2.1. *We illustrate the enforcement mechanism of non-interference with the controlled program presented in Figure 3.2 and with input (*$\texttt{cL1} = \boldsymbol{T}$*)(*$\texttt{cH1} = M_1$*), which means that the input contains two vectors, the first one contains* $\boldsymbol{T}$ *from* `cL1`*, and the last one contains* M_1 *from* `cH1`*. Default values for channels* `cH1` *and* `cH2` *are respectively* D_1 *and* D_2*. We assume that* $check(D_1) = \boldsymbol{T}$ *(i.e. true) and* $check(M_1) = \boldsymbol{F}$ *(i.e.* $false$*).*

Executions of high execution and low execution are shown in respectively Figure 4.3a and Figure 4.3b. At Figure 4.3a-line 1, the high execution sends a request to MAP *and moves to a sleeping state.* MAP *is activated. Since* *the high execution can be told low input items but cannot ask for them, there is no input operation performed, no input item sent to the high execution, and this execution keeps sleeping. At the first instruction, the low execution sends a signal to* MAP *and moves to a sleeping state.* MAP *is activated, gets input item* $\boldsymbol{T}$ *from the global queue, sends it to local input queues of both local executions, and wake both local executions up.*

When the output instruction at Figure 4.3a-line 7 of the high execution is executed, REDUCE *is activated. Since the high execution can send its generated output values to* **cH3**, REDUCE *retrieves the output item generated by the high execution, sends the item to the global output queue, cleans the output queue of the high execution, and wakes this execution up. When the high execution executes the output instruction at Figure 4.3a-line 8,* REDUCE *is also activated. Because the high execution cannot send output item to low channels,* REDUCE *does not send any output item to* **cL2**. *Notice that when the low execution needs an input item from* **cH2**, *a fake input item will be fetched to it by* MAP.

The enforcement mechanism terminates and generates output (**cH3** $= M_1 + 0$)(**cL2** $= D_1 + D_2$). *The values sent to* **cH3** *and* **cL2** *are respectively* $M_1 + 0$ *and* $D_1 + D_2$. *We describe the global input, output queues, and local input, output queues in Figure 4.4, where each column in the tables corresponds to an input/output operation.*

At time 0, MAP *consumes* $\boldsymbol{T}$ *from channel* **cL1** *and sends this value to local inputs of high and low executions. At time* 1, MAP *consumes value* M_1 *from channel* **cH1**, *sends this value to the local input of the high execution, and sends a fake value* (D_1) *to the local input of the low execution. At time* 3, *when the low execution requests an input value from channel* **cH2**, MAP *gives it a fake value* (D_2).

At time 3 *and* 4, REDUCE *sends respectively the output value* ($M_1 + 0$) *from the high execution to channel* **cH3** *and the output value* ($D_1 + D_2$)

```
input l1 from cL1                 //Use T asked by π[1].
input h1 from cH1                 //Get M_1 from cH1.
h2 := 0
b := check(h1)                    //b = F = check(M_1).
if l1 and b then
    input h2 from cH2             //This instruction is not executed since b is F.
output h1 + h2 to cH3             //Send M_1 + 0 to cH3.
~~output h1 + h2 to cL2~~         //The output is ignored.
```

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

```
input l1 from cL1                 //Get T from cL1.
input h1 from cH1                 //A fake value D_1 is used.
h2 := 0
b := check(h1)                    //b = T = check(D_1).
if l1 and b then
    input h2 from cH2             //A fake value D_2 is used.
~~output h1 + h2 to cH3~~         //The output is ignored.
output h1 + h2 to cL2             //Send D_1 + D_2 to cL2.
```

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

Figure 4.3: Executions of local copies for non-interference

from the low execution to channel `cL2`*. Output values of the high execution to channel* `cL2` *and output values of the low execution to channel* `cH3` *are ignored by* REDUCE*.*

4.3 Enforcement mechanism of deletion of inputs

Deletion of inputs [34] prevents attackers from deducing the occurrence of the last high input event. The policy 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' are equivalent on the low input events and the high input events, and so are parts α and α'; parts α and α' are also equivalent on low output events.

Input to MAP:

Time / Channel	0	1
cH1	$\perp$	M_1
cH2	$\perp$	$\perp$
cL1	**T**	$\perp$

$\Longrightarrow$ MAP

Local Executions:

High execution $\pi[0]$:

Local input:

cH1	$\perp$	M_1
cH2	$\perp$	$\perp$
cL1	**T**	$\perp$

Local output:

cH3	$\perp$	$\perp$	$\perp$	$M_1 + 0$	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	$M_1 + 0$

Low execution $\pi[1]$:

Local input:

cH1	$\perp$	D_1	$\perp$
cH2	$\perp$	$\perp$	D_2
cL1	**T**	$\perp$	$\perp$

Local output:

cH3	$\perp$	$\perp$	$\perp$	D_1+D_2	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	D_1+D_2

Output by REDUCE:

REDUCE $\Longrightarrow$

Time / Channel	0	1	2	3	4
cH3	$\perp$	$\perp$	$\perp$	$M_1 + 0$	$\perp$
cL2	$\perp$	$\perp$	$\perp$	$\perp$	D_1+D_2

Each column in the table corresponds to an input/output operation. At time 0 and 1 respectively, MAP consumes value **T** and M_1 from channel `cL1` and `cH1`. It sends appropriate values to local inputs of local executions. At time 2, MAP sends a fake value (D_2) to the local input of the low execution. At time 3 and 4, REDUCE sends $M_1 + 0$ and $D_1 + D_2$ to respectively `cH3` and `cL2`.

Figure 4.4: Example of input and output queues for non-interference

In our notation, for a program to satisfy deletion of inputs, on any input such that the execution of the program on this input terminates, the high input item that follows by only low input items or high default items can be replaced by a default item. The obtained input can be modified further by adding or removing default input items and the execution of the program on the resulted input terminates and the output items generated at the low level are not changed.

Put differently, w.r.t. $\vec{df}$ that maps input channels to default values, a program satisfies deletion of input iff for any input $I = I_1.\vec{v}.I_2$, where $\vec{v}$ contains a value from high channel c (i.e. $\vec{v}[c] \neq \bot$ and $lvl(c) = H$) and in I_2 there are either no high items or only high items with default values (i.e. $I_2|_H = (\vec{v}_{df})^*$ and if $\vec{v}_{df}[c]$ contains a value from channel c' then $\vec{v}_{df}[c'] = \vec{df}[c']$), and the execution of the program on I terminates, input I can be changed by replacing $\vec{v}$ by a default vector; 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 input is consumed completely by the program and the generated output is still low-equivalent to the original output generated with input I (i.e. $O'|_L = O|_L$).

Definition 4.3.1 (DI). *W.r.t. vector $\vec{df}$ that maps input channels to default values, a program π satisfies* deletion of inputs *iff*

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

where $\vec{v}_{df}$ is a vector that contains a default value (i.e. if $\vec{v}_{df}$ contains a value from channel c', then $\vec{v}_{df}[c'] = \vec{df}[c']$).

Notice that in the definition of deletion of inputs, $I_2|_H$ and $I'_2|_H$ may have different lengths and may contain inputs from different high input channels.

Enforcement mechanism. The enforcement mechanism of deletion of inputs is similar to the one of non-interference except for the way handling input requests from the high execution on high input channels. Whenever

```
if beCloned($j, c$) then
    clone(identical($j$))
end if
if canAsk$_{\text{DI}}$($j, c$) then
    input $x$ from $c$
    map($x, c$, canTell$_{\text{DI}}$($c$))
    map($\vec{df}[c], c, \neg$canTell$_{\text{DI}}$($c$))
    wake(isReady($c$))
else
    if $\neg$canTell$_{\text{DI}}$($c$)[$j$] then
        map($\vec{df}[c], c$, identical($j$))
        wake(identical($j$))
    else
        skip
    end if
end if
```

(a) MAP for DI on an input request from local execution $\pi[j]$ on channel c

```
if canSend$_{\text{DI}}$($j, c$) then
    retrieve $x$ from $(j, c)$
    output $x$ to $c$
end if
clean($c$, identical($j$))
wake(identical($j$))
```

(b) REDUCE for DI on an output request from local execution $\pi[j]$ on channel c

When the high execution needs a high input item, MAP creates a clone of the high execution. Only the high execution receives the actual high input items. Clones of the high execution and the low execution use fake high input items ($\vec{df}[c]$) sent by MAP. As in the enforcement mechanism of NI, only the high execution (resp. the low execution) can send output items to high output channels (resp. low output channels).

Figure 4.5: Programs of MAP and REDUCE for the enforcement mechanism of deletion of inputs

the high execution ($\pi[0]$) requests a high input item, this execution will be cloned. To ensure that clones do not influence the observation at the low level, we force the clones not to ask MAP perform input operations on low input channels and their outputs to low channels are ignored. Since a clone is responsible to the execution with an input I' in the definition, it does not receive genuine values from high channels and it has to reuse low input items requested by the low execution ($\pi[1]$). A clone cannot send outputs to high channels (only the high execution can send its generated output items to these channels). It is worth noting that clones can influence the termination of the enforcement mechanism.

The program of MAP is in Figure 4.5. The program of REDUCE is as in the one of the mechanism of non-interference. Function beCloned(j, c) is defined as below:

$$\text{beCloned}(j, c) \triangleq j = 0 \ \wedge \ lvl(c) = H \tag{4.8}$$

Functions $\text{canAsk}_{\text{DI}}(j, c)$, $\text{canTell}_{\text{DI}}(c)$, and $\text{canSend}_{\text{DI}}(j, c)$ are as below. Notice that in the enforcement mechanism of deletion of inputs, the number of local executions is not a constant since MAP can create clones of the high execution.

$$\text{canAsk}_{\text{DI}}(j, c) \triangleq (j = 0 \ \wedge \ lvl(c) = H) \ \vee \ (j = 1 \ \wedge \ lvl(c) = L) \tag{4.9}$$

$$\text{canTell}_{\text{DI}}(c) \triangleq \lambda x.(lvl(c) = H \ \wedge \ x = 0) \ \vee \ (lvl(c) = L) \tag{4.10}$$

$$\text{canSend}_{\text{DI}}(j, c) \triangleq (j = 0 \ \wedge \ lvl(c) = H) \ \vee \ (j = 1 \ \wedge \ lvl(c) = L) \tag{4.11}$$

Example 4.3.1. *We illustrate the enforcement mechanism of deletion of inputs with the controlled program presented in Figure 3.2. The input is* $(\texttt{cL1} = \boldsymbol{T})(\texttt{cH1} = M_3)$ $(\texttt{cH2} = M_2)$*, which means that the input contains three vectors, the first vector contains* $\boldsymbol{T}$ *from* **cL1***, the second vector contains* M_3 *from* **cH1***, and the last one contains* M_2 *from* **cH2***. The default values for channels* **cH1** *and* **cH2** *are respectively* D_1 *and* D_2*. We assume* *that check*$(D_1) = \boldsymbol{T}$ *and check*$(M_3) = \boldsymbol{T}$, *where* $\boldsymbol{T}$ *denotes the boolean value true.*

The execution of the high copy is similar to the one in the enforcement mechanism of non-interference, except that when it needs a high input item (Figure 4.6a-line 2 or Figure 4.6a-line 6) , MAP *creates a clone of it. The execution of the low execution* $\pi[1]$ *is the same as the execution of the low copy in the enforcement mechanism of non-interference. The execution of the first clone* $\pi[2]$ *is similar to the execution of the low execution. The second clone* $\pi[3]$ *is created when the high execution executes the input instruction at Figure 4.6a-line 6. At this point, the high execution receives the genuine item from* ***cH2*** *while the second clone receives the default one. All output items generated by clones are ignored by* REDUCE*.*

The enforcement mechanism terminates and generates output $(\textit{cH3} = M_3+M_2)(\textit{cL2} = D_1+D_2)$. *The values sent to* ***cH3*** *and* ***cL2*** *are respectively* $M_3 + M_2$ *and* $D_1 + D_2$. *We describe the global input, output queues, and local input, output queues in Figure 4.7, where each column in the tables corresponds to an input/output operation.*

At time 0, *when receiving the request from the low execution,* MAP *consumes value* $\boldsymbol{T}$ *from channel* ***cL1***, *sends this value to local inputs of all local executions. At time* 1, MAP *consumes* M_3 *from channel* ***cH1*** *when receiving the input request from the high execution. It creates the first clone and sends* M_3 *to the local input of the high execution and sends a fake value* (D_1) *to local inputs of the low execution and the first clone. At time* 2, MAP *consumes* M_2 *from channel* ***cH2*** *and creates the second clone.* MAP *then sends* M_2 *to the local input of the high execution and sends a fake value* (D_2) *to local inputs of other local executions.*

At time 3 *and* 4, REDUCE *sends respectively the output value* (M_3+M_2) *from the high execution to channel* ***cH3*** *and the output value* (D_1+D_2) *from the low execution to channel* ***cL2***. *Output values of the high execution to*

```
1 input l1 from cL1              //Use $\mathbf{T}$ asked by $\pi[1]$.
2 input h1 from cH1              //Get $M_3$ from cH1. The first clone $\pi[2]$ is created.
3 h2 := 0
4 b := check(h1)                 //$b = \mathbf{T} = \mathit{check}(M_3)$.
5 if l1 and b then
6     input h2 from cH2          //Get $M_2$ from cH2. The second clone $\pi[3]$ is created.
7 output h1 + h2 to cH3          //Send $M_3 + M_2$ to cH3.
8 ~~output h1 + h2 to cL2~~      //The output is ignored.
```

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

```
1 input l1 from cL1              //Get $\mathbf{T}$ from cL1.
2 input h1 from cH1              //A fake value $D_1$ is used.
3 h2 := 0
4 b := check(h1)                 //$b = \mathbf{T} = \mathit{check}(D_1)$.
5 if l1 and b then
6     input h2 from cH2          //A fake value $D_2$ is used.
7 ~~output h1 + h2 to cH3~~      //The output is ignored.
8 output h1 + h2 to cL2          //Send $D_1 + D_2$ to cL2.
```

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

```
1 input l1 from cL1              //Use $\mathbf{T}$ asked by $\pi[1]$.
2 input h1 from cH1              //Get $M_3$ from cH1.
3 h2 := 0
4 b := check(h1)                 //$b = \mathbf{T} = \mathit{check}(M_3)$.
5 if l1 and b then
6     input h2 from cH2          //A fake value $D_2$ is used.
7 ~~output h1 + h2 to cH3~~      //The output is ignored.
8 ~~output h1 + h2 to cL2~~      //The output is ignored.
```

(c) The second clone $\pi[3]$

The execution $\pi[2]$ is created when the high execution needs an input item from `cH1`. The execution of this copy is similar to the one of the low. The execution $\pi[3]$ is created when the high execution requests a high input item from `cH2`. When $\pi[3]$ is created, it is sleeping at the input instruction at Figure 4.6c-line 6. All output items generated by $\pi[2]$ and $\pi[3]$ are ignored by REDUCE.

Figure 4.6: Executions of local copies for deletion of inputs

channel **`cL2`***, output values of the low execution to channel* **`cH3`***, outputs of clones to all channels are ignored by* REDUCE*.*

Input to MAP:

Channel \ Time	0	1	2
`cH1`	$\perp$	M_3	$\perp$
`cH2`	$\perp$	$\perp$	M_2
`cL1`	**T**	$\perp$	$\perp$

$\Longrightarrow$ MAP

Local Executions:

High execution $\pi[0]$:

Local input:

`cH1`	$\perp$	M_3	$\perp$
`cH2`	$\perp$	$\perp$	M_2
`cL1`	**T**	$\perp$	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	$M_3 + M_2$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$M_3 + M_2$

Low execution $\pi[1]$:

Local input:

`cH1`	$\perp$	D_1	$\perp$
`cH2`	$\perp$	$\perp$	D_2
`cL1`	**T**	$\perp$	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	$D_1 + D_2$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$D_1 + D_2$

Second clone $\pi[3]$:

Local input:

`cH1`	$\perp$	M_3	$\perp$
`cH2`	$\perp$	$\perp$	D_2
`cL1`	**T**	$\perp$	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	$M_3 + D_2$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$M_3 + D_2$

Output by REDUCE:

REDUCE $\Longrightarrow$

Channel \ Time	0	1	2	3	4
`cH3`	$\perp$	$\perp$	$\perp$	$M_3 + M_2$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$D_1 + D_2$

Each column in the tables corresponds to an input/output operation. At time 0, MAP consumes **T** from `cL1` and sends this value to all local executions. At time 1, MAP consumes M_3 from `cH1`, sends it to the high execution, and sends D_1 to other executions. At time 2, MAP consumes M_2 from `cH2`, sends it to the high execution, and sends D_2 to other executions. At time 3 and 4, REDUCE sends respectively $M_3 + 0$ from the high execution to `cH3`, and $D_1 + D_2$ from the low execution to `cL2`. Local input and local output of the first clone $\pi[2]$ are as of the low execution and hence are not described here.

Figure 4.7: Example of input and output queues for deletion of inputs

4.4 Enforcement mechanism of non-deducibility

Sutherland defines non-deducibility by using two views: the first view corresponds to events that attackers at the low level could not deduce (e.g. high input events), and the second view corresponds to observations of attackers at the low level [45]. 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 sequence of events in the first view took place, because it can be always replaced by another sequence and the observation to the second view is not changed.

