Precise Enforcement of Confidentiality for Reactive Systems (extended version)

Dante Zanarini Mauro Jaskelioff Alejandro Russo

May 22, 2013

Abstract

In the past years, researchers have been focusing on applying information flow security to web applications. These mechanisms should raise a minimum of false alarms in order to be applicable to the millions of existing web pages. A promising technique to achieve this is secure multi-execution (SME). If a program is already secure, its secure multi-execution produces the same output events; otherwise, this correspondence is intentionally broken in order to preserve security. Thus, there is no way to know if unexpected results are due to bugs or due to semantics changes produced by SME. Moreover, SME provides no guarantees on the relative ordering of output events from different security levels. We argue that these shortcomings limit the applicability of SME.

In this article, we propose a scheduler for secure multi-execution which makes it possible to preserve the order of output events. Using this scheduler, we introduce a novel combination between monitoring and SME, called *multi-execution monitor*, which raises alarms only for actions breaking the non-interference notion of ID-security for reactive systems. Moreover, we show that the monitor guarantees transparency even for CP-similarity, a progress-sensitive notion of observation.

1 Introduction

In recent years there has been an increasing interest in applying information flow control as a general mechanism to preserve confidentiality on web applications (e.g. [11, 19, 22, 27, 46]). The adoption of this technology promises to reduce the need for ad-hoc and purpose-specific counter-measures (e.g. architectures to contain advertisement scripts [26], browser extension to control cache-based leaks [20], etc.) In fact, several of the OWASP top-ten web vulnerabilities [49] can be rephrased in terms of information-flow problems.

In a web scenario, where millions of web pages have been written and deployed, it is important to provide permissive information-flow mechanisms, i.e. mechanisms that raise as few false alarms as possible. Traditional Denning-style [16, 35, 48] information-flow enforcements perform over-approximations that could lead to reject secure programs. Driven by permissiveness and dynamic features of web scripting languages, researchers tend to adopt dynamic techniques in the form of execution monitors (e.g. [2, 4, 5, 37]). Despite efforts to push the limits of dynamic information-flow, execution monitors are still not capable of enforcing sound and precise information-flow policies [29, 39], and must therefore reject possibly secure and useful web pages.

Recently, Devriese and Piessens [17] devised an alternative dynamic approach, called secure multi-execution (SME), based on the idea of executing the same program several times, once for each security level. As opposed to previous enforcement mechanisms, this novel technique works with a black-box approach; it only requires applying specific actions when inputs and outputs (I/O) are produced. Secure multi-execution does not require either static analysis or execution monitoring. As claimed by their authors, secure multi-execution enforces a specific version of non-interference with significantly better precision than traditional static and/or other dynamic techniques. More precisely, they prove that termination-sensitive non-interferent programs, which terminate under normal execution, match the behaviour produced under secure multi-execution. In contrast, if the program is leaking information, secure multi-execution will change its semantics in order to enforce security. It is claimed that this approach is the first one to achieve both soundness and precision.

Although promising, secure multi-execution suffers from some drawbacks which may limit the applicability of this technique. More concretely, we identify the following weaknesses.

- *Precision* As described above, it is postulated that secure-multi execution is *precise*, and thus more permissive than static analysis and execution monitoring [7, 9, 17, 24]. We argue that such comparison might be somehow unfair. While static analysis and execution monitors are capable of accepting or rejecting programs, secure multi-execution just runs them. Trading permissiveness by uncertainty, secure multi-execution makes it not possible to distinguish when the semantics of a program has been altered in order to preserve security. As a result, users might experience that their programs do not behave as expected without knowing if it is due to software errors or due to security reasons. We believe that a better terminology to refer to the precision of secure multi-execution is *transparency* [25], i.e. the ability of an enforcement mechanism to preserve the semantics of executions which already obey the security policy in question.
- *Order of events* Secure multi-execution is claimed to be transparent for terminating runs of termination-sensitive non-interferent programs. Under scrutiny, this claim only holds if the interleaving of events from different security levels is not relevant for the computation. In fact, output events might be arbitrarily interleaved when coming from different security levels. Not preserving the order of events might be problematic, for instance, in web pages with complex DOM-elements reacting to the same event (e.g. mouse click.)[1]