Let $I|_H$ return the projection of the input I containing only items at the high level. Termination-insensitive non-deducibility 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 is not changed. Termination-sensitive non-deducibility assumes that attackers can observe terminations of executions. It also assumes the existence of the default view where all input items are default values. If the input with only default values could not be accepted by an execution then it would be possible to deduce that the sequence of genuine high input items is actually different from the sequence of default values.

Definition 4.4.1 (TIND). *A program π satisfies* termination-insensitive non-deducibility *iff*

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

```
if $\text{canAsk}_{\text{ND}}(j, c)$ then
    input $x$ from $c$
    map($x, c, \text{canBeTold}_{\text{ND}}(c)$)
    map($\vec{df}[c], c, \neg\text{canBeTold}_{\text{ND}}(c)$)
    wake(isReady($c$))
else
    if $\neg\text{canBeTold}_{\text{ND}}(c)[j]$ then
        map($\vec{df}[c], c, \text{identical}(j)$)
        wake(identical($j$))
    else
        skip
    end if
end if
```

(a) MAP for ND on an input request from local execution $\pi[j]$ on channel c

```
if $\text{canSend}_{\text{NI}}(j, c)$ then
    retrieve $x$ from $(j, c)$
    output $x$ to $c$
end if
clean($c$, identical($j$))
wake(identical($j$))
```

(b) REDUCE for ND on an output request from local execution $\pi[j]$ on channel c

Only the shadow execution can ask MAP to perform input operations on high input channels. The shadow execution cannot be told genuine input items from low input channels and it has to use fake values for low input items. MAP does not perform input operations when it receives input requests from the high execution. The high execution reuses genuine items asked by shadow and low executions. The high execution is the only one that can send output items to high output channels.

Figure 4.8: Programs of MAP and REDUCE for the enforcement mechanism of non-deducibility

Definition 4.4.2 (TSND). *W.r.t. vector $\vec{df}$ that maps input channels to default values, a program π satisfies* termination-sensitive non-deducibility *iff*

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

and the execution with the input that contains only default values specified by $\vec{df}$ is always present and terminates.

Enforcement mechanism. The enforcement mechanism of non-deducibility has three local executions: the high execution ($\pi[0]$), the shadow execution ($\pi[1]$), and the low execution ($\pi[2]$). To enforce non-deducibility, we configure the mechanism such that the consumed high input items and the consumed low input items are independent. Thus, any combination between the high inputs and the low inputs are always possible.

In order to ensure that the consumed high input items do not influence the consumed low input items, as in the enforcement mechanism of non-interference, we require that MAP performs input operations on low input channels only on input request from the low execution. When this execution needs a high input item, MAP fetches it a default value.

In order to guarantee that low input items do not determine high inputs, we need the shadow execution. Indeed the shadow execution is the only one that can ask MAP to consume input items on high channels. However, the shadow execution can only receives fake (default) low input items. We use the word shadow as its generated output items are ignored by REDUCE (only legitimate output items from the high execution are going to be sent to high output channels).

The high execution here cannot request MAP to perform input operations on high input channels. It can be told genuine values from high input channels (which are asked by the shadow execution), and genuine values from low input channels (which are asked by the low execution). The high execution is the only one that can send output items to high channels.

The programs of MAP and REDUCE are similar to the corresponding ones for the enforcement mechanism of non-interference, except that functions $\text{canAsk}_{\text{ND}}(j, c)$, $\text{canBeTold}_{\text{ND}}(c)$, and $\text{canSend}_{\text{ND}}(j, c)$ are redefined.

$$\text{canAsk}_{\text{ND}}(j, c) \triangleq (j = 1 \ \wedge \ lvl(c) = H) \ \vee \ (j = 2 \ \wedge \ lvl(c) = L) \quad (4.12)$$

$$\begin{aligned}\text{canBeTold}_{\text{ND}}(c) \triangleq \lambda x. \ \ &(lvl(c) = H \ \wedge \ (x = 0 \vee x = 1)) \ \vee \\ &(lvl(c) = L \ \wedge \ (x = 0 \vee x = 2))\end{aligned} \quad (4.13)$$

$$\text{canSend}_{\text{ND}}(j, c) \triangleq (j = 0 \ \wedge \ lvl(c) = H) \ \vee \ (j = 2 \ \wedge \ lvl(c) = L) \quad (4.14)$$

Example 4.4.1. *We illustrate the enforcement mechanism of non-deducibility on the controlled program presented in Figure 3.2 with the input* $(\texttt{cL1} = \boldsymbol{T})(\texttt{cH1} = M_1)$*, which means that the input contains two vectors, the first one contains* $\boldsymbol{T}$ *from* `cL1`*, and the last one contains* M_1 *from* `cH1`*. The fake values for channels* `cH1` *and* `cL1` *are respectively* D_1 *and* $\boldsymbol{F}$*. We assume that check*$(D_1) = \boldsymbol{T}$ *(i.e. true) and check*$(M_1) = \boldsymbol{F}$ *(i.e. false).*

The execution of the low execution is the same as the one in the enforcement mechanism of non-interference. Both high and shadow executions execute instructions from line 1 to 5, and from line 7 to 8. At Figure 4.9b-line 1, the shadow execution consumes a fake value ($\boldsymbol{F}$*) returned by* MAP*. At Figure 4.9b-line 2, it consumes an input from* `cH1`*. All output values generated by the shadow execution at Figure 4.9b-lines 7 and 8 are ignored. The high execution reuses the input items asked by the low execution (Figure 4.9a-line 1) and the shadow execution (Figure 4.9a-line 2). The execution of output instructions of the high execution is the same as the ones in the enforcement mechanism of non-interference.*

The enforcement mechanism terminates and generates output $(\texttt{cH3} = M_1 + 0)(\texttt{cL2} = D_1 + D_2)$*. The values sent to* `cH3` *and* `cL2` *are respectively* $M_1 + 0$ *and* $D_1 + D_2$*. We describe the global input, output queues, and local input, output queues in Figure 4.10, where each column in the tables corresponds to an input/output operation.*

At time 0*, on the input request from the low execution,* MAP *consumes* $\boldsymbol{T}$ *from channel* `cL1` *and sends this value to local inputs of all local executions. At time* 1*, on the input request from the shadow execution,* MAP *consumes* M_1 *from channel* `cH1`*, sends this value to local inputs of high and shadow executions, and sends a fake value (*D_1*) to the local input of the low execution. At time* 2*, when the low execution needs an input item from channel* `cH2`*,* MAP *gives it a fake value (*D_2*).*

At time 3 *and* 4*,* REDUCE *sends respectively the output value (*$M_1 + 0$*)*

```
input l1 from cL1              //Use $\mathbf{T}$ asked by the low execution.
input h1 from cH1              //Use $M_1$ asked by the shadow execution.
h2 := 0
b := check(h1)                 //$b = \mathbf{F} = check(M_1)$.
if l1 and b then
    input h2 from cH2          //This instruction is not executed since $b$ is $\mathbf{F}$.
output h1 + h2 to cH3          //Send $M_1 + 0$ to cH3.
~~output h1 + h2 to cL2~~      //The output is ignored.
```

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

```
input l1 from cL1              //A fake value $\mathbf{F}$ is used
input h1 from cH1              //Get $M_1$ from cH1.
h2 := 0
b := check(h1)                 //$b = \mathbf{F} = check(M_1)$.
if l1 and b then
    input h2 from cH2          //This instruction is not executed since $b$ and $l1$ are $\mathbf{F}$
~~output h1 + h2 to cH3~~      //The output is ignored.
~~output h1 + h2 to cL2~~      //The output is ignored.
```

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

```
input l1 from cL1              //Get $\mathbf{T}$ from cL1.
input h1 from cH1              //A fake value $D_1$ is used.
h2 := 0
b := check(h1)                 //$b = \mathbf{T} = check(D_1)$.
if l1 and b then
    input h2 from cH2          //A fake value $D_2$ is used.
~~output h1 + h2 to cH3~~      //The output is ignored.
output h1 + h2 to cL2          //Send $D_1 + D_2$ to cL2.
```

(c) The low execution $\pi[2]$

Figure 4.9: Executions of local copies for the enforcement mechanism of non-deducibility

from the high execution to channel **cH3** *and the output value* $(D_1 + D_2)$ *from the low execution to channel* **cL2**. *Output values of the high execution to channel* **cL2**, *output values of the low execution to channel* **cH3**, *and output values of shadow executions are ignored by* REDUCE.

Input to MAP:

Time / Channel	0	1
`cH1`	$\perp$	M_1
`cH2`	$\perp$	$\perp$
`cL1`	**T**	$\perp$

$\Longrightarrow$ MAP

Local Executions:

High execution $\pi[0]$:

Local input:

`cH1`	$\perp$	M_1
`cH2`	$\perp$	$\perp$
`cL1`	**T**	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	$M_1 + 0$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$M_1 + 0$

Shadow execution $\pi[1]$:

Local input:

`cH1`	$\perp$	M_1
`cH2`	$\perp$	$\perp$
`cL1`	**F**	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	$M_1 + 0$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	$M_1 + 0$

Low execution $\pi[2]$:

Local input:

`cH1`	$\perp$	D_1	$\perp$
`cH2`	$\perp$	$\perp$	D_2
`cL1`	**T**	$\perp$	$\perp$

Local output:

`cH3`	$\perp$	$\perp$	$\perp$	D_1+D_2	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	D_1+D_2

Output by REDUCE:

REDUCE $\Longrightarrow$

Time / Channel	0	1	2	3	4
`cH3`	$\perp$	$\perp$	$\perp$	$M_1 + 0$	$\perp$
`cL2`	$\perp$	$\perp$	$\perp$	$\perp$	D_1+D_2

Each column in the tables corresponds to an input/output operation. At time 0, MAP consumes **T** from `cL1` and sends it to all local executions. At time 1, MAP consumes M_1 from cH_1, sends it to high and shadow executions, and sends D_1 to the low execution. At time 2, when the low execution needs a value from `cH2`, MAP gives it a fake value. At time 3 and 4, REDUCE sends respectively $M_1 + 0$ from the high execution to `cH3`, and $D_1 + D_2$ from the low execution to `cL2`.

Figure 4.10: Example of input and output queues for non-deducibility

4.5 Soundness of constructed enforcement mechanisms

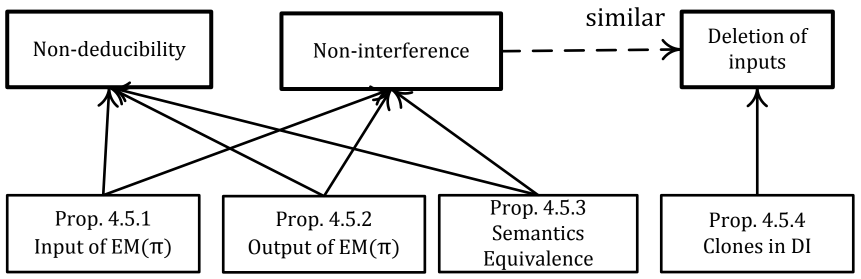
The soundness property states that the instantiated enforcement mechanism correctly enforces the desired policy on an arbitrary program [20, 18]. Soundness does not hold for enforcement mechanisms of sample termination-sensitive policies because one local execution might terminate but the others might not. Thus, the whole enforcement mechanism does not terminate.

Definition 4.5.1. *For a policy $\mathcal{P}$, its enforcement mechanism $\mathsf{EM}_{\mathcal{P}}$ is sound iff for any program π, $\mathsf{EM}_{\mathcal{P}}(\pi)$ satisfies $\mathcal{P}$.*

Theorem 4.5.1. *Enforcement mechanisms of termination-insensitive non-interference, termination-insensitive non-deducibility, and deletion of inputs are sound.*

The proof strategy of soundness is sketched in Fig. 4.11. In order to prove soundness, we state two basic properties specifying the behaviour of MAP on receiving requests for low input items and the behaviour of REDUCE when receiving an output request from a local execution. For the actual proof of the theorem, we perform a case-based reasoning showing that at the end, the output is what we expect. In this respect an important assumption is that the program to be enforced is deterministic. In addition to these two basic properties, we need an additional preliminary property showing the relationship between the execution of a controlled program and the corresponding local execution. For deletion of inputs, we need another preliminary property about the input items that can be consumed by clones ($\pi[j]$ where $j > 1$).

Proposition 4.5.1 (Input items consumed by enforcement mechanisms and input items sent to local input queues of local executions)**.** *Consider the enforcement mechanisms of sample information flow policies, it follows that:*


Non-deducibility
Non-interference
similar
Deletion of inputs
Prop. 4.5.1 Input of EM(π)
Prop. 4.5.2 Output of EM(π)
Prop. 4.5.3 Semantics Equivalence
Prop. 4.5.4 Clones in DI

Proposition 4.5.1 is about inputs consumed by the enforcement mechanisms, and the relationships between those inputs and inputs of local executions. Proposition 4.5.2 is about outputs generated by the mechanisms, and the relationships between those outputs with outputs of local executions. The controlled program and local executions handle their input differently. However, if they receive same input items on all channels, their outputs are the same. This fact is proven in Proposition 4.5.3. Proposition 4.5.4 is about the influence of clones on input and outputs of the enforcement mechanism of deletion of inputs.

Figure 4.11: Proof strategy for soundness of constructed enforcement mechanisms

- MAP *will only ask low input items from the environment for low input requests from the low execution. The low execution can only receive default values for high input items. This local execution can receive real values for low input items.*
- *For the enforcement mechanism of non-interference and deletion of inputs,* MAP *will only ask high input items from the environment for high input requests from the high execution.*
- *For the enforcement mechanism of deletion of inputs, the clones ($\pi[j]$ with $j > 1$) receive real values for low input items, and receive default values for high input items.*
- *For the enforcement mechanism of non-deducibility,* MAP *will ask high input items from the environment for high input request from the shadow execution.*
- *For the enforcement mechanism of non-deducibility, the shadow execution can receive only real values for high input items. It receives default values for low input items.*

- *The high execution of the enforcement mechanism of non-interference, non-deducibility or deletion of inputs can receive real values for low and high input items.*

Proof. The proposition is obvious from the configurations of the corresponding enforcement mechanisms, where all input items in local input queues are sent by MAP; and only MAP can get input items from the environment.

The proposition is proven by using the induction technique on the number of times of activation of MAP on input requests from local executions. Let k be the number of times of activation of MAP on the input request.

For the base case, $k = 0$, we can check that the proposition holds vacuously. Assume that the proposition holds for the case that $k < n$. We now prove that the proposition holds for $k = n$. We consider the n-th activation of MAP on a request from $\pi[j]$ on channel c. Notice that:

- Each instruction of MAP, REDUCE, and local executions is executed atomically,
- MAP and REDUCE are executed separately,
- Local executions are executed separately,
- Local executions interact with MAP and REDUCE via interrupt signals. The execution of a local execution does not influence the executions of MAP and REDUCE.

Therefore, we have:

- Case 1: $j = 0$
  - Case 1.1: $lvl(c) = L$ (the high execution asks a low input item): For non-interference, the instructions at lines 1, 7 and 11 in Fig. 4.2a are executed. For deletion of inputs, the instructions at lines 1, 4

and 14 in Fig. 4.5a are executed. MAP does not perform any input action. This activation does not influence the items received by the low execution. For non-deducibility, the instructions at lines 1, 7, and 11 in Fig. 4.8a are executed. In all the constructed enforcement mechanisms, the high execution keeps sleeping.

- Case 1.2: $lvl(c) = H$ (the high execution asks a high input item): For non-interference, the instructions from line 1 to 5 in Fig. 4.2a are executed. MAP performs an input action and sends a default value to the local input queue of the low execution, and the real value to the local input queue of the high execution. For deletion of inputs, the instructions from line 1 to 8 in Fig. 4.5a are executed. MAP performs actions as in the case of non-interference. In addition, it also makes a clone of the high execution and fetch a default value to local input queues of clones. For non-deducibility, the instructions at lines 1, 7, 11 in Fig. 4.8a are executed. The high execution keeps sleeping.

- Case 2: $j = 1$.

  - Case 2.1: $lvl(c) = L$: For non-interference, the instructions executed are the same as the ones in Case 1.2. For deletion of inputs, the instructions executed are also the same as the ones in Case 1.2 except for the clone instruction at line 2 in Fig. 4.5a that is not executed. In the enforcement mechanisms of non-interference and deletion of inputs, the real value is sent to the local input queues of all local executions. For non-deducibility, the instructions at lines 1, 7, 8, 9 in Fig. 4.8a are executed. MAP sends a default value to the shadow execution.
  - Case 2.2: $lvl(c) = H$: For non-interference, (respectively deletion of inputs), the default value is sent to the local input queue of

$\pi[1]$ by the execution of the instruction at line 8 in Fig. 4.2a (resp. line 11 in Fig. 4.5a). For non-deducibility, the instructions from line 1 to 5 in Fig. 4.8a are executed. MAP performs an input action, sends the read value to high and shadow executions, and sends a default value to the low execution.

- Case 3: $j = 2$ (only for the enforcement mechanism of non-deducibility)
  - Case 3.1: $lvl(c) = H$ (the low execution asks a high input item): the instructions at lines 1, 7, 8, 9 in Fig. 4.8a are executed. MAP sends a default value to the low.
  - Case 3.2: $lvl(c) = L$ (the low execution asks a low input item): the instructions from line 1 to 5 in Fig. 4.8a are executed. MAP performs an input action and sends a default value to the local input queue of the shadow execution, and the real value to local input queues of high execution and low execution.
- Case 4: $j > 1$ (only for the enforcement mechanism of deletion of inputs)
  - Case 4.1: $lvl(c) = L$: (a clone asks a low input item) MAP does not perform any input actions. MAP does not send any input item to the local input queue (line 14 in Fig. 4.5a).
  - Case 4.2: $lvl(c) = H$: (a clone asks a high input item) MAP will send only a default input item to the local input queue of $\pi[j]$ (line 11 in Fig. 4.5a).

The proposition holds for $k = n$. Therefore, the proposition holds for all $k \geq 0$. □

Proposition 4.5.2 (Outputs of enforcement mechanisms). *Concerning the output of enforcement mechanisms of sample information flow policies, it follows that:*

- *For the enforcement mechanism of non-deducibility, non-interference, or deletion of inputs, only the high execution $\pi[0]$ sends output items to high output channels.*

- *For the enforcement mechanism of non-deducibility, non-interference, or deletion of inputs, only the low execution ($\pi[2]$ in the enforcement mechanism of non-deducibility, $\pi[1]$ in the enforcement mechanism of non-interference, or $\pi[1]$ in the enforcement mechanism of deletion of inputs) can send output items to low output channels.*

- *For the enforcement mechanism of deletion of inputs, the output items generated by the local execution $\pi[j]$ with $j > 1$ are ignored.*

Proof. The proposition is proven by using the induction technique on the number of times of activation of REDUCE on output requests from local executions. The proof is similar to the proof of Prop. 4.5.1. □

For the proof of the soundness of the constructed enforcement mechanisms, we need a property stating the relationship between the controlled program and a local execution. We also need another simple property showing how MAP handles the high input requests from local executions.

Proposition 4.5.3 (Controlled programs and local executions)**.** *Let I_1 and I_2 be two input queues, such that for all input channels c, $I_1|_c = I_2|_c$. Then we have: for all programs π,*

$$\forall I_1 : (\pi, I_1, \epsilon) \rightarrowtriangle^k (\pi_k, I_{1_k}, O_k) \implies \forall I_2 : \forall c \in C_{in} : I_1|_c = I_2|_c : (\pi, I_2, \epsilon) \Rightarrow^k (\pi_k, I_{2_k}, O_k)$$

and $I_{1_k}|_c = I_{2_k}|_c$ for all c.

And we have:

$$\forall I_1 : (\pi, I_1, \epsilon) \Rightarrow^k (\pi_k, I_{1_k}, O_k) \implies \exists I_2 : \forall c \in C_{in} : I_1|_c = I_2|_c : (\pi, I_2, \epsilon) \rightarrowtriangle^k (\pi_k, I_{2_k}, O_k)$$

and $I_{1_k}|_c = I_{2_k}|_c$ *for all* c*.*

Proof. The proposition is proven by using the induction technique on k and the length of the input queue I_1, along with the fact that controlled programs and the local executions are deterministic. □

4.5.1 Soundness of mechanism of non-interference

Proof. Let us consider two executions: $(\mathsf{EM}(\pi), I) \Downarrow O$ and $(\mathsf{EM}(\pi), I') \Downarrow O'$, where $I|_L = I'|_L$.

The following holds:

1. The low input items consumed by the enforcement mechanism depends only on the low execution (by Prop. 4.5.1).
2. The low executions in the runs of the enforcement mechanism on I and I' always consume default values for high input items (by Prop 4.5.1).
3. The low executions in these two runs consume the same low input items and same high output items. (By 1, 2 and π be deterministic).
4. These low executions generate the same outputs (By 3 and the fact that π is deterministic).
5. The output items sent to low output channels are always generated by the low executions (by Prop 4.5.2).
6. $O|_L = O'|_L$ (by 4 and 5, and Prop. 4.5.3.)

This concludes the proof. □

4.5.2 Soundness of mechanism of deletion of inputs

To prove the soundness of the enforcement mechanism of deletion of inputs, we need a proposition showing the influence of local execution $\pi[j]$ (with $j > 1$) on the input consumed by the enforcement mechanism.

Proposition 4.5.4. *For the enforcement mechanism of deletion of inputs, clones of the high execution ($\pi[j]$ with $j > 1$) has no effect on the input consumed by the enforcement mechanism.*

Proof. The proof is obvious from the configuration of the enforcement mechanism. □

Proof of Theorem 4.5.1 for the enforcement mechanism of deletion of inputs. The idea of the enforcement mechanism of deletion of inputs is that when the high execution requests a high input item, the high execution will be duplicated and the newly duplicated execution will receive the default values for high input items. If we replace the last high input item in the original input I with a default item, then there exists another input queue satisfying the definition of deletion of inputs. Such an input queue is the input consumed by the $\pi[TOP]$.

Let I be an input, such that $(\mathsf{EM}(\pi), I) \Downarrow O$. The proof of soundness of the enforcement mechanism of deletion of inputs is based on the induction technique on the number of high input item in I.

Base case: since there is no high input item in I, the theorem holds vacuously.

We assume that the theorem holds for all I, such that the number of high input items is smaller than n. We now need to prove that the theorem also holds for the case when the number of high input items is equal to n. Now I can be written as $I_1.\vec{v}_1.\ldots.I_n.\vec{v}_n.I_{n+1}$

Based on the configuration of the enforcement mechanism, $\pi[TOP]$ is created when the high execution requests the last high input item. Let I_{rc}^{TOP} be the input consumed by $\pi[TOP]$, I_{rc}^{1} be the input consumed by $\pi[1]$. We have:

1. $I_{rc}^{TOP} = I_1.\vec{v}_1.\ldots.I_n.\vec{v}_{df}.I'_{n+1}$, where $I'_{n+1}|_H = (\vec{v}_{df})^*$.
2. $I|_L = I_{rc}^1|_L$

Let I^* be an input queue, such that

$$I^* = I_1.\ldots.I_n.I_{n+1}.\vec{v}_1.\ldots.\vec{v}_{n-1}.\vec{v}_{df}.I^*_{n+1},$$

where $I^*_{n+1} = I'_{n+1}|_H$. Assume that the order of executing local executions is first $\pi[1]$, then $\pi[0]$. We have:

3. $I^*|_L = I|_L$.
4. The low execution will consume the part $I_1.\ldots.I_n.I_{n+1}$ for low input items and default values for high input items (by Prop 4.5.1).
5. The high execution $\pi[0]$ will consume $\vec{v}_1.\ldots\vec{v}_{n-1}.\vec{v}_{df}.I^*_{n+1}$ (by 1 and Prop. 4.5.3).
6. For every j, such that $1 < j \leq TOP$, the execution of the local execution $\pi[j]$ terminates and it does not effect the input consumed by the enforcement mechanism (by the assumption that $(\mathsf{EM}(\pi), I) \Downarrow O$ and Prop. 4.5.4).
7. I^* is consumed completely by the enforcement mechanism (by 4 and 5).
8. The high execution terminates (by the assumption that $(\mathsf{EM}(\pi), I) \Downarrow O$).
9. $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$ (by 6, 7, 8).

10. $O^*|_L = O|_L$ (by Prop. 4.5.2 and Prop. 4.5.3).

From 3, 9, and 10, the theorem holds for the case of the number of high input item in I is n. Therefore, the theorem holds for the enforcement mechanism of deletion of inputs. □

4.5.3 Soundness of mechanism of non-deducibility

Proof. For an arbitrary pair input (I, I^*) where $(\pi, I) \Downarrow O$ and $(\pi, I^*) \Downarrow O^*$, we first check the existence of I^{**} such that $I^*|_H = I^{**}|_H$, $I|_L = I^{**}|_L$ and $O|_L = O^{**}|_L$.

Considering an input I such that $(\mathsf{EM}(\pi), I) \Downarrow O$, we have:

1. $(\pi[j], I_j) \Downarrow O_j$, where $0 \leq j \leq TOP$ ($TOP = 2$), and I_j and O_j are respectively the input consumed and output generated by $\pi[j]$.
2. The low execution $\pi[2]$ can ask and be told real values for low input items. It can generate output for low output channels (by Prop. 4.5.1, Prop. 4.5.2) . Therefore: $I|_L = I_2|_L$, $I_2|_H = (\vec{v}_{df})^*$, and $O|_L = O_2|_L$.
3. The shadow execution $\pi[1]$ can ask and be told real values for high input channels. It uses fake values for low input channels (by Prop. 4.5.1, Prop. 4.5.2).
4. The high execution $\pi[0]$ cannot ask but can be told real values from any channel (by Prop. 4.5.1, Prop. 4.5.2).
5. From 3 and 4, $I|_H = I_1|_H$ and $I_1|_L = (\vec{v}_{df})^*$.

We next consider two inputs I and I^* such that $(\mathsf{EM}(\pi), I) \Downarrow O$, and $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$. Let us investigate the running of the enforcement mechanism on I^{**} such that $I^{**}|_H = I^*|_H$ and $I^{**}|_L = I|_L$. Based on the construction of the enforcement mechanism, the running of the enforcement mechanism on I^{**}:

6. From 3, the input consumed by the shadow execution is I_1^{**} where $I^{**}|_H = I^*|_H$ and $I^{**}|_L = (\vec{v}_{df})^*$. Since $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$, the shadow execution terminates$(\pi[1], I_1^{**}) \Downarrow O_1^{**}$.

7. From 2, the input consumed by the low execution is I_2^{**} where $I_2^{**}|_L = I|_L$ and $I_2^{**}|_H = (\vec{v}_{df})^*$. Since $(\mathsf{EM}(\pi), I) \Downarrow O$, the low execution terminates $(\pi[2], I_2^{**}) \Downarrow O_2^{**}$.

8. The high execution consumes partially I^{**}. It cannot influence the input consumed by the mechanism (by Prop. 4.5.1).

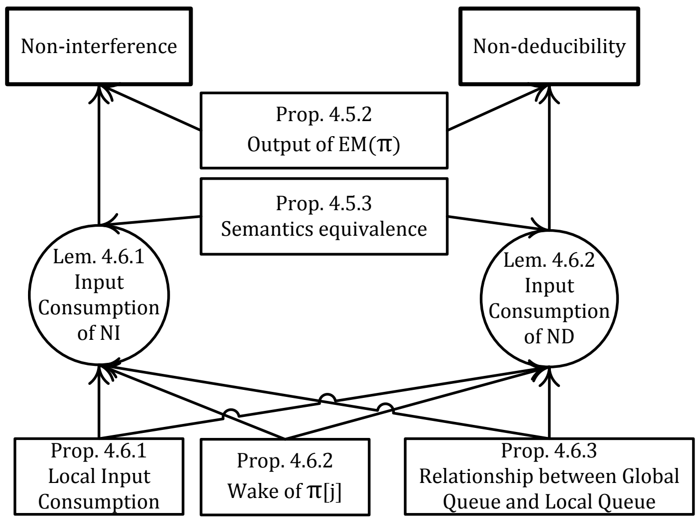
From 6, 7, and 8, there exist I^{**} such that $I^{**}|_H = I^*|_H$ and $I^{**}|_L = I|_L$. If $(\mathsf{EM}(\pi), I^{**}) \Downarrow O^{**}$, then $O^{**}|_L = O|_L$ (by 7, Prop. 4.5.2, and Prop. 4.5.3). In other words, the definition of TIND holds for the arbitrary input pair (I, I^*). This concludes the proof. $\square$

4.6 Precision of constructed enforcement mechanisms

The notion of precision for enforcement of a property is taken from [20, 18]. The intuition is that the instantiated enforcement mechanism 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 4.6.1. *An enforcement mechanism is* precise *w.r.t. a policy iff 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 enforcement mechanism, regardless of the order of executing local executions, are such that the enforcement mechanism terminates and $I^*|_c = I|_c$ and $O^*|_c = O|_c$ for all channels c.*

Theorem 4.6.1. *Enforcement mechanisms of termination-sensitive non-interference, termination-sensitive non-deducibility, deletions of inputs are precise.*



Proposition 4.5.2 is about outputs generated by the mechanisms, and the relationships between those outputs with outputs of local executions. The controlled program and local executions handle their input differently. However, if they receive same input items on all channels, their outputs are the same. This fact is proven in Proposition 4.5.3. Proposition 4.6.1 is about the usage of local inputs of local executions. Proposition 4.6.2 is about the transitions from sleeping states to executing states of local executions. Proposition 4.6.3 is about the relationship between the input of the enforcement mechanism and the input of the high execution. Lemma 4.6.1 is about the input consumed by an enforcement mechanism of non-interference on a program satisfying TSNI. Similarly, Lemma 4.6.2 is about the inut consumed by the enforcement mechanism of non-deducibility on a program satisfying TSND. The proof of the precision of the enforcement mechanism of deletion of inputs follows the structure of the one of non-interference.

Figure 4.12: Proof strategy for precision of constructed enforcement mechanisms

Figure 4.12 shows the proof strategy for precision. The proof of precision is more complex than the proof of soundness. At first, we need to prove a number of simple properties on the correct handling of interrupt signals and the equivalence between the semantics of controlled programs and the semantics of local executions.

Proposition 4.6.1 (Local executions and local input queues)**.** *For a local execution, when the input instruction is executed, if the input item required is in its local input queue, this item will be consumed. Otherwise, an interrupt signal is generated.*

Proof. Proof follows obviously from the semantics of local executions. □

Proposition 4.6.2 (The wake of local executions)**.** *The following facts hold:*

1. *If a local execution is sleeping on an input instruction that required an input item from the channel c, this local execution will be waken up when the input item is ready and the instruction of π_M executed is the wake instruction. In addition, when a local execution is awaken, there is no interrupt signal in its configuration.*
2. *A local execution is not awaken when the input item required is not ready or when the input item required is ready, but the instruction executed of π_M is not the wake instruction.*

Proof. Proof follows by induction on the length of the derivation sequence of the enforcement mechanism. □

Next we show that from $\pi[0]$'s input, we can reconstruct the original global input.

Proposition 4.6.3 (Global input and local inputs)**.** *Let k be the number steps of derivation of the execution of the enforcement mechanism of NI,* *DI, or ND. Assume that we have* $(\mathsf{EM}(\pi), I, \epsilon) \Rightarrow^k (\mathsf{EM}(\pi)_k, I_k, O_k)$*, and* $I^0_{rc_k}$ *is the queue of the input items that have been received by* $\pi[0]$*, then it follows that:*

- $I^0_{rc_k}.I_k = I$

Proof. The lemma is proven by using the induction technique on the length of the global input queue and the length of the derivation sequence of the enforcement mechanism, along with the fact that the execution of the controlled program and the executions of local executions are deterministic. □

4.6.1 Precision of mechanism of non-interference

At this point, we have all that is needed to present the key lemma for the proof of precision for non-interference that shows that all inputs have been processed and there is nothing left within the enforcement mechanism.

Lemma 4.6.1 (Inputs of a controlled program and inputs consumed by the corresponding enforcement mechanism)**.** *Let* π *be a program satisfying termination sensitive non-interference 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 and low input items as in* I*, then it follows that the execution of the enforcement mechanism terminates, and the input consumed by the enforcement mechanism is* I^**, where* $I^*|_c = I|_c$ *for all* c*.*

Proof. The proof of this lemma is based on the proposition of equivalence between semantics of controlled programs and semantics of local executions (Prop. 4.5.3) and the proposition of the relationships between the global input queue and local input queues (Prop. 4.6.3).

According to the semantics of the enforcement mechanism of NI, the high execution does not influence the termination of the low execution, the input consumed and the output generated by the low execution.

Therefore, regardless of the order of executing local executions, if the low execution consumes the same low input items as in I, then the input consumed by the low execution is $I|_L.(\vec{v}_{df})^*.I_a$, where I_a contains only low input items. We next prove that $I_a = \epsilon$ and the low execution terminates.

- Assume that $I_a \neq \epsilon$. This means there exists an execution of π on input I', where $I'|_L = I|_L.I_a$ and $I'|_H = (\vec{v}_{df})^*$. Since π is deterministic, this case cannot happen.

- Assume that $\pi[1]$ does not terminate. However, this leads to the conclusion that π does not satisfy TSNI.

We now prove that the high execution also terminates and does not request any high input item that is not in I.

- Case 1: Assume that the high execution is stuck on a request for low input items. If the high input execution needs a low input item, the enforcement mechanism will behave accordingly to Prop. 4.6.1 and Prop. 4.6.2. The high execution is stuck on low input items when it requests for an input item that is never requested by the low execution. Since the low input items consumed by the low execution is $I|_L$, the stuck of the high execution leads to the conclusion that π is non-deterministic (notice that π and $\pi[0]$ (the high execution) are equivalent, that is if they receive the same input, they have the same behavior).

- Case 2: The high execution requests a high input items that is not in I. Regarding this assumption, because of Prop. 4.5.3, there are two instances of π that consume some input items, but at some point run in different paths of execution. In other words, π is non-deterministic.

- Case 3: The high execution receives all input items it needs, but is in an infinite loop. This case also leads to the conclusion that π is non-deterministic.