[1] The W3C consortium specifies the expected behaviour for such scenarios. `http://www.`

- *Schedulers* In [7, 17], the soundness and transparency arguments are given for a particular scheduler, called $\texttt{select}_{\text{lowprio}}$. This scheduler prioritises the execution of the copy of the program associated to the lowest security level. Due to this choice, secure multi-execution rules out leaks through the *external timing covert channel*, i.e. revealing confidential data by precisely measuring the time in which external (and observable) events are triggered. However, one major drawback of this scheduler is that it requires a total extension of the lattice order. In web scenarios, however, web domains are often modelled as incomparable security levels [27], which prohibits the use of such scheduler. Authors in [24] discuss alternative scheduling strategies for arbitrary lattices which guarantee different security policies for different security levels.

The main contribution of this paper is *multi-execution monitoring*, a novel combination of monitoring and secure multi-execution. This technique respects the interleaving of events from different security levels, and thus provides better transparency results than secure multi-execution. More importantly, multi-execution monitoring allows to detect insecure commands with precision, i.e. the monitor only raises an alarm when a command breaks the non-interference notion of ID-security [11] for reactive systems.

Intuitively, the idea of multi-execution monitoring is simply to monitor a program by comparing it with its secure multi-execution. A multi-execution monitor runs a program simultaneously with its secure multi-execution version. The two programs will be in sync (perform exactly the same I/O operations) for as long as the execution is secure. If one version tries to do something different from the other, then the monitor reports that the program is insecure.

The most important contributions of this paper can be summarised as follows.

- Inspired by the coalgebraic theory of systems [21], we propose a novel semantics for programs based on *interaction trees*. This formulation treats programs as black-boxes, about which nothing is known except what is inferred from their I/O interactions with the environment. In this manner, we gain modularity since new programming features related to internal operations do not affect our formal results.
- We define a scheduler that significantly improves the transparency guarantees for secure multi-execution. With this scheduler, we can guarantee that secure multi-execution preserves the order of events and progress of secure programs.
- We introduce a multi-execution monitor which can precisely detect when commands violate ID-security. This feature, not only allows us to notify users when programs are malicious, but also enables the debugging of insecure programs. Moreover, multi-execution monitoring gives good transparency guarantees for the non-interference notions of ID- and CP-security [11]. In fact, these transparency guarantees make it possible to

`w3.org/TR/DOM-Level-2-Events/events.html` report leaks as traces of the original program.

The paper is organised as follows. In Section 2 we introduce reactive interaction trees, our model of reactive systems, show how JavaScript-like programs might be interpreted on it, and define the notion of ID-secure programs. Secure multi-execution is presented in Section 3. Section 4 presents a scheduler for secure multi-execution which preserves the order of events and provides better transparency guarantees. In Section 5, we define the security condition on executions that the multi-execution monitor will enforce. Multi-execution monitoring is introduced in Section 6. In Section 7 we discuss related work. Finally, in Section 8 we conclude and discuss future work.

2 Reactive Systems and Non-Interference

We model reactive programs as *interaction trees*, i.e. data structures in the form of trees describing every possible interaction with the environment. We assume a set of channels *Chan*, input values in a set I, and output values in a set O. An *event* is a piece of data paired with the name of the communication channel associated to it. Let $E_A = Ch \times A$ denote the set of events of type A. Interaction trees are then defined as the following coinductive datatype.