Therefore both local executions terminate. Let I^0_{rc} be the input queue received by $\pi[0]$. Since $\pi[0]$ does not request any other input items that are not in I, then $I^0_{rc}|_c = I|_c$. From Prop. 4.6.3, we have $I^* = I^0_{rc}$. Thus $I^*|_c = I|_c$ and $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$.

□

We have now all that is needed for the main theorem.

Proof of Theorem 4.6.1 for the enforcement mechanism of non-interference. Let I be an input queue, such that $(\pi, I) \Downarrow O$. We need to prove that regardless of the order of executing local execution, the input I^* and output Oj will be such that $I^*|_c = I|_c$, $O^*|_c = O|_c$, and $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$.

The proof of precision of the enforcement mechanism of non-interference is based on Lem. 4.6.1 and Prop. 4.5.2. We have:

- Regardless of the order of executing local execution, the input I^* and output O^* will be such that $I^*|_c = I|_c$, and $(\mathsf{EM}(\pi), I^*) \Downarrow O^*$ (by Lem. 4.6.1).

- $O^*|_c = O|_c$ (by Prop. 4.5.2 and π satisfying TSNI).

Therefore, the theorem holds for the enforcement mechanism of non-interference. □

4.6.2 Precision of mechanism of deletion of inputs

The proof for the precision of the enforcement mechanism of deletion of inputs follows the same structure as the one of non-interference. We first prove that regardless of the order of executing local execution, if the controlled program is a good program, then the input consumed by the enforcement mechanism (i.e. I^*) and the input consumed by the original controlled program (i.e. I) are equal on all channels (i.e. $I^*|_c = I|_c$ for all input channels c).

4.6.3 Precision of mechanism of non-deducibility

To prove the precision of enforcement of TSND, we also need a key lemma about the input consumed by the enforcement mechanism.

Lemma 4.6.2 (Input consumption of the enforcement mechanism of non-deducibility)**.** *Let π be a program satisfying termination sensitive non-deducibility, 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 shadow execution consumes the same high input items as in I, then it follows that the execution of the enforcement mechanism terminates, and the input consumed by the enforcement mechanism is I^*, where $I^*|_c = I|_c$ for all c.*

Proof. We have that π satisfies TSND, and $(\pi, I) \Downarrow O$. We write I as $I_H.I_L$ where I_H (resp. I_L) is an input containing only high (resp.) low input items. Let I_D be an input that contains only default values. We consider inputs I^1 and I^2 such that $I^1|_H = I_H$, $I^1|_L = (\vec{v}_{df})^*$, and $I^2|_L = I_L$, $I^2|_H = (\vec{v}_{df})^*$.

Since the program π is deterministic and satisfies TSND, it follows that $I^1|_L = I^D|_L$, $I^2|_H = I^D|_H$, $(\pi, I^1) \Downarrow O^1$, and $(\pi, I^2) \Downarrow O^2$. We next prove that regardless of the order of execution, local executions are not stuck (i.e. all input items necessary for their executions are provided by MAP).

Regardless of the order of executing local executions, if the low execution consumes the same low input items as in I, then the input consumed by the low execution is $(\vec{v}_{df})^*.I_L.I_a$, where I_a contains only low input items. We next prove that $I_a = \epsilon$.

Assume that $I_a \neq \epsilon$. This means there exists an execution on input I', where $I'|_L = I|_L.I_a$. However, this leads to the conclusion that π is not deterministic. Contradiction.

Since π is deterministic, it follows that the low execution terminates. Using the same reasoning, we also prove that the executions of shadow and high executions are not stuck, and these executions terminate. Therefore, the lemma holds. □

Proof of Theorem 4.6.1 for the enforcement mechanism of non-deducibility. The proof follows directly from Lem. 4.6.2, Prop. 4.5.2, and the fact that the controlled program satisfies TSND. □

4.7 Summary

This chapter illustrated the proposed framework by presenting enforcement mechanisms of three information flow policies: non-interference [20], deletion of inputs [34], and non-deducibility [45]. The enforcement mechanism of non-interference is similar to the one proposed by Devriese and Piessens, except that local executions are executed in parallel. Based on the enforcement mechanism of non-interference, by few changes, the modified enforcement mechanisms can enforce non-deducibility and deletion of inputs. From the formal semantics of controlled programs and enforcement mechanisms specified in Chapter 3, this chapter demonstrated that the constructed enforcement mechanisms are sound and precise.

Chapter 5

Testable Hypersafety Policies

This chapter presents the investigation on which policies can be enforced by using the framework on input total reactive programs. Reactive programs have to handles inputs in finite time and have to process completely an input item before handling another one. It shows that the framework can be used to construct enforcement mechanisms for non-empty testable hypersafety policies.

5.1 Reactive programs

We consider deterministic and black box reactive programs. Assumptions on reactive programs are:

1. input total: a program accepts all inputs defined over an enumerable input item set;
2. computable: a program handles any input in finite time and generates an output defined over an enumerable output item set;
3. an input item must be handled completely before another input item can be processed.

Such reactive programs can be implemented in our language by using the pattern in Figure 5.1, where **T** is the boolean value *true*, env_{in} is the only

```
while T do
    input $x$ from $env_{in}$
    //Calculate output o with a terminating and deterministic program.
    output $o$ to $env_{out}$
end while
```

Figure 5.1: Implementation of a reactive program

input channel and env_{out} is the only output channel. Hereafter, without any further notice, by programs, we mean reactive programs following the pattern in Figure 5.1. Since there is only one input channel, we use i to denote an input item. Similarly, we use o to denote an output item.

Notation. As presented in Chapter 3, $\mathbf{I}$ and $\mathbf{O}$ are respectively the enumerable set of input items and the enumerable set of output items. Let $\mathbf{I}^*$ be the set of all finite inputs. An input with length $n \geq 1$ can be expressed as $[i_1. \ \ldots \ .i_n]$. This notation is also used for outputs.

We write $\pi(I) = O$, where I is a finite input, O is a finite output, I and O are of equal length, to mean that from an initial state with input I (i.e. $\{\mathsf{prg}{:}\pi, \mathsf{mem}{:}m_0, \mathsf{in}{:}I, \mathsf{out}{:}\epsilon\}$, where m_0 is the initial memory), π consumes completely I and generates output O. Formally, $\pi(I) = O$ means that there is a sequence of transitions of π from the initial state with input I (i.e. $\{\mathsf{prg}{:}\pi, \mathsf{mem}{:}m_0, \mathsf{in}{:}I, \mathsf{out}{:}\epsilon\}$, where m_0 is the initial memory) to $\{\mathsf{prg}{:}\pi, \mathsf{mem}{:}m', \mathsf{in}{:}\epsilon, \mathsf{out}{:}O\}$, where m' is a the memory of the program after it finishes handling input I. Notice that for a program π follows the pattern in Figure 5.1, the program before consuming any input I and the program after consuming I are the same.

A program π is *input total* if for any I, there exists O such that $\pi(I) = O$. A program π is *deterministic* if for any I, O and O', if $\pi(I) = O$ and $\pi(I) = O'$ then $O = O'$.

A *primitive observation* is a pair (I, O), where I and O are respectively a finite input and a finite output, and of equal length. A program π has the primitive observation (I, O) iff $\pi(I) = O$. An *observation* M is a finite set of primitive observations. M is an observation of program π (denoted by $M \subseteq \pi$) iff all the primitive observations in M is of π.

An observation M is *prefixed-closed* if $(I.i, O.o)$ in M then (I, O) is also in M. The *prefix closure* of M (denoted by $\overline{M}$) is the smallest set that includes M and is prefixed-closed. M is *possible* if it can be exposed by a deterministic reactive program, i.e if (I, O) and (I, O') are in $\overline{M}$, then O and O' are equal. Hereafter, we consider only possible observations. We use **OBS** to denote the set of all observations.

We write input(M) for the set $\{I \mid (I, O) \in M\}$. Given a finite set of inputs Is, map(π, Is) returns the observation of π on Is:

$$\text{map}(\pi, Is) = \{(I, O) | I \in Is \wedge \pi(I) = O\}$$

Example 5.1.1. *Let* $\boldsymbol{I} = \boldsymbol{O} = \{a, b\}$*. The prefix closure of* $M_1 = \{([a.b], [a.b])\}$ *is* $\overline{M}_1 = \{(\epsilon, \epsilon), ([a], [a]), ([a.b], [a.b])\}$*. Observation* $M_2 = \{([a.b], [a.b]), ([a], [b])\}$ *is not a possible observation since there is no deterministic reactive program* π *such that* $\pi([a]) = [a]$ *and* $\pi([a]) = [b]$*.*

5.2 Policies

A policy $\mathcal{P}$ can be defined as the set of programs allowed by the policy. Membership of the policy is required to be compatible with observational equivalence: if $\pi \in \mathcal{P}$ then all programs observationally equivalent with π must also be in $\mathcal{P}$. Hence, one can also think of a policy as a set of sets of primitive observations: a program satisfies the policy iff the set of primitive observations of the program is an element of the policy [17].

In [17], Clarkson and Schneider define hypersafety policies. Informally, if a program is not in a hypersafety policy $\mathcal{P}$, then this program has an observation disallowed by the policy [17, 16].

Definition 5.2.1 ([17, 16]). *A policy $\mathcal{P}$ is a* hypersafety policy *iff*

$$\forall \pi \notin \mathcal{P} \implies \exists M_{bad} \in \mathbf{OBS}.M_{bad} \subseteq \pi \wedge (\forall \pi'.M_{bad} \subseteq \pi' \implies \pi' \notin \mathcal{P})$$

Hypersafety policies can be specified by defining a set $\mathcal{M}$ of bad or disallowed observations. The corresponding policy $\mathcal{P}$ is then defined as: $\pi \notin \mathcal{P}$ if and only if π has one of the specified bad observations $M_{bad} \in \mathcal{M}$. However, for a given hypersafety policy, $\mathcal{M}$ need not be unique. For example, one may choose $\mathcal{M}$ to be the set containing any one M_{bad} for every $\pi \notin \mathcal{P}$.

Example 5.2.1. *Non-interference (NI), the policy that low (L) outputs do not depend on high (H) inputs is a hypersafety policy, and can be specified by defining a set of disallowed observations as follows. Let lvl be a function that assigns H or L to input and output values. Given an input I, let $I|_L$ be the resulting input after filtering out the input values i with $lvl(i) = H$ (and similarly for $O|_L$).*

A reactive program π is non-interferent *(or $\pi \in \mathcal{P}_{NI}$) iff*

$$\forall I, I' \in \boldsymbol{I}^* : I|_L = I'|_L \implies O|_L = O'|_L,$$

where $\pi(I) = O$ and $\pi(I') = O'$.

A set of bad observations that specifies $\mathcal{P}_{NI}$ is:

$$\{\{(I, O), (I', O')\} \mid I|_L = I'|_L \wedge \; O|_L \neq O'|_L\}.$$

A program π that has an observation in this set is not non-interferent. If π does not have any such observation, then it is non-interferent.

5.2.1 Testable hypersafety policies

A hypersafety policy can be specified by different sets of bad observations. A canonical set of bad observations for a given policy is the *maximal* set.

Definition 5.2.2. *For a hypersafety policy $\mathcal{P}$, we define $\mathcal{M}_{\mathcal{P}}$, the* maximal *set of bad observations as:*

$$\mathcal{M}_{\mathcal{P}} = \{M_{bad} \mid \forall\pi : M_{bad} \subseteq \pi \implies \pi \notin \mathcal{P}\}$$

An observation M is allowed *by a policy $\mathcal{P}$ iff $M \notin \mathcal{M}_{\mathcal{P}}$.*

It is *maximal* in the sense that any other set $\mathcal{M}$ that specifies the same policy $\mathcal{P}$ is a subset of $\mathcal{M}_{\mathcal{P}}$. For any hypersafety policy $\mathcal{P}$, the maximal set $\mathcal{M}_{\mathcal{P}}$ always exists.

Definition 5.2.3. *A hypersafety policy $\mathcal{P}$ is* testable *iff membership in $\mathcal{M}_{\mathcal{P}}$ is decidable.*

For the construction of enforcement mechanisms, we limit our attention to testable hypersafety policies. Such policies can be specified by giving a total computable boolean function reject(M) that for an observation M returns $\mathbf{T}$ iff $M \in \mathcal{M}_{\mathcal{P}}$.

It is non-trivial to check whether a set of bad observations specified by a reject function is actually maximal. For example, the set of bad observations that we specified in Example 5.2.1 for NI is *not* maximal. It does not contain observations that have violated the policy in the past but where things have "re-adjusted" as the execution progressed, as shown in the following example.

Example 5.2.2. *Suppose that $\boldsymbol{I} = \boldsymbol{O} = \{0, 1\}$ and that $lvl(0) = L$ whereas $lvl(1) = H$. Consider the following observation $\{([0.1.0], [0.1.0]), ([1.0.0], [0.0.1])\}$. This observation is possible and it satisfies the simple test presented in Example 5.2.1 because the outputs are equivalent $[0.1.0]|_L = [0.0] = [1.0.0]|_L$. However, no program that satisfies the non-interference policy can generate this observation because it will have to first generate the observation $\{[0.1, 0.1], ([1.0], [0.0])\}$ which would violate the policy.*

Fortunately, the maximal set for non-interference is still decidable.

Example 5.2.3 (NI, a testable hypersafety policy)**.** *For the non-interference policy discussed in Example 5.2.1, a reject predicate can be constructed as follows:*

$$\mathit{reject}(M) = \begin{cases} \boldsymbol{T} & \exists M_{bad} = \{(I,O),(I',O')\}, \\ & \quad M_{bad} \subseteq \overline{M} \text{ s.t. } I|_L = I'|_L \text{ and } O|_L \neq O'|_L, \\ \boldsymbol{F} & \text{otherwise.} \end{cases}$$

It is straightforward to check that this is a total computable function. We show that it specifies the maximal set of bad observations by contradiction.

Suppose that there is an observation M such that reject$(M) = \boldsymbol{F}$ and $M \in \mathcal{M}_{\mathcal{P}}$. By construction of the reject() function we have that reject$(\overline{M}) = \boldsymbol{F}$. Since $\mathcal{M}_{\mathcal{P}}$ is maximal, $\overline{M} \in \mathcal{M}_{\mathcal{P}}$.

By definition of $\mathcal{M}_{\mathcal{P}}$, all programs π such that $\overline{M} \subseteq \pi$ must not belong to the policy. Consider now one of such programs π_{good} such that for all $(I,O) \in \overline{M}$, $\pi_{good}(I) = O$ and otherwise it always outputs a value o with $lvl(o) = H$. This program satisfies the policy. Contradiction.

5.2.2 Incrementally constructing observations allowed by a policy

An enforcement mechanism must not only decide whether or not an observation is rejected by the policy. It must also "correct" programs that turn out to have observations that are not allowed (for instance by terminating the program, or more generally by modifying the outputs of the program).

Given an observation M of the untrusted program that is still allowed so far, when we find for the next input item i that the corresponding output will lead to a violation of the policy, we need to find another output that will *not* lead to a violation of the policy.

Definition 5.2.4. *Given a set of bad observations $\mathcal{M}$ that specifies a hypersafety policy, a function extend$_{\mathcal{M}}(M, I, i)$ is an* extension function *for $\mathcal{M}$ iff, for any observation $M \notin \mathcal{M}$, where $(I, O) \in M$, it returns an $o \in \boldsymbol{O}$ such that $M \cup \{(I.i, O.o)\} \notin \mathcal{M}$.*

One of the reasons why it is useful to work with the maximal set of bad observations to specify a policy is that for the maximal set, an extension function always exists.

Proposition 5.2.1. *For any hypersafety policy $\mathcal{P}$, there exists an extension function for the maximal set of bad observations $\mathcal{M}_{\mathcal{P}}$.*

Proof. If the policy $\mathcal{P}$ is empty then all observations are bad observations (in $\mathcal{M}_{\mathcal{P}}$) and therefore the precondition for the applicability of the extension function is false, and we are done.

If the policy is not empty, consider an observation M that is allowed by $\mathcal{M}_{\mathcal{P}}$. By definition of maximal set of bad observations there must be a program π_{good} such that $M \subseteq \pi_{good}$ and $\pi_{good} \in \mathcal{P}$ (if none existed M would have been in $\mathcal{M}_{\mathcal{P}}$).

Let I be an arbitrary input such that $(I, O) \in M$ and i an arbitrary input value. By definition of observation on a program, it must be $\pi_{good}(I) = O$. Since programs are input total, $\pi_{good}(I.i)$ has the form $O.o$ for some $o \in \mathbf{O}$. Pick this o as the return value for extend$_{\mathcal{M}_{\mathcal{P}}}(M, I, i)$.

Suppose now $M \cup \{(I.i, O.o)\} \in \mathcal{M}_{\mathcal{P}}$. Since $M \cup \{(I.i, O.o)\} \subseteq \pi_{good}$ by definition of maximal set of bad observation it should be $\pi_{good} \notin \mathcal{P}$. Contradiction. Therefore the set $M \cup \{(I.i, O.o)\}$ is also allowed by $\mathcal{M}_{\mathcal{P}}$. $\square$

For a given hypersafety policy $\mathcal{P}$, we write extend$_{\mathcal{P}}(M, I, i)$ as an abbreviation for some function extend$_{\mathcal{M}_{\mathcal{P}}}(M, I, i)$ that is guaranteed to exist by the proposition above.