Definition 1 (Reactive Interaction Trees)**.**

$$\begin{aligned} \mathit{React} = {} & \mathit{Read}\ (E_I \to \mathit{React}) \\ & \mid \mathit{Write}\ (E_O \times \mathit{React}) \\ & \mid \mathit{Step}\ \mathit{React} \end{aligned}$$

Intuitively, constructor *Read* denotes a program that receives an input from a channel determined by the environment (E_I) and, based on that, decides how to continue (*React*). Constructor *Write* represents programs which write an output in a chosen channel (E_O) and continue with another computation (*React*). Finally, constructor *Step* corresponds to a silent step, that is, a computation which does not affect the environment. Silent steps allows us to model divergence. We do not model termination (*React* trees are necessarily infinite) as reactive systems are usually meant to run forever. However, we could easily model termination by adding a new constructor *Stop*.

Interaction trees have no notion of state, and therefore are more abstract than concrete labelled transition systems and make the semantics and proof machinery simpler.

The idea of modelling the interactions of reactive programs through an infinite tree comes from a coalgebraic view of systems [21]. An interaction tree is the carrier of the final coalgebra of a functor, implementations of concrete systems are given by coalgebras for this functor, and the interpretation of a program into *React* arises from the universal property of the final coalgebra.

$$(R_1)\frac{}{(Read(f), []) \Rightarrow []} \qquad (R_2)\frac{(f(e), i) \Rightarrow o}{(Read(f), e :: i) \bullet \Rightarrow o}$$

$$(W)\frac{(t, i) \Rightarrow o}{(Write\ (e, t), i) \Rightarrow e :: o} \qquad (S)\frac{(t, i) \Rightarrow o}{(Step(t), i) \Rightarrow \bullet :: o}$$

Figure 1: Evaluation relation for interaction trees

2.1 Semantics

An interaction tree is essentially a static description of the possible inputs and outputs that might occur during execution. To know exactly which interactions occur for a given run, we need to provide an evaluation relation. We start by defining possibly infinite sequences.[2]

Definition 2 (Colists)**.** Let A be a set. Consider the type of possibly infinite sequences of A to be coinductively defined as follows.

$$Colist_A = \ [] \mid A :: Colist_A$$

We write $[a, b, c]$ to denote a finite colist $a :: b :: c :: []$.

The structure of input and output events of the system is given by colists.

Definition 3 (Colists of input and output events)**.** We define the set of colists of inputs $\mathcal{I}$ and the set of colists of outputs $\mathcal{O}$ for reactive systems as follows.

$$\mathcal{I} = Colist_{E_I} \qquad\qquad \mathcal{O} = Colist_{E_o \cup \{\bullet\}}$$

The elements of an output colist are either an event E_O, or an invisible output $\bullet$.

The evaluation relation $\Rightarrow \subseteq (React \times \mathcal{I}) \times \mathcal{O}$ is coinductively defined by the rules in Figure 2.1, where we write $(t, i) \Rightarrow o$ for $(t, i, o) \in \Rightarrow$. Intuitively, feeding an interaction tree t with a colist of input events $i \in \mathcal{I}$ yields the output colist $o \in \mathcal{O}$ iff $(t, i) \Rightarrow o$.

Rule (R_1) produces no outputs ($[]$) when no input events are present. Rule (R_2) consumes the first available input event (e), and based on that, produces the output o ($(f(e), i) \Rightarrow o$). Rule (W) outputs an event e ($e :: o$), and then the outputs triggered by program t ($(t, i) \Rightarrow o$). Rule (S) simply outputs $\bullet$ when a silent computation step is performed.

Interaction trees are not concerned with the features of the programming language used to code a reactive system. This level of abstraction allows us to

[2] We distinguish lists (finite sequences), colists (possibly infinite sequences), and streams (infinite sequences). However, we overload the notation for constructors.

apply our technique and results to, for instance, different imperative or functional languages. For each language, it is enough to describe how interaction trees are generated from programs. To illustrate this point, we briefly describe interaction trees for reactive JavaScript-like programs.

2.2 Interaction trees for a JavaScript-like language

$$
\begin{aligned}
p &::= \cdot \mid h; p \\
h &::= ch(x)\,\{c\} \\
c &::= \ \texttt{skip} \\
&\quad \mid c\,\texttt{;}\,c \\
&\quad \mid x \;\texttt{:=}\; e \\
&\quad \mid \texttt{if}\ e\ \{c\}\,\{c\} \\
&\quad \mid \texttt{while}\ e\ \texttt{do}\ c \\
&\quad \mid \texttt{out(}ch\texttt{,}e\texttt{)} \\
&\quad \mid \texttt{new}\ h \\
&\quad \mid \texttt{eval}\ (e)
\end{aligned}
$$

Figure 2: A JavaScript-like language. Symbols ch and e range over channels and expressions, respectively.

Despite its simplicity, *React* is able to model the interactions of complex languages. As an example of that, Figure 2 presents a language inspired by [32]. This language is a subset of JavaScript and describes many of its features. Expressions are side-effect free and denote strings, numbers, and boolean values. Event handlers can change the state of the system as well as define new ones. Input and output channels are disjoint since we focus on how programs react with the environment rather than themselves. Programs (p) are defined as a sequence of event handlers. Event handlers (h) indicate which commands (c) to execute when an input arrives to a channel ($ch(x)\,\{c\}$). Most of the commands are self-explanatory. However, some of them require further explanation. Command `out(`ch`,`e`)` outputs the value denoted by e into channel ch. Command `new` h declares a new event handler (or replaces an existing one). Command `eval` (e) dynamically evaluates the instructions denoted by a string expression e.

Throughout the examples of this article, we will make the following assumptions: events have integer values; there are two input channels `L?`, `H?` and two output channels `L!`, `H!`; events received on `L?` and events sent to `L!` are considered public events; events received on `H?` and events sent to `H!` are considered private or secret events. Programs in this language may be seen as a loop which reads an event and executes the handler associated to it. Such a handler may produce some outputs, end silently or diverge. We can interpret every JavaScript-like program using interaction trees. More specifically, there exists a function $[\![-]\!] : \mathit{Prg} \to M \to \mathit{React}$, that given a program $p \in \mathit{Prg}$ and a memory $\mu \in M$, it gives us the resulting interaction tree $[\![p]\!](\mu)$ (see Appendix A for details). This tree denotes the interactions that may happen when program p is run under memory μ. Here, M is the set of memories, i.e. mappings of variables to values.

The programs of this language have a special structure: no input event may be handled *inside* a handler. However, this structure plays no role once we abstract away programs by interpreting them into interaction trees. The same observation can be made about many features of this language, such as assignments, conditionals, loops, and dynamic code evaluation, as well as features not present in it, e.g., objects and DOM-trees.

Example 4. Consider the following JavaScript-like program.

```
p =  H?(x){ r := x };

     L?(x){ if r = 0
               {out(L!,0)}
               { while 1 do skip };
```

The program diverges when the secret value stored in `r` is different from zero. The interpretation of p for a memory μ is (isomorphic to) the following interaction tree.

$$
\begin{aligned}
[\![p]\!](\mu) = \mathit{Read}\ (\lambda(ch, v).\ &\mathbf{case}\ ch\ \mathbf{of} \\
&\mathtt{H?} \rightarrow \mathit{Step}\ ([\![p]\!](\mu[r \mapsto v])) \\
&\mathtt{L?} \rightarrow \mathit{Step}\ (\mathbf{if}\ \mu(r) = 0 \\
&\qquad\qquad \mathbf{then}\ \mathit{Write}\ ((\mathtt{L!}, 0), [\![p]\!](\mu)) \\
&\qquad\qquad \mathbf{else}\ \mathit{diverge}) \\
&\_ \ \ \rightarrow \mathit{Step}\ ([\![p]\!](\mu)))
\end{aligned}
$$

We write $f[x \mapsto y]$ for the function that behaves like f, except on x where it maps to y. We denote with *diverge* the infinite sequence of *Step*s (i.e. $\mathit{diverge} = \mathit{Step}\ \mathit{diverge}$.)