Interestingly, for testable hypersafety policies, there is always a *total computable* extension function.

Proposition 5.2.2. *Let $\mathcal{P}$ be a testable hypersafety policy, then there exists a total computable extension function for $\mathcal{M}_{\mathcal{P}}$.*

Proof. Let M be an observation, I be an arbitrary input in input(M) and i be an arbitrary input value. The total computable extension function is constructed as follows:

1. if the first argument M already belongs to $\mathcal{M}_{\mathcal{P}}$ then the function returns an arbitrary output value; (in this case the precondition for the extension function is not satisfied, and we can return any value)
2. otherwise the function enumerates all output values o and submits each observation $M \cup \{(I.i, O.o)\}$ to the total computable membership test for $\mathcal{M}_{\mathcal{P}}$ continuing until the reject() function returns false.

Since (by Proposition 5.2.1) an o_{good} exists such that $M \cup \{(I.i, O.o_{good})\}$ is not in $\mathcal{M}_{\mathcal{P}}$ this procedure terminates. $\square$

The function constructed in the proof of Proposition 5.2.2 is not very efficient. Some policies admit much more efficient ways of extending allowed observations.

Example 5.2.4. *For the non-interference policy, we can define $extend_{\mathcal{P}_{NI}}(M, I, i)$ as follows. Let o_H be an arbitrary output value with $lvl(o_H) = H$.*

1. *if $(I.i, O.o)$ is in $\overline{M}$, then return o,*
2. *else if $lvl(i) = H$, then return o_H,*
3. *else if $lvl(i) = L$:*
   (a) *if there exists $(I'.i, O'.o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$ then return o'*

(b) otherwise, return o_H.

We show that this is a correct extension function by contradiction.

Suppose there exists an input I, an input value i, and an output O such that $(I,O) \in M$, reject$(M) = \boldsymbol{F}$, and reject$(M \cup \{(I.i, O.o)\}) = \boldsymbol{T}$, where the reject( )predicate is specified as in Example 5.2.3 and $o =$ extend$_{\mathcal{P}_{NI}}(M, I, i)$ is the result of the above algorithm.

By construction of the reject() function we also have reject$(\overline{M'}) =$ reject(M') for all M'. Let $M_1 = M \cup \{(I.i, O.o)\}$, then by hypothesis and the properties of the reject() predicate we have that reject$(\overline{M_1}) = \boldsymbol{T}$. Further, since $(I,O) \in M$ we have that $\overline{M_1} = \overline{M} \cup \{(I.i, O.o)\}$.

- *If $(I.i, O.o)$ is in $\overline{M}$, then $\overline{M_1} = \overline{M}$. Thus, reject$(\overline{M}) = \boldsymbol{T}$. Contradiction.*

- *If $lvl(i) = H$, then $o = o_H$, hence $lvl(o) = H$. Since reject$(M_1) = \boldsymbol{T}$, there exists (I_b, O_b) in $\overline{M}$ s.t $I_b|_L = I.i|_L$ and $O_b|_L \neq O.o|_L$. Because $lvl(i) = lvl(o) = H$, it follows that $I.i|_L = I|_L$ and $O.o|_L = O|_L$. But then $I_b|_L = I|_L$ and $O_b|_L \neq O|_L$. Thus, reject$(\overline{M}) = \boldsymbol{T}$. Contradiction.*

- *If $lvl(i) = L$ and there exists $(I'.i, O'.o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$, then o is o'. Since reject$(\overline{M_1}) = \boldsymbol{T}$, there exists (I_b, O_b) in $\overline{M}$ s.t $I_b|_L = I.i|_L$, and $O_b|_L \neq O.o|_L$. But $I'.i|_L = I.i|_L$ and $O'.o'|_L = O.o|_L$, and since both (I_b, O_b) and $(I'.i, O'.o')$ are in $\overline{M}$ it follows that reject$(\overline{M}) = \boldsymbol{T}$. Contradiction.*

- *If $lvl(i) = L$ and there is no $(I'.i, O'.o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$, then $o = o_H$. Since reject$(\overline{M_1}) = \boldsymbol{T}$, there must exist I_b in input$(\overline{M})$ s.t $I_b|_L = I.i|_L$. Let I'_b be the prefix of I_b that removes all elements with level H at the end of I_b. Then I'_b is in input$(\overline{M})$, and it must have the form $I''_b.i$ and it must have $I''_b|_L = I_L$. Contradiction.*

5.3 General enforcement mechanism

An important question needs to be addressed for a general enforcement mechanism is that which alternative executions the enforcement mechanism should look at. Another one is how to correct executions consistently.

5.3.1 Generating sufficient test inputs

For hypersafety policies like non-interference, the enforcement mechanism should not only look at the input/output (I, O) of the current execution. For such policies, it should also make sure that other primitive observations that the policy defines to be incompatible with the current execution do not exist, and in general there will be infinitely many alternate inputs that can possibly lead to incompatible observations.

Example 5.3.1. *For the non-interference policy, for a given primitive observation (I, O), the set of all other primitive observations that are incompatible with (I, O) is:*

$$\{(I', O') \quad | \quad I|_L = I'|_L \wedge O'|_L \neq O|_L\}$$

This set contains an infinite number of inputs I', so the enforcement mechanism can not query the program π with all of them in finite time.

Fortunately, it is not necessary to check the behavior of the program on *all* inputs that might be potentially conflicting with the current input.

We introduce the notion of *test generator*, a function that computes a finite and sufficient set of alternative inputs to check.

Definition 5.3.1. *A* test generator *for a hypersafety policy $\mathcal{P}$ is a function $g : \boldsymbol{I}^* \rightarrow$ finite $2^{\boldsymbol{I}^*}$ s.t. for all π and all $M_{bad} \in \mathcal{M}_{\mathcal{P}}$:*

$$\forall I \in input(M_{bad}), map(\pi, g(I) \cup \{I\}) \notin \mathcal{M}_{\mathcal{P}} \implies M_{bad} \not\subseteq \pi$$

In other words, if a program π has a bad observation M_{bad}, then there is at least one input $I \in \text{input}(M_{bad})$ for which $g(I)$ is a sufficiently large set of inputs such that testing the program π on these inputs *in addition to* the actual input I will detect a policy violation.

Lemma 5.3.1. *If, for every I, map$(\pi, g(I) \cup \{I\}) \notin \mathcal{M}_{\mathcal{P}}$, then $\pi \in \mathcal{P}$.*

Proof. Suppose $\pi \notin \mathcal{P}$. By definition, there is a bad observation $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ such that $M_{bad} \subseteq \pi$.

From the property of generators in Definition 5.3.1, it follows that there exists an $I \in \text{input}(M_{bad})$ such that $\text{map}(\pi, g(I) \cup I) \in \mathcal{M}_{\mathcal{P}}$. But this contradicts the condition of the lemma. □

Example 5.3.2. *For the non-interference policy from Example 5.2.1 (with reject specified in Example 5.2.3), $g(I) = \{I|_L\}$ is a test generator.*

Let $M_{bad} \in \mathcal{M}_{\mathcal{P}_{NI}}$. This means there must exist $(I, O), (I', O') \in M_{bad}$ with $I|_L = I'|_L$ and $O|_L \neq O'|_L$.

Now, suppose $M_{bad} \subseteq \pi$, i.e. $\pi(I) = O$ and $\pi(I') = O'$. We show that π has a bad observation either on inputs I and $I|_L$, or on inputs I' and $I'|_L$.

Consider the output O'' of π on $I|_L = I'|_L$. Since $O|_L \neq O'|_L$, we must have either that $O'' \neq O|_L$ or $O'' \neq O'|_L$.

- *If $O'' \neq O|_L$, then π has a bad observation on inputs I and $I|_L$.*
- *If $O'' \neq O'|_L$, then π has a bad observation on inputs I' and $I|_L$.*

This implies that g is a generator.

We can further reduce the size of the generator. Let us define g' as follows:

$$g'(I) \quad = \quad \textit{if } I|_L = I \textit{ then } \{\} \textit{ else } \{I|_L\}$$

Since for all I, $g(I) \cup I = g'(I) \cup I'$, it follows easily from Definition 5.3.1 that g' is a generator if g is a generator.

5.3.2 Consistently correcting executions

A second challenge that needs to be addressed is *how to correct executions.* While processing an input I, the enforcement mechanism will explore other inputs in order to check that there are no bad observations. So the output for input I will be computed in different circumstances: while the enforcement mechanism is actually processing I, as well as while the mechanism's actual input is another input I' and the input I is only considered as a potential candidate jointly with I' for membership in a bad set. The enforcement mechanism should compute the same output for I in any of these circumstances.

Example 5.3.3. *Assume that $\boldsymbol{I} = \boldsymbol{O} = \{0,1\}$. $I \oplus I'$ is defined for I and I' of equal length as a pair-wise xor of I and I', where $(0 \oplus 1) = (1 \oplus 0) = 1$ and $(0 \oplus 0) = (1 \oplus 1) = 0$. Similarly, we define $O \oplus O'$. Let $\boldsymbol{1}$ (resp. $\boldsymbol{0}$) denote an input or output consisting of only 1 values (resp. 0 values).*

A program π satisfies a policy $\mathcal{P}_{xor}$ iff

$$\forall I, I' : I \oplus I' = \boldsymbol{1} \Rightarrow O \oplus O' = \boldsymbol{1}$$

The reject function that decides $\mathcal{M}_{\mathcal{P}_{xor}}$ is as follows:

$$reject(M) = \begin{cases} \boldsymbol{T} & \exists M_{bad} = \{(I,O),(I',O')\}, \\ & \qquad M_{bad} \subseteq \overline{M} \text{ s.t. } I \oplus I' = \boldsymbol{1} \text{ and } O \oplus O' \neq \boldsymbol{1}, \\ \boldsymbol{F} & \text{otherwise.} \end{cases}$$

We define $g(I) \triangleq \{I' \mid I \oplus I' = \boldsymbol{1}\}$. Obviously, given an I, such I' is unique. It is easy to show that it actually is a test generator.

Now consider a naive construction of an enforcement mechanism $\mathsf{EM}_{\mathcal{P}_{xor}}$ that on execution of a program π on input I, checks the behavior of π also on $g(I) = \{I'\}$. If the enforcement mechanism finds that $\pi(I) \oplus \pi(I') \neq \boldsymbol{1}$, it corrects the output for $\pi(I)$ to make it compliant with the policy. For *instance, let* π *be the program that just outputs* 0 *for any input value (i.e.* $\pi(I) = \boldsymbol{0}$ *for any* I*). If* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ *is executed on* $[0]$*, the enforcement mechanism would see that* $\pi([0]) = [0]$ *and that* $\pi([1]) = [0]$*, and it would decide to correct the output for* $[0]$ *to* $[1]$*. It is easy to see that* $\mathsf{EM}_{\mathcal{P}_{xor}}$ *is not a secure enforcement mechanism, because if* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ *is executed on* $[1]$*, it would see that* $\pi([1]) = [0]$ *and that* $\pi([0]) = [0]$*, and it would decide to correct the output for* $[1]$ *to* $[1]$*. Essentially* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ *will be a program that always outputs* 1 *on every input, and it violates the policy* $\mathcal{P}_{xor}$ *as badly as* π *does.*

This is an example of inconsistent corrections: on execution of $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ *on* $[0]$*, we are considering the alternate execution* $[1]$*, but we are not taking into account that the alternate execution might as well be corrected if it were ever executed.*

Guaranteeing consistency of corrections is challenging. One idea is to use recursive invocations of the enforcement mechanism while checking alternative inputs.

Example 5.3.4. *For the example above, if* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ *is executed on* $[0]$*, the enforcement mechanism would see that* $\pi([0]) = [0]$ *and then it should not check this against* $\pi([1])$*, but against* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)([1])$*. Unfortunately, for the given generator, this would lead to divergence, as* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)([1])$ *will again recursively call* $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)([0])$*.*

We address the issue of divergence by means of the notion of well-founded test generator: a generator is *well-founded*, if there exists a well-founded partial order $\sqsubset$ on the set of finite inputs, such that:

- $I \sqsubset I.i$
- $\forall I' \in g(I), I' \sqsubset I$

Now, we can recursively call the enforcement mechanism on alternative inputs generated by the generator, and this will make sure that corrections are done consistently.

Example 5.3.5. *Consider again the $\mathcal{P}_{xor}$ policy. We now propose the following generator: $g(I.0) \triangleq \{\}$ and $g(I.1) \triangleq \{I'.0 | I.1 \oplus I'.0 = \boldsymbol{1}\}$.*

This is a well-founded generator. The partial order $\sqsubset$ can be defined as (the transitive closure of) $I \sqsubset I.i$ and $I.0 \sqsubset I.1$.

Now consider again an enforcement mechanism $\mathsf{EM}_{\mathcal{P}_{xor}}$ that on execution of an untrusted program π on input I, checks the behaviour of $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ also on $g(I)$. Now the enforcement mechanism will let any program π do its original output on 0s and it will correct the output on 1s so that the policy is satisfied.

For instance, let π again be the program that just outputs 0 for any input value. If $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ is executed on $[0]$, the enforcement mechanism would output $[0]$. If $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ is executed on $[1]$, our algorithm would see that $\pi([1]) = [0]$ and that $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)([0]) = [0]$, and it would decide to correct the output for $[1]$ to $[1]$. Essentially $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ will now be a program that echoes inputs on outputs, and hence it is a secure program. The recursive calls to $\mathsf{EM}_{\mathcal{P}_{xor}}(\pi)$ always terminate thanks to the well-founded generator.

For every non-empty testable hypersafety policy, there is a well-founded test generator.

Lemma 5.3.2. *Every non-empty testable hypersafety policy has a well-founded generator.*

Proof. Construct an enumeration of $\mathbf{I}^*$ (the set of finite inputs) that has the property that I is enumerated before $I.i$ for all i, I. The constructed enumeration defines a total order on $\mathbf{I}^*$. Say $I \sqsubset I'$ if I is enumerated before I'. Define the generator $g(I) = \{I' \mid I' \sqsubset I\}$. It is easy to check that this is a generator: for any M_{bad}, let I be the maximal element in the set $inputs(M_{bad})$. Then $inputs(M_{bad}) \subseteq g(I) \cup \{I\}$. Since π is deterministic and $inputs(M_{bad}) \subseteq g(I) \cup \{I\}$, if $M_{bad} \subseteq \pi$ then $M_{bad} \subseteq \mathrm{map}(\pi, g(I) \cup \{I\})$, and hence $\mathrm{map}(\pi, g(I) \cup \{I\}) \in \mathcal{M}_{\mathcal{P}}$ (from the property of maximal set of bad observations). $\square$

5.3.3 General enforcement mechanism

We denote the general enforcement mechanism of a non-empty testable hypersafety policy $\mathcal{P}$ on a program π by $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$. The enforcement mechanism is constructed with three computable functions that are reject, extension, and generator functions. Notice that for a testable hypersafety policies, the existence of these three functions are guaranteed.

To simplify the presentation, following the notation used in [37], in this section, we use $\langle \pi, m' \rangle$ to denote a configuration of the controlled program, where π is the instruction to be executed and m' is the memory. We abuse $\twoheadrightarrow$ and write $\langle \pi, m' \rangle \overset{i|o}{\twoheadrightarrow} \langle \pi, m'' \rangle$ to denote that on $\langle \pi, m' \rangle$, the program consumes input i, generates output o and moves to $\langle \pi, m'' \rangle$. Notice that the instructions to be executed before consuming the input and after generating the output are the same.

A state of the enforcement mechanism $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ is a tuple $\langle I, \pi, m_0, m', O \rangle$, where m_0 is the initial state of π, and m' is the memory of the controlled program after input I, and O is the output of the enforcement mechanism on I (i.e. $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(I) = O$). The initial state of the enforcement mechanism is $\langle \epsilon, \pi, m_0, m_0, \epsilon \rangle$, where m_0 is the initial memory of program π.

The semantics of the enforcement mechanism is described in Figure 5.2. The first rule says that, if the observation obtained by combining the recursive application of $\mathsf{EM}^{\bullet}_{\mathcal{P}}$ to $g(I.i)$ and the new observation that π is producing are allowed by the policy, then we just release the output of π. The second rule says that, if the obtained observation is not allowed, we will correct the execution. We correct it by selecting a new output

$$\text{OK}\ \frac{\langle \pi, m' \rangle \overset{i|o}{\rightarrow} \langle \pi, m'' \rangle \qquad M = \text{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I.i)) \qquad \text{reject}(M \cup \{(I.i, O.o)\}) = \mathbf{F}}{\langle I, \pi, m_0, m', O \rangle \overset{i|o}{\Rrightarrow} \langle I.i, \pi, m_0, m'', O.o \rangle}$$

$$\text{NOK}\ \frac{\langle \pi, m' \rangle \overset{i|o}{\rightarrow} \langle \pi, m'' \rangle \qquad \text{reject}(M \cup \{(I.i, O.o)\}) = \mathbf{T} \qquad \begin{array}{r} M = \text{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I.i)) \\ o' = \text{extend}_{\mathcal{P}}(M \cup \{(I, O)\}, I, i) \end{array}}{\langle I, \pi, m_0, m', O \rangle \overset{i|o'}{\Rrightarrow} \langle I.i, \pi, m_0, m'', O.o' \rangle}$$

Figure 5.2: Semantics of the enforcement mechanism $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$

using the consistent extension function $\text{extend}_{\mathcal{P}}$. It is worth noting that rule NOK presented here is a subcase of rule NOK in [37]. In [37], in rule NOK, the configuration of the controlled program after the transition can be an arbitrary configuration which is computed deterministically from state $\langle \pi, m'' \rangle$ and input item i.

We next show that the rules OK and NOK are a proper definition for a program.

Lemma 5.3.3. $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ *is total computable, and deterministic.*

Proof. We first prove that for every state $\langle I, \pi, m_0, m', O \rangle$ of the enforcement mechanism, and for every input i,

1. there exists m'', o such that $\langle I, \pi, m_0, m', O \rangle \overset{i|o}{\Rrightarrow} \langle I.i, \pi, m_0, m'', O.o \rangle$,
2. for any o, o', m'', m''', if $\langle I, \pi, m_0, m', O \rangle \overset{i|o}{\Rrightarrow} \langle I.i, \pi, m_0, m'', O.o \rangle$ and $\langle I, \pi, m_0, m', O \rangle \overset{i|o'}{\rightarrow} \langle I.i, \pi, m_0, m''', O.o' \rangle$ then $m' = m''$ and $o = o'$.

We show both properties by total induction on the well-founded order $\sqsubset$ on $\mathbf{I}^*$, the set of all finite inputs. So, suppose both properties (1) and (2) hold for all states $\langle I_0, \pi, m_0, m'_{\bullet}, O_0 \rangle$ and input i_0 with $I_0.i_0 \sqsubset I.i$.

The first property holds because (a) π is total computable (in any state, it accepts any input value and handles it in finite time), (b) the reject and extension functions are total computable, and (c) the computation of the map of $\mathsf{EM}^\bullet_{\mathcal{P}}(\pi)$ only needs a finite number of transitions on states $\langle I_0, \pi, m_0, m'_\bullet, O_0\rangle$ and inputs i_0 such that $I_0.i_0 \sqsubset I.i$ (this follows from the fact that g is well-founded and hence all $I' \in g(I.i) \sqsubset I.i$). Hence by the induction hypothesis all these transitions are total computable.

The second properties holds because (a) π is deterministic, (b) the computation of the map of $\mathsf{EM}^\bullet_{\mathcal{P}}(\pi)$ only needs transitions on states $\langle I_0, \pi, m_0, m'_\bullet, O_0\rangle$ and inputs i_0 such that $I_0.i_0 \sqsubset I.i$. Hence by the induction hypothesis all these transitions are deterministic. Now reject() deterministically returns either true or false. For the false case, we are done. For the true case, since the extension function is indeed a function and its parameters are deterministically determined by the input state and input value, this function deterministically returns an o'.

We have just shown that the two properties hold. These two properties imply that $\mathsf{EM}^\bullet_{\mathcal{P}}(\pi)$ is total computable and deterministic. □

Soundness. The definition of soundness is similar to the one in Chapter 4.

Definition 5.3.2. *The enforcement mechanism $\mathsf{EM}_{\mathcal{P}}$ of a policy $\mathcal{P}$ is sound iff for all programs π, $\mathsf{EM}_{\mathcal{P}}(\pi) \in \mathcal{P}$.*

Theorem 5.3.1 (Soundness)**.** *Let $\mathcal{P}$ be a testable and non-empty hypersafety policy. Then $\mathsf{EM}^\bullet_{\mathcal{P}}$ is sound.*

Proof. Let us say that a program π is I-level secure if it does not have any bad observation M_{bad} with for all $I' \in inputs(M_{bad})$, $I' \sqsubseteq I$.

We first show the following property: For all $I \in \mathbf{I}^*$, $\mathsf{EM}^\bullet_{\mathcal{P}}(\pi)$ is I-level secure. We prove this by complete induction on the well-founder order $\sqsubset$. So suppose the property holds for all inputs $I_1 \sqsubset I$. We prove it holds for I.

For the case where I is empty: Since the empty list is a minimal element under the $\sqsubset$ relation, we just have to show that the primitive observation (ϵ, ϵ) is not in $\mathcal{M}_{\mathcal{P}}$. This follows from the fact that $\mathcal{P}$ is non-empty: there is a program $\pi \in \mathcal{P}$, and every π has the observation (ϵ, ϵ).

For the case where I has the form $I_1.i$, we show that $\mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I_1.i) \cup \{I_1.i\}) \notin \mathcal{M}_{\mathcal{P}}$.

- For the subcase where the last output on this input was derived by the OK rule, this follows from the fact that $\mathrm{reject}(M \cup \{(I_1.i, O.o)\}) = \mathbf{F}$ for $M = \mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I_1.i))$.

- For the subcase where the last output on this input was derived by the NOK rule, we can use the induction hypothesis. All the inputs in $g(I_1.i) \cup \{I_1\}$ are $\sqsubset I_1.i$. Hence, the first argument to the extend function is an allowed observation. By the property of the extend function, we then also get for this subcase that $\mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I_1.i) \cup \{I_1.i\}) \notin \mathcal{M}_{\mathcal{P}}$.

Now we can show that $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ is I-level secure. Suppose there is an M_{bad} with all elements of $\mathrm{inputs}(M_{bad}) \sqsubseteq I$. Then we can easily see that for all $I' \in \mathit{inputs}(M_{bad})$ it holds that $\mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I) \cup \{I\}) \notin \mathcal{M}_{\mathcal{P}}$. (For $I' \sqsubset I$ this follows from the induction hypothesis and the fact that g is well-founded, for $I' = I$ we have just shown it.) But then the definition of test generator tells us that $M_{bad} \notin \mathcal{M}_{\mathcal{P}}$.

Finally, using this fact that $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ is I-level secure for all I, we can apply Lemma 5.3.1 and we get that $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi) \in \mathcal{P}$. $\square$

Precision. In Chapter 4, the definition of precision focuses on terminating executions. For reactive programs which run forever, we modify the notion of precision as below.

Definition 5.3.3. *An enforcement mechanism* $\mathsf{EM}_{\mathcal{P}}$ *is* precise *iff for any program* π *that satisfies* $\mathcal{P}$*, for any input* I*,* $\mathsf{EM}_{\mathcal{P}}(\pi)(I) = \pi(I)$*.*

Theorem 5.3.2 (Precision)**.** *Let* $\mathcal{P}$ *be a testable and non-empty hypersafety policy. Then* $\mathsf{EM}^{\bullet}_{\mathcal{P}}$ *is precise.*

Proof. We have to show that π and $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ have exactly the same primitive observations when $\pi \in \mathcal{P}$. We show this by complete induction on the well-founded partial order $\sqsubset$ on $\mathbf{I}^*$.

Assume that π and $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ have the same primitive observations (I', O') for all $I' \sqsubset I.i$. We have to show that $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(I.i) = \pi(I.i)$.

From the induction hypothesis, it follows that the derivation of the last step of $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ processing input $I.i$ was done by the OK rule: $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ is applied only on $I' \sqsubset I.i$, hence the induction hypothesis applies, and $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ has the same outputs as π on $g(I.i)$. Hence, $\mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I.i) \cup \{(I.i, O.o)\})$ is actually an observation on π, and since $\pi \in \mathcal{P}$ it follows that $\mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I.i)) \cup \{(I.i, O.o)\} \notin \mathcal{M}_{\mathcal{P}}$, and hence the call to reject must return false. As a consequence, the primitive observation of $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$ on $I.i$ is the same as the primitive observation of π on $I.i$. □

Theorem 5.3.3. *Every non-empty testable hypersafety policies can be enforced soundly and precisely.*

Proof. For any non-empty testable hypersafety policy:

- the reject predicate is total computable by definition of testable.
- Proposition 5.2.2 gives us a total computable extension function.
- Lemma 5.3.2 gives us a well-founded generator.

Hence, we can construct an enforcement mechanism with semantics as in Figure 5.2, and Theorems 5.3.1, and 5.3.2 tell us that this enforcement mechanism is sound and precise. □

5.4 Programming the general enforcement mechanism

In this section, we use the framework to program the general enforcement mechanism in Theorem 5.3.3. Notice that the general enforcement mechanism is constructed based on the extension function constructed in Proposition 5.2.2, and the well-founded generator g constructed in Lemma 5.3.2. The generator is based on an enumeration of finite inputs such that I is enumerated before $I.i$ for any I and i. The enumeration defines a total order between finite inputs. Let $\mathit{enum}(j)$ return the j-th input, where $1 \leq j$, and $\mathit{position}(I)$ return the position of I in the order defined by the enumeration procedure. We have $\text{position}(I) = j$ iff $\text{enum}(j) = I$.

The activation of MAP and REDUCE are described in Figure 5.3. The programs of MAP and REDUCE are described respectively in Figure 5.4 and Figure 5.5. Here we assume that a map instruction can send an input to a local execution instead of only an input item. This assumption can be implemented by using a loop on input items of the input. Initially, the enforcement mechanism has only one local execution $\pi[0]$ at a sleeping state with its initial memory

MAP is activated when REDUCE terminates, and there is only one local execution which is the local execution that does not receive any input item. Notice that MAP stores the input that it has received so far. This information is necessary for MAP to calculate all smaller inputs when it handles a new input item.

Assume that $\mathsf{EM}_{\mathcal{P}}(\pi)(I) = O$ and $\pi(I.i) = O_\pi.o$. Now the enforcement mechanism has to handle a new input item i. To know whether the output o is good or not, like $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)$, the enforcement mechanism needs to test whether $(I.i, O.o_\pi)$ and observation of the mechanism on $g(I.i)$ are good or not; and the enforcement mechanism calculates good outputs for inputs in the order from 1 to $\text{position}(I.i)$.

$$\text{MACT}_{\text{TESTABLE}}\ \frac{\mathsf{red.prg}{:}\mathbf{skip} \qquad \mathsf{top}{:}TOP = 0}{\Delta, \mathsf{map.prg}{:}\mathbf{skip} \Rightarrow \Delta, \mathsf{map.prg}{:}\pi_M}$$

$$\text{RACT}_{\text{TESTABLE}}\ \frac{\begin{array}{c}\forall j, 0 \le j \le TOP, EX[j].\mathsf{in}{:}I = \epsilon \\ \forall j, 0 \le j \le TOP, EX[j].\mathsf{int}{:}sig \ne \bot\end{array}}{\Delta, \mathsf{red.prg}{:}\mathbf{skip} \Rightarrow \Delta, \mathsf{red.prg}{:}\pi_R}$$

Figure 5.3: Activation of MAP and REDUCE

```
1: input $i$ from $env_{in}$
2: $j := 1$
3: while ($j \le \text{position}(I.i)$) do
4:     clone(identical(0))
5:     map(enum($j$), identical($j$))
6:     wake(identical($j$))
7:     $j := j + 1$
8: end while
9: $I := I.i$
```

Figure 5.4: MAP for a non-empty testable hypersafety policy

MAP goes through all inputs with order from 1 to position$(I.i)$. For each input, MAP creates a clone of the controlled program (Figure 5.4-line 4) and fetch this clone the input (Figure 5.4-line 5). After the iteration, MAP updates the input that it has received so far (Figure 5.4-line 9). Since MAP starts the iteration from 1, $\pi[0]$ does not receive any input item from MAP. Thus, a clone of this local execution can be used by the enforcement mechanism when it wants to know output of the controlled program on a specific input.

REDUCE is activated when all local executions consumes all local inputs and finish calculating outputs. It uses *outputs*, a function that maps integer numbers to outputs, to manage the generated output of the enforcement mechanism on an input. On activation, REDUCE calculates outputs of the

```
1:  outputs := $\lambda x.x \mapsto \epsilon$
2:  $j := 2$
3:  while ($j \leq TOP$) do
4:      $I'.i' :=$ enum($j$)
5:      //Get the output of $\pi[j]$ on $I'.i'$
6:      retrieve $O'_\pi.o'$ from j
7:      //Get the output of the enforcement mechanism on $I'$
8:      $O' :=$ outputs(position($I'$))
9:      //Get outputs of the enforcement mechanism on inputs in $g(I'.i')$ and output of $\pi$
    on $i'$.
10:     $M := \bigcup_{1 \leq k < j}\{\text{enum}(k), \text{outputs}(k)\} \cup \{(I'.i', O'.o')\}$
11:     //Test observation $M$
12:     if $reject(M)$ then
13:         //If the observation is bad, the output returned by the extend function is used
14:         outputs := outputs[$j \mapsto O'.\text{extend}_{\mathcal{P}}(M, I', i)$]
15:     else
16:         //If the observation is good, output $o'$ is used
17:         outputs := outputs[$j \mapsto O'.o'$]
18:     end if
19:     $j := j + 1$
20: end while
21: $O.o :=$ outputs($TOP$)
22: kill($\lambda x.(x > 0) \wedge (x \leq TOP)$)
23: output $o$ to $env_{out}$
```

Figure 5.5: REDUCE for a non-empty testable hypersafety policy

enforcement mechanism on inputs smaller than or equal to enum(TOP), where enum(TOP) is the input that the enforcement mechanism has received. Notice that REDUCE does not need to handle the case of enum(1) which is the empty input since $\mathsf{EM}_{\mathcal{P}}(\pi)(\epsilon) = \epsilon$. Therefore, the loop to calculate outputs (from Figure 5.5-line 3 to Figure 5.5-line 20) starts from 2.

At Figure 5.5-line 5, REDUCE gets input enum(j) which is not an empty input (since $j > 1$). At Figure 5.5-line 6, REDUCE collects output $O'_{\pi}.o'$ of the controlled program on $I'.i'$. At Figure 5.5-line 8, REDUCE collects the output of the enforcement mechanism on I' by outputs(position(I')). Notice that when $I'.i'$ is handled, the output of the enforcement mechanism on I' was calculated and can be retrieved by outputs(position(I')).

Next, at Figure 5.5-line 10, REDUCE constructs observation M that contains $(I'.i', O'.o')$ and all primitive observations of the enforcement mechanism on inputs smaller than $I'.i'$. After that, REDUCE tests whether o' is a good output for the enforcement mechanism on i' by using reject(M). If o' is a bad output (reject(M) returns true), the output item returned by extend$_{\mathcal{P}}(M, I, i)$ is used as a good output item for the enforcement mechanism on i' (Figure 5.5-line 14). Otherwise, o' is used (Figure 5.5-line 17).

In the last iteration of the loop, j is equal to TOP and enum(j) returns $I.i$. Following the above description, the output of the enforcement mechanism on $I.i$ is calculated and stored in outputs(TOP). Thus, after the loop, REDUCE gets the output item for the enforcement mechanism on i from outputs(TOP) (Figure 5.5-line 21) and sends this output item to the environment (Figure 5.5-line 23). Before sending the output, REDUCE removes all local executions except $\pi[0]$ from the array of local executions (Figure 5.5-line 22).

Lemma 5.4.1. *For any non-empty testable hypersafety policy $\mathcal{P}$, for any program π and any finite input I, $\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(I) = \mathsf{EM}_{\mathcal{P}}(\pi)(I)$, where $\mathsf{EM}^{\bullet}_{\mathcal{P}}$ is the enforcement mechanism in Theorem 5.3.3.*

Proof. Base case: It is easy to check that $\mathsf{EM}_{\mathcal{P}}(\pi)(\epsilon) = \mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(\epsilon) = \epsilon$ (ϵ is the minimal element in the order defined by g).

Assume that the lemma holds for all inputs smaller than $I.i$. We now look at $I.i$.

For REDUCE, the loop from 2 to $TOP - 1$ calculates the output of the enforcement mechanism on inputs smaller than $I.i$. From the assumption, we have that for all I' smaller than $I.i$, $\mathsf{EM}_{\mathcal{P}}(\pi)(I') = \mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(I')$. Thus, $\mathrm{map}(\mathsf{EM}_{\mathcal{P}}(\pi), g(I.i)) = \mathrm{map}(\mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi), g(I.i))$, where $\mathrm{map}(\mathsf{EM}_{\mathcal{P}}(\pi), g(I.i))$ is calculated by REDUCE in the last iteration. From the construction of $\mathsf{EM}_{\mathcal{P}}(\pi)$ and $\mathsf{EM}^{\bullet}_{\mathcal{P}}$, we have $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = \mathsf{EM}^{\bullet}_{\mathcal{P}}(\pi)(I.i)$. □

Theorem 5.4.1. *For any non-empty testable hypersafety policy $\mathcal{P}$, the constructed enforcement mechanism $\mathsf{EM}_{\mathcal{P}}$ is sound and precise.*

Proof. The proof of this theorem follows from the facts that on any program and input our constructed mechanism and the mechanism $\mathsf{EM}^{\bullet}_{\mathcal{P}}$ in Theorem 5.3.3 generate the same output (Lemma 5.4.1), and $\mathsf{EM}^{\bullet}_{\mathcal{P}}$ is sound and precise (Theorem 5.3.3). □

5.5 Instances of the general mechanism

Given a hypersafety policy $\mathcal{P}$, the steps that need to be taken to enforce the policy using our general enforcement mechanism presented in Section 5.4 are:

1. specify a total computable reject function that decides membership in $\mathcal{M}_{\mathcal{P}}$. Prove that the set of observations for which reject returns true is indeed maximal in the sense of Definition 5.2.2.