2.3 Reactive Non-Interference

We organise security levels in a lattice $(\mathcal{L}, \sqsubseteq)$, with the intention to express that data at level ℓ_1 can securely flow into data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$, and we associate a security level to each channel. The security level of an event e (noted $lvl(e)$) is determined by the security level of its channel. We define the predicate $\mathit{visible}_\ell$ on events that determines when an event is observable for an observer at level ℓ.

$$\frac{lvl(e) \sqsubseteq \ell}{\mathit{visible}_\ell(e)}$$

Output colists may also have a silent event $\bullet$. Hence, for output colists we extend the predicate so that silent events are not visible at any security level $(\forall \ell.\, \neg \mathit{visible}_\ell(\bullet))$.

Definition 5 (ID-similarity)**.** ID-similarity between colists for an observer at level ℓ is formalised coinductively by the following rules.

$$\frac{}{[\,] \sim_\ell^{ID} [\,]} \qquad \frac{\neg \mathit{visible}_\ell(e) \quad s \sim_\ell^{ID} s'}{e :: s \sim_\ell^{ID} s'} \qquad \frac{\neg \mathit{visible}_\ell(e) \quad s \sim_\ell^{ID} s'}{s \sim_\ell^{ID} e :: s'}$$

$$\frac{\mathit{visible}_\ell(e) \quad s \sim_\ell^{ID} s'}{e :: s \sim_\ell^{ID} e :: s'}$$

ID-similarity is reflexive and symmetric, but it is not transitive. Note that two colists will be considered similar while there is hope that they may produce the same observable events. In particular, a colist s will be similar to every other colist if it consist of nothing but infinitely many events not visible at level ℓ. Hence, transitivity would imply that every pair of colists is similar, but this is clearly not the case.

We define the non-interference notion of ID-security for reactive programs [9, 11] as follows.

Definition 6 (ID-security)**.** An interaction tree t is ID-secure iff, for every level ℓ and input colists i, i' such that $(t, i) \Rightarrow o$ and $(t, i') \Rightarrow o'$, $i \sim_\ell^{ID} i'$ implies $o \sim_\ell^{ID} o'$.

The following definitions will be used exclusively in the proofs of the next sections.

Level by level ID-security We define ID-similarity of colists on a level-by-level basis.

Definition TR 1. The following relation coinductively defines ID-similarity for events at security level ℓ.

$$LS_1 \frac{}{[] \simeq_\ell^{ID} []} \qquad LS_2 \frac{lvl(e) = \ell \quad s \simeq_\ell^{ID} s'}{e :: s \simeq_\ell^{ID} e :: s'}$$

$$LS_3 \frac{lvl(e) \neq \ell \quad s \simeq_\ell^{ID} s'}{e :: s \simeq_\ell^{ID} s'} \qquad LS_4 \frac{lvl(e) \neq \ell \quad s \simeq_\ell^{ID} s'}{s \simeq_\ell^{ID} e :: s'}$$

To deal with output colists, we assume that for every level $\ell \in \mathcal{L}$, $lvl(\bullet) \neq \ell$.

Two colists are ID-level-by-level similar when the colists of events at level ℓ are similar (for any ℓ). More formally, we have the following definition.

Definition TR 2 (ID-Level-by-level similarity)**.** Given two colists s and s', they are ID-level-by-level similar, written $s \simeq^{ID} s'$, if and only if $\forall \ell.\ s \simeq_\ell^{ID} s'$.

Similarity at one security level ($\simeq_\ell^{ID}$) is weaker than similarity to an observer of level ℓ ($\sim_\ell^{ID}$):

$$s \sim_\ell^{ID} s' \quad \Rightarrow \quad s \simeq_\ell^{ID} s'$$

Similarity to an observer at level ℓ ($\sim_\ell^{ID}$) tell us that the two colists are similar if they coincide in every event e such that $lvl(e) \sqsubseteq \ell$. In contrast, $\simeq_\ell^{ID}$ is more modest: it only demands colists to coincide on events at level ℓ.