2. specify a total computable test generator function as in Lemma 5.3.2.

3. specify a total computable extension function, and prove that it has the property required of such a function as specified in definition 5.2.4. For a testable hypersafety policy, the construction of such an extension function can be found in Proposition 5.2.2.

Non-interference with two levels. Let *pr* be a total idempotent function from finite inputs to finite inputs. That is, for all $I \in \mathbf{I}^*$, $pr(I) = pr(pr(I))$. We think of *pr* as a projection that removes confidential information from the input.

A program π is *non-interferent w.r.t. pr* iff

$$\forall I, I' \in \mathbf{I}^* : pr(I) = pr(I') \implies O|_L = O'|_L,$$

where $\pi(I) = O$ and $\pi(I') = O'$.

The projection *pr* can be instantiated in many ways, and our enforcement mechanism can handle all these instantiations.

- $pr(I) = I|_{L,HD}$ where $I|_{L,HD}$ is the resulting input after replacing the high input values in I with default values: this is a variation of non-interference where *content* of input events is secret but the *occurrence* of the input event is not. Our general enforcement mechanism can be reduced to standard secure multi-execution [20, 38].
- $pr(I) = I|_L$: models standard non-interference as in Example 5.2.1. Our general enforcement mechanism defines a reactive variant of secure multi-execution as in [9, 51].
- *pr* can more generally project to values that depend on all previous values in the input. This can model for instance non-interference with stateful declassification policies [47]. Our general enforcement mechanism even improves on the mechanism in [47], as it does not require declassify annotations for precision since in [47] a declassify operator is just a directive indicating that a particular value is computed by the release function.

Construction of the reject predicate for this policy is similar to Example 5.2.3. As mentioned above, construction of test generator and extension function is described in Lemma 5.3.2 and Proposition 5.2.2.

$$\text{reject}(M) = \begin{cases} \mathbf{T} & \exists M_{bad} = \{(I,O),(I',O')\}, \\ & \quad M_{bad} \subseteq \overline{M} \text{ s.t. } pr(I) = pr(I')\text{and } O|_L \neq O'|_L, \\ \mathbf{F} & \text{otherwise.} \end{cases} \tag{5.1}$$

Non-interference for multiple levels. It is relatively straigtforward to extend all the variants of non-interference above to multiple confidentiality levels. We illustrate this for standard non-interference.

Let $\langle \mathcal{L}, \leq \rangle$ be a complete lattice of security levels with a top level ($\top$) and a bottom level ($\perp$), and let *lvl* be a function from $\mathbf{I} \cup \mathbf{O}$ to $\mathcal{L}$. A program π is *non-interferent* with respect to *lvl* iff

$$\forall I, I' \in \mathbf{I}^* : I|_l = I'|_l \implies O|_l = O'|_l$$

where $\pi(I) = O$ and $\pi(I') = O'$, and $I|_l$ filters out all i with $lvl(i) \not\leq l$ (and similarly for $O|_l$).

Construction of the reject predicate for this policy is as below. Construction of test generator and extension function is described in Lemma 5.3.2 and Proposition 5.2.2.

$$\text{reject}(M) = \begin{cases} \mathbf{T} & \exists M_{bad} = \{(I,O),(I',O')\}, \\ & \quad M_{bad} \subseteq \overline{M} \text{ s.t. } I|_l = I'|_l\text{and } O|_l \neq O'|_l, \text{ for some } l \\ \mathbf{F} & \text{otherwise.} \end{cases} \tag{5.2}$$

5.6 Summary

This chapter presented the investigation on sufficient condition of policies that can be enforced by using the proposed framework. Controlled programs are reactive programs which accepts any inputs, process inputs in finite time and have to finish processing an input item before handling another one.

This chapter described the notion of testable hypersafety policies, presented a general enforcement mechanism of testable hypersafety policies, and proved that the general enforcement mechanism is sound and precise. It then used the proposed framework to program the general enforcement mechanism.

Chapter 6

Downward Closed w.r.t. Termination Policies

Chapter 5 presented the investigation on which policies can be enforced by using the framework on reactive programs. One important constraint on reactive programs is that reactive programs must handle inputs in finite time. However, there are reactive programs that might diverge when handling inputs. Thus, this chapter investigates which policies can be enforced on such reactive programs. Specifically, the investigation focuses on an enforcement mechanism similar to the one in Chapter 5.

6.1 Overview

We consider reactive programs written in the template described in Figure 6.1. Compared to reactive programs in Chapter 5, on handling an input item, the calculation of output might not terminate. Thus, on input $I.i$, a program might consume I, generate O, and then diverge when calculating the output for i.

We use $\uparrow$, a special output value to model diverged output calculation. In a primitive observation, output values after $\uparrow$ are only $\uparrow$. If there is no

```
1: while T do
2:     input $x$ from $env_{in}$
3:     //Calculate output o with a deterministic program that might diverge.
4:     output $o$ to $env_{out}$
5: end while
```

Figure 6.1: Implementation of a reactive program that might diverge on an input item.

```
1: input $i$ from $env_{in}$
2: $j := 1$
3: while ($j \leq$ position($I.i$)) do
4:     clone(identical(0))
5:     map(enum($j$), identical($j$))
6:     wake(identical($j$))
7:     $j := j + 1$
8: end while
9: $I := I.i$
```

Figure 6.2: MAP for a non-empty, downward closed w.r.t. termination policy

$\uparrow$ in O, then (I, O) is a *terminating primitive observation.* Otherwise, it is a *diverging primitive observation.* Hereafter, by O we mean an output without $\uparrow$, and by o an output item that is different from $\uparrow$. We write $(I, O \uparrow^n)$ to denote a diverging primitive observation in which $\uparrow$ is repeated n times.

We investigate which policies can be enforced soundly and precisely by the enforcement mechanism with MAP and REDUCE programs are described respectively in Figure 6.2 and Figure 6.3. The MAP program here is similar to the one in Chapter 5. We modify the program of REDUCE a bit since now the extension function may return $\uparrow$. In this case, REDUCE goes into a forever loop. The modified program of REDUCE is in Figure 6.3, and the forever loop is at lines 16-18. Notice that the enforcement mechanism in Chapter 5 is constructed based on a total order between inputs such that for any input I and input item i, I is smaller than $I.i$.

```
1: outputs := $\lambda x.x \mapsto \epsilon$
2: $j := 2$
3: while ($j \leq TOP$) do
4:     $I'.i'$ := enum($j$)
5:     //Get the output of $\pi[j]$ on $I'.i'$
6:     retrieve $O'_\pi.o'$ from j
7:     //Get the output of the enforcement mechanism on $I'$
8:     $O'$ := outputs(position($I'$))
9:     //Get outputs of the enforcement mechanism on inputs in $g(I'.i')$ and output of $\pi$
   on $i'$.
10:    $M := \bigcup_{1 \leq k < j} \{\text{enum}(k), \text{outputs}(k)\} \cup \{(I'.i', O'.o')\}$
11:    //Test observation $M$.
12:    if reject($M$) then
13:        //If the observation is bad, the output returned by the extend function is used
14:        $o''$ := extend$_{\mathcal{P}}(M, I', i)$
15:        if $o'' = \uparrow$ then
16:            while T do
17:                skip
18:            end while
19:        else
20:            outputs := outputs[$j \mapsto O'.o''$]
21:        end if
22:    else
23:        //If the observation is good, output $o'$ is used
24:        outputs := outputs[$j \mapsto O'.o'$]
25:    end if
26:    $j := j + 1$
27: end while
28: $O.o$ := outputs($TOP$)
29: kill($\lambda x.(x > 0) \wedge (x \leq TOP)$)
30: output $o$ to $env_{out}$
```

Figure 6.3: REDUCE for a non-empty, downward closed w.r.t. termination policy

6.2 Orderly terminating policies

For a policy to be enforced precisely by this enforcement mechanism, we require that if the enforcement mechanism on I terminates, then the enforcement mechanism also terminates on inputs smaller than I. Thus, in a good observation M, if (I, O) is a terminating primitive observation in $\overline{M}$, then for any I' that is smaller than I and is in a primitive observation in $\overline{M}$, then I' is also in a primitive terminating observation in $\overline{M}$.

Let $M{\downarrow} \triangleq \{(I, O) \in M\}$ be the observation that contains all terminating primitive observations in M.

Definition 6.2.1 (Orderly Terminating). *A testable hypersafety policy $\mathcal{P}$ is orderly terminating iff there exists a total, well-founded order $\sqsubset$ such that*

- *for all I and i, $I \sqsubset I.i$, and*
- *for all $M \notin \mathcal{M}_{\mathcal{P}}$, all I' and I:*

$$I \in \mathit{input}(\overline{M}{\downarrow}) \;\wedge\; I' \in \mathit{input}(\overline{M}) \;\wedge\; I' \sqsubset I \implies I' \in \mathit{input}(\overline{M}{\downarrow})$$

Example 6.2.1. *An example of orderly terminating policies is the policy that requires a good program terminates on any input. Hereafter, we refer to this policy by TER. The maximal set of bad observations of the policy:*

$$\mathcal{M}_{\mathcal{P}} = \{M | \exists (I, O^n) \in \overline{M}\}.$$

For this policy, we construct a total order between inputs such that I smaller than $I.i$ for any I and i. It is easy to check that this order satisfies the first condition specified in Definition 6.2.1.

Assume that the order does not satisfy the last condition of orderly terminating policy. Thus, there exist an observation $M \notin \mathcal{M}_{\mathcal{P}}$, I and I' such that:

$$I \in input(\overline{M}\!\downarrow) \;\wedge\; I' \in input(\overline{M}) \;\wedge\; I' \sqsubset I \wedge I' \not\in input(\overline{M}\!\downarrow).$$

It follows that there exist an observation $M \not\in \mathcal{M}_{\mathcal{P}}$, I and I' such that:

$$I \in input(\overline{M}\!\downarrow) \;\wedge\; I' \in input(\overline{M}) \;\wedge\; I' \sqsubset I \wedge I' \in input(\overline{M}\!\uparrow),$$

where $\overline{M}\!\uparrow$ contains all diverting primitive observation in $\overline{M}$.

In other words, there exists a good observation containing a diverging primitive observation. From the specification of the maximal set of bad observations, this observation is a bad one. Contradiction.

6.3 Allowably divergent policies

Is the constructed enforcement mechanism for TER sound? Unfortunately, the answer is negative. When the controlled program diverges on an input, the whole enforcement mechanism also diverges on inputs larger than or equal to this input.

The enforcement mechanism is sound if the divergence of the enforcement mechanism because of the divergence of a local execution is always accepted by the enforced policy. In addition, from the construction of the enforcement mechanism, when the enforcement mechanism diverges on an input I', it also diverges on inputs larger than I'. Those divergences also have to be accepted by the enforced policy. Thus, for a policy to be enforced soundly, if a bad observation is bad because it contains a diverging primitive observation $(I', O' \uparrow^n)$, then this bad observation must contain a primitive observation with input I, where I' is smaller than I. In other words, a diverging primitive observation is not bad because of itself.

Definition 6.3.1 (Allowably Divergent)**.** *A testable hypersafety policy $\mathcal{P}$ is allowably divergent iff there exists a total, well-founded order $\sqsubset$ such that*

- *for all I and i, $I \sqsubset I.i$, and*
- *for all $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ and all I' in input($\overline{M}_{bad}$):*

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \implies \exists I : I' \sqsubset I \wedge I \in input(\overline{M}_{bad})$$

Example 6.3.1 (TINI for reactive programs)**.** *Reactive programs run forever. After finishing handling an input item, a controlled program can consume another one. Thus, in the below definition, instead of requiring $(\pi, I) \Downarrow O$ as in Definition 4.2.1, we only require that $\pi(I) = O$.*

A reactive program π satisfies termination-insensitive non-interference *iff*

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

The maximal set of bad observations of TINI:

$$\mathcal{M}_{\mathcal{P}} = \{M | \exists\{(I', O'), (I, O)\} \subseteq \overline{M} \text{ s.t. } I|_L = I'|_L \wedge O|_L \neq O'|_L\}$$

We construct a partial order $\sqsubset_{po}$ such that $I \sqsubset_{po} I.i$ for any I and i; and $I' \sqsubset_{po} I$ if $I' = I|_L$ and $I' \neq I$. We next prove that for all $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ and all I' in input($\overline{M}_{bad}$):

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \implies \exists I : I' \sqsubset^{po} I \wedge I \in input(\overline{M}_{bad}).$$

Assume that the constructed partial order does not satisfy the property. Thus, there must exist $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ and $I' \in input(\overline{M}_{bad})$ such that:

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \wedge \big(\forall I : I' \sqsubset^{po} I \implies I \notin input(\overline{M}_{bad})\big).$$

However, from the specification of the maximal set of bad observations, there is no such bad observation. Contradiction.

We next prove that the constructed partial order $\sqsubset^{po}$ can be extended to a total order $\sqsubset$ that satisfies the conditions of allowably divergent policy.

The first condition is trivial. Because $\sqsubset$ is an extension of $\sqsubset^{po}$, for any I and i if $I \sqsubset^{po} I.i$ then $I \sqsubset I.i$. We now look at the second condition.

Let $Q(\sqsubset_a)$ be a predicate on $\sqsubset_a$, where $\sqsubset_a$ is an arbitrary order defined on all finite inputs. If $Q(\sqsubset_a)$, then for all $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ and all I' in input($\overline{M}_{bad}$):

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \implies \exists I : I' \sqsubset_a I \wedge I \in input(\overline{M}_{bad}).$$

We need to prove that if $Q(\sqsubset^{po})$, and $\sqsubset$ is an extension of $\sqsubset^{po}$, then $Q(\sqsubset)$. Assume that this statement does not hold. Thus, $Q(\sqsubset^{po})$, and $\sqsubset$ is an extension of $\sqsubset^{po}$, and $\neg Q(\sqsubset)$. Since $\neg Q(\sqsubset)$, there must exist $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ and $I' \in$ input($\overline{M}_{bad}$) such that:

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \wedge \big(\forall I : I' \sqsubset I \implies I \notin input(\overline{M}_{bad})\big).$$

We consider an arbitrary bad observation $M_{bad} \notin \mathcal{M}_{\mathcal{P}}$ and $I' \in$ input($\overline{M}_{bad}$) such that $\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}}$. Since $Q(\sqsubset^{po})$, there exists I such that $I' \sqsubset^{po} I$ and $I \in$ input(M_{bad}). Because $\sqsubset$ is an extension of $\sqsubset^{po}$, $I' \sqsubset^{po} I$ implies $I' \sqsubset I$. Thus, for an arbitrary M_{bad} and I', if $\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}}$, then there exists an I such that $I' \sqsubset I$, and $I \in$ input($\overline{M}_{bad}$). Contradiction with $\neg Q(\sqsubset)$.

6.4 Downward closed w.r.t. termination policies

A policy that is both allowably divergent and orderly terminating might not be enforced soundly and precisely by our construction. We consider a policy that has two different orders, one satisfies the condition of orderly terminating policies and one satisfies the condition of allowably divergent policies. However, there is no order for this policy that satisfies both condition. This policy cannot be enforced soundly and precisely by our constructed enforcement mechanism.

We next define downward closed w.r.t. termination policies and prove that a non-empty downward closed w.r.t. termination policy can be enforced soundly and precisely by our constructed enforcement mechanism.

Definition 6.4.1 (Downward Closed w.r.t. Termination). *A testable hypersafety policy $\mathcal{P}$ is downward closed w.r.t. termination iff there exists a total, well-founded order $\sqsubset$ such that:*

- *for all I and i, $I \sqsubset I.i$, and*
- *for all $M \notin \mathcal{M}_\mathcal{P}$, all I' and I:*

$$I \in \mathit{input}(\overline{M}\!\downarrow) \;\wedge\; I' \in \mathit{input}(\overline{M}) \;\wedge\; I' \sqsubset I \implies I' \in \mathit{input}(\overline{M}\!\downarrow)$$

- *for all $M_{bad} \in \mathcal{M}_\mathcal{P}$ and all I' in $\mathit{input}(\overline{M}_{bad})$:*

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_\mathcal{P} \implies \exists I : I' \sqsubset I \wedge I \in \mathit{input}(\overline{M}_{bad})$$

Hereafter, when we mention an order of a downward closed w.r.t. termination policy, we mean the order that satisfies all conditions in Definition 6.4.1.

Example 6.4.1. *We consider a policy $\mathcal{P}$ defined on $\boldsymbol{I} = \{a, b\}$ and $\boldsymbol{O}$ is an enumerable set. The policy requires that good programs might terminates only on $[a]$ or $[b]$, and if good programs terminates on $[b]$, they also terminate on $[a]$.*

$$\begin{aligned}\mathcal{M}_\mathcal{P} = \big\{M| \;\; & \exists\{([a],[\uparrow]),([b],[o])\} \subseteq \overline{M} \text{ for some } o \quad \vee \\ & \exists\{(i.I, o.O)\} \subseteq \overline{M} \text{ for some } o \text{ and } O,\ i \text{ and non-empty } I\big\}\end{aligned}$$

We construct a total order between finite inputs such that $\epsilon \sqsubset a \sqsubset b$ and $I \sqsubset I.i$ for any $I.i$. It is easy to check that this order satisfies the first condition of downward closed w.r.t. termination policies. We next *prove that this order satisfies the other conditions of downward closed w.r.t. termination policies.*

Assume that the constructed order does not satisfy the second condition. Thus, there exist $M \notin \mathcal{M}_{\mathcal{P}}$, I', I such that:

$$I \in \mathit{input}(\overline{M}\downarrow) \wedge I' \in \mathit{input}(\overline{M}) \wedge I' \sqsubset I \wedge I' \notin \mathit{input}(\overline{M}\downarrow)$$

From the specification of the maximal set of bad observations, good programs might terminate only on $[a]$ and $[b]$. In addition, if they terminates on $[b]$, they must terminate on $[a]$. Thus, there is no good observation M such that $I \in \mathit{input}(\overline{M}\downarrow)$, $I' \in \mathit{input}(\overline{M})$, $I' \sqsubset I$ and $I' \notin \mathit{input}(\overline{M}\downarrow)$. Contradiction.

Assume that the constructed order does not satisfy the third condition. Thus, there exist $M_{bad} \in \mathcal{M}_{\mathcal{P}}$, $I' \in \mathit{input}(\overline{M}_{bad})$:

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \wedge \big(\forall I : I' \sqsubset I \implies I \notin \mathit{input}(\overline{M}_{bad})\big)$$

From the specification of the maximal set of bad observation, to satisfy the condition $\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}}$, M_{bad} must include $\{([a], [\uparrow]), ([b], [o])\}$ for some o, and does not include $\{(i.I, o.O)\}$ for some o and O, i and non-empty I. By removing $([a], [\uparrow])$ from such a bad observation, we get a good observation. However, there are inputs (e.g. $[b]$) larger than $[a]$ in the bad observation. Contradiction.

6.4.1 Enforcement mechanism

The MAP program of the enforcement mechanism of a non-empty downward closed w.r.t. termination policy is the same as the one described in Figure 5.4. The REDUCE program is described in Figure 6.3. The activation of MAP and REDUCE is the same as the one in the enforcement mechanisms of testable hypersafety policies. Notice that our enforcement mechanism is constructed with the generator based on an enumeration of inputs such that I is enumerated before $I.i$ for any I and i.

Lemma 6.4.1. *For any non-empty, downward closed w.r.t termination policy $\mathcal{P}$, for any π, any finite input I, map$(\mathsf{EM}_{\mathcal{P}}(\pi), g(I) \cup \{I\})$ is an allowed observation.*

Proof. We use the complete induction technique on order $\sqsubset$ defined in the set of all finite inputs.

For the base case, the lemma holds since $\mathsf{EM}_{\mathcal{P}}(\pi)(\epsilon) = \epsilon$ and $g(\epsilon) = \{\}$ (ϵ is the minimal element in the order $\sqsubset$).

Assume that the lemma holds for all inputs smaller than $I.i$. Let $M^{\bullet}$ be the observation that contains the observations of the enforcement mechanism on all inputs smaller than $I.i$, and $M = M^{\bullet} \cup \{(I.i, \mathsf{EM}_{\mathcal{P}}(\pi)(I.i))\}$.

Case 1: $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = O \uparrow^{n}$. This case happens when there is a local execution diverges, or there is an input on which the fix is $\uparrow$. Assume that M is a bad observation. From the induction hypothesis, $M^{\bullet} = M \setminus \{(I.i, O \uparrow^{n})\}$ is a good observation. From the definition of the policy, there must be an I^{b} in input$(\overline{M})$, where $I.i \sqsubset I^{b}$. From the construction, $I.i$ is the maximal input in M. Contradiction.

Case 2: $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = O.o$. From the property of the reject function and extension function, M is an allowed observation. □

Theorem 6.4.1. *For any non-empty, downward closed w.r.t. termination policy $\mathcal{P}$, the constructed enforcement mechanism is sound.*

Proof. Suppose that $\mathsf{EM}_{\mathcal{P}}(\pi)$ does not satisfy $\mathcal{P}$. Thus, there exists a bad observation M_{bad} exposed by $\mathsf{EM}_{\mathcal{P}}(\pi)$. From the definition of generators, there exists an I in M_{bad} such that map$(\mathsf{EM}_{\mathcal{P}}(\pi), g(I) \cup \{I\}) \in \mathcal{M}_{\mathcal{P}}$. From Lemma 6.4.1, for any I, map$(\mathsf{EM}_{\mathcal{P}}(\pi), g(I) \cup \{I\}) \notin \mathcal{M}_{\mathcal{P}}$. Contradiction. □

Theorem 6.4.2. *For any non-empty, downward closed w.r.t termination policy, the constructed enforcement mechanism is precise.*

Proof. For the base case, since $\pi(\epsilon) = \mathsf{EM}_{\mathcal{P}}(\pi)(\epsilon) = \epsilon$, the theorem holds for this case.

Assume that the theorem holds for all inputs smaller than $I.i$. We consider the following cases.

Case 1: $\pi(I) = O\uparrow^n$ for some n. Since I is smaller than $I.i$, the assumption applies. Thus, $\mathsf{EM}_{\mathcal{P}}(\pi)(I) = O\uparrow^n$. From the computation model, we have that $\pi(I.i) = O\uparrow^{n+1}$. From the construction of the mechanism, $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = O\uparrow^{n+1}$. Thus, the theorem holds for this case.

Case 2: $\pi(I) = O$. From the assumption, we also have $\mathsf{EM}_{\mathcal{P}}(\pi)(I) = O$.

a) $\pi(I.i) = O\uparrow$. Since there is a local execution that does not terminate, REDUCE is not activated and $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = O\uparrow$.

b) $\pi(I.i) = O.o$. From the property of the policy, for any input I' smaller than $I.i$, $\pi(I') = O'$. Thus, REDUCE is activated. Since there is no violation, o is used as the output of the enforcement mechanism. Thus, $\mathsf{EM}_{\mathcal{P}}(\pi)(I.i) = O.o$.

□

Remark 6.1. *As mentioned above, TER is not enforceable by our constructed enforcement mechanism. We showed that TER is an orderly terminating policy. We now give a formal proof that TER is not an allowably divergent policy. Assume that TER is an allowably divergent policy. Assume that TER is an allowably divergent policy. Let i and i' be two inputs. We consider a bad observation $M_{bad1} = \{(i, o_1), (i', \uparrow)\}$ for some o_1. It follows that $\overline{M}_{bad1} \setminus \{i', \uparrow\}$ is a good observation. Thus, there must exist an terminating primitive observation in $\overline{M}_{bad1}$ and the input of this primitive observation is larger than i'. Hence, $i' \sqsubset i$. We consider another bad observation $M_{bad2} = \{(i, \uparrow), (i', o'_2)\}$ for some o'_2. Using the similar reasoning as above, $i \sqsubset i'$. Contradiction.*

Reactive TINI is an allowably divergent policy and it is not enforceable by our general construction. We prove that reactive TINI is not an orderly terminating policy. Assume that it is an orderly terminating policy. Thus, there exists a total, well founded order $\sqsubset$ that satisfies the condition specified in Definition 6.2.1.

Let I and I' be inputs such that $I|_L = I'|_L$. We consider a good observation $M_1 = \{(I, O_1), (I', O'_1 \uparrow^n)\}$ for some O_1 and O'_1. From the specification of the order, $I \sqsubset I'$. We look at another good observation $M_2 = \{(I, O_2 \uparrow^m), (I', O'_2)\}$ for some O_2 and O'_2. It follows that $I' \sqsubset I$. Contradiction.

Remark 6.2. *There are policies that can be enforced by other enforcement mechanisms but cannot by our constructed one. In Example 6.4.2, we present such a policy.*

Example 6.4.2. *Given a program π^* such that the number of inputs on which π^* terminates is infinite and there is an input on which π^* diverges. Policy EQUI is defined as a set of programs that have the same outputs as π^* on all inputs (in other words, the set of programs that are equivalent to π^*):*

$$EQUI = \{\pi | \forall I : \pi(I) = \pi^*(I)\}$$

The maximal set of bad observations of the policy:

$$\mathcal{M}_{\mathcal{P}} = \{M | \exists (I, O) \in \overline{M} : \quad \pi^*(I) \neq O\}$$

This policy cannot be enforced by our constructed enforcement mechanism. The problem is that if $\pi^*(I.i) = O.o$ and $\pi(I.i) = O \uparrow$, our enforcement mechanism on π and $I.i$ diverges. However, this policy can be enforced soundly and precisely by a rewriting mechanism that maps any program to π^*.

This policy does not satisfy the specification of allowably divergent policy which requires that a diverging execution is not bad by itself. This policy is also not a orderly terminating policy.

First we prove that the policy is not an allowably divergent policy. Assume that this policy is a allowably divergent policy. Thus, there exists an order $\sqsubset$ *such that* $I \sqsubset I.i$ *for any* I *and* i*; and for all* $M_{bad} \in \mathcal{M}_{\mathcal{P}}$ *and all* I' *in* $input(\overline{M}_{bad})$*:*

$$\overline{M}_{bad} \setminus \{(I', O' \uparrow^n)\} \notin \mathcal{M}_{\mathcal{P}} \implies \exists I : I' \sqsubset I \land I \in input(\overline{M}_{bad})$$

We consider $I^\bullet.i^\bullet$ *such that* $\pi^*(I^\bullet.i^\bullet) = O^\bullet.o^\bullet$ *for some* $O^\bullet$ *and* $o^\bullet$ *(the existence of such input is guaranteed since there are inputs on which* π^* *terminates). A bad observation* $M_{bad} = \{I^\bullet.i^\bullet, O^\bullet \uparrow\}$*. By removing* $(I^\bullet.i^\bullet, O^\bullet \uparrow)$ *from* $\overline{M}_{bad}$*, we get a good observation. In this good observation, all inputs are prefixes of* $I^\bullet.i^\bullet$ *and smaller than* $I^\bullet.i^\bullet$*. Thus, there is no input that is larger than* $I^\bullet.i^\bullet$ *in* M_{bad}*. Contradiction.*

Now we prove that that the policy is not an orderly terminating policy. We first recall the definition of orderly terminating policies: there exists a well-founded, total order $\sqsubset$ *such that for any* I *and* i*,* $I \sqsubset I.i$*, and for all* $M \notin \mathcal{M}_{\mathcal{P}}$*, all* I' *and* I*:*

$$I \in input(\overline{M}{\downarrow}) \ \land \ I' \in input(\overline{M}) \ \land \ I' \sqsubset I \implies I' \in input(\overline{M}{\downarrow})$$

Thus, for all $M \notin \mathcal{M}_{\mathcal{P}}$*, all* I' *and* I*:*

$$\neg(I \in input(\overline{M}{\downarrow})) \ \lor \ \neg(I' \in input(\overline{M})) \ \lor \ \neg(I' \sqsubset I) \ \lor \ (I' \in input(\overline{M}{\downarrow}))$$

It follows that:

$$(I \in input(\overline{M}{\downarrow})) \ \land \ (I' \in input(\overline{M})) \ \land \ \neg(I' \in input(\overline{M}{\downarrow})) \implies \neg(I' \sqsubset I)$$

From $(I' \in input(\overline{M}))$ *and* $\neg(I' \in input(\overline{M}\downarrow))$*, it follows that* $I' \in input(\overline{M}{\uparrow})$*, where* $\overline{M}{\uparrow}$ *contains all diverging primitive observation in* $\overline{M}$*. Since the order is total,* $\neg(I' \sqsubset I)$ *implies* $I \sqsubset I'$*.*

Therefore, for all $M \notin \mathcal{M}_{\mathcal{P}}$, all I' and I:

$$(I \in input(\overline{M}\downarrow)) \wedge (I' \in input(\overline{M})) \wedge (I' \in input(\overline{M}\uparrow)) \implies (I \sqsubset I')$$

Given an input I' on which π^* diverges. Since the number of inputs on which π^* converges is infinite, it follows that there is an infinite number of inputs that are smaller than I'. Thus, the order is not well-founded. Contradiction.

6.5 Summary

This chapter presented the investigation on which policies can be enforced by an enforcement mechanism similar to the one in Chapter 5. Controlled programs are reactive programs that are input total and have to finish handling an input item before consuming another one. Different from reactive programs in Chapter 5, reactive programs in this chapter might diverge on handling inputs. This chapter proposed downward closed w.r.t. termination policies and proved that any non-empty downward closed w.r.t. termination policy can be enforced by the constructed enforcement mechanism. As shown in Example 6.4.2, there are enforceable policies that cannot be enforced by the constructed enforcement mechanism. Thus, expanding the sufficient condition for policies to be enforced is an interesting avenue for future work.

Chapter 7

Conclusion

This thesis proposed the MAP-REDUCE framework, a *programmable* framework, which can be used to construct enforcement mechanisms of different security policies. The framework is constructed based on the secure multi-execution technique. An enforcement mechanism from this framework can execute multiple instances of the controlled program, and handle input/output events for these executions.

To construct an enforcement mechanisms of a security policy, users have to write a MAP program and a REDUCE program to control executions of instances of the controlled program, inputs consumed and outputs generated by the enforcement mechanism. The thesis illustrated the framework by presenting enforcement mechanisms for selected information flow policies: non-interference, non-deducibility, and deletion of inputs.

The thesis also presented the investigation on which policies can be enforced soundly and precisely by the framework on reactive programs that accept any input and have to finish processing an input item before handling another one. On reactive programs that always terminate on handling inputs, it showed that any non-empty testable hypersafety policy can be enforced. On reactive programs that might diverge on inputs, it demonstrated that any non-empty downward closed w.r.t. termination policies can be enforced.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[2] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, 2009.

[3] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 2010 Workshop on Programming Languages and Analysis for Security*, PLAS '10, page 3, 2010.

[4] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, 2012.

[5] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 186–202, 2012.

[6] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1):3:1–3:26, June 2013.

[7] David A. Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zalinescu. Enforceable security policies revisited. In *Proceedings of the 1st Principles of Security and Trust Conference*, POST '12, pages 309–328, 2012.

[8] Frederic Besson, Nataliia Bielova, and Thomas Jensen. Hybrid information flow monitoring against web tracking. In *Proceedings of the 26th Computer Security Foundations Symposium*, CSF'13, pages 240–254, June 2013.

[9] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security*, NSS '11, pages 97–104, 2011.

[10] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? In *Proceedings of the 2008 Workshop on Formal Aspects in Security and Trust*, pages 287–301, 2008.

[11] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? - edited automata revisited. *International Journal of Information Security*, 10(4):239–254, 2011.

[12] Nataliia Bielova and Fabio Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*, 20(1):51–79, 2012.

[13] Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *Proceedings of the 2010 USENIX*

Conference on Web Application Development, WebApps'10, pages 11–11, 2010.

[14] Iulia Bolosteanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the 5th International Conference on Principles of Security and Trust*, POST ' 16, pages 24–45, 2016.

[15] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 322–331, 2008.

[16] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 2008 IEEE 21st Computer Security Foundations Symposium*, CSF '08, pages 51–65, June 2008.

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

[18] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 748–759, 2012.

[19] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4):469–509, July 2014.

[20] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, 2010.

[21] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, May 2004.

[22] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[23] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, January 2006.

[24] Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *Proceedings of the 2011 International Andrei Ershov Memorial Conference*, pages 170–178, 2011.

[25] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 413–428, 2011.

[26] Tejas Khatiwala, Raj Swaminathan, and V. N. Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *Proceedings of the 2006 Annual Computer Security Applications Conference*, ACSAC '06, pages 223–234, 2006.

[27] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[28] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[29] Gurvan Le Guernic. Precise Dynamic Verification of Confidentiality. pages 82–96.

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

[31] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science*, ASIAN'06, pages 75–89, 2007.

[32] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[33] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[34] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 185–, 2000.

[35] Jonathan K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, SP '87, pages 60–66, 1987.

[36] Minh Ngo and Fabio Massacci. Programmable enforcement framework of information flow policies. In *Proceedings of the 15th Italian Conference on Theoretical Computer Science*, ICTCS '14, pages 197–221, 2014.

[37] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. Runtime enforcement of security policies on black box reactive pro-

grams. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 43–54, 2015.

[38] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 33–48, 2013.

[39] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 92–106, Washington, DC, USA, 2009. IEEE Computer Society.

[40] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 86–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, September 2006.

[42] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. of Computer Security*, 17(5):517–548, October 2009.

[43] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[44] Daniel Schoepe, Musard Balliu, Benjamin Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, EuroS&P'16, 2016.

[45] David Sutherland. A model of information. In *Proceedings of the 1986 National Computer Security Conference*, pages 175–183, 1986.

[46] Chamseddine Talhi, Nadia Tawbi, and Mourad Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2-4):158–184, February 2008.

[47] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, pages 293–307, 2014.

[48] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.

[49] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[50] Dante Zanarini and Mauro Jaskelioff. Monitoring reactive systems with dynamic channels. In *Proceedings of the 9th Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 66:66–66:78, 2014.

[51] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. Precise enforcement of confidentiality for reactive systems. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 18–32, 2013.

[52] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 2001 IEEE Workshop on Computer Security Foundations*, CSFW '01, page 15, 2001.