3 Secure Multi-Execution

Secure multi-execution runs a program multiple times, once for each security level, but giving I/O operations a level-dependent semantics. Outputs to a channel at security level ℓ are only performed in the execution corresponding to

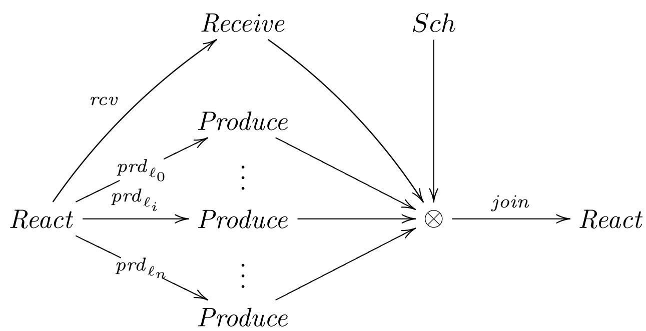

Figure 3: Transformation providing secure-multi execution

level ℓ. Inputs from a channel at security level ℓ are only available to executions corresponding to a level ℓ_e, where $\ell \sqsubseteq \ell_e$. It is clear that the secure multi-execution of programs does not leak information. The execution at level ℓ produces outputs only at that level, while consuming data with less or equal confidentiality than ℓ.

We will define secure multi-execution directly on interaction trees as a transformation that takes an interaction tree and returns another one representing its secure multi-execution. The returned interaction tree is guaranteed to be non-interferent, i.e. it does not leak secrets (see Definition 6). The transformation proceeds in two steps:

- The original interaction tree *React* gives rise to a receiver interaction tree *Receive* and to one producer interaction tree *Produce* for each security level. These new types of interaction trees are generated by functions *rcv* and prd_ℓ, respectively (explained below). Following the modelling of reactive systems [9–11], the interaction tree *Receive* captures actions related to obtaining input data from the environment, while *Produce* captures actions related to producing outputs.
- The *Receive* and *Produce* interaction trees are re-interpreted back into an interaction tree *React* with the help of a scheduler *Sch*. This is done by the function *join*.

Figure 3 illustrates the transformation process.

3.1 Receiver and Producers

Interaction trees *Receive* are coinductively defined as follows.

$$\begin{aligned} \mathit{Receive} = \ & \mathit{RnQ}\ (E_I \to \mathit{Receive}) \\ \mid\ & \mathit{Step}\ \mathit{Receive} \end{aligned}$$

The transformation from the original interaction tree to a receiver simply transforms writes into silent steps, and reads into “read and queue” (constructor

RnQ).

$$\begin{array}{lcl} rcv & : & React \to Receive \\ rcv\ (Read\ f) & = & RnQ\ (\lambda e.\, rcv\,(f(e))) \\ rcv\ (Write\ (e,t)) & = & Step\ (rcv(t)) \\ rcv\ (Step\ t) & = & Step\ (rcv(t)) \end{array}$$

When an input is obtained at security level ℓ, secure multi-execution dictates that it should be observable for executions at level ℓ' such that $\ell \sqsubseteq \ell'$. In order to distribute data to the appropriate executions, we use queues of events. Constructor RnQ is then denoting the fact that every input data is obtained from the environment and placed in the appropriate queues (see function *join*).

Interaction trees *Produce* are coinductively defined as follows.

$$\begin{array}{rl} Produce = & Reuse\ (E_I \to Produce) \\ & \mid Write\ (E_O \times Produce) \\ & \mid Step\ Produce \end{array}$$

The producer function is parameterised by a security level (prd_ℓ). Reads are replaced by fetching data obtained by the receiver (constructor *Reuse*). How data is reused is explained in detail in the function *join*. Writes are only performed if the security level of the channel coincides with the security level of the producer.

$$\begin{array}{ll} prd_\ell & : React \to Produce \\ prd_\ell\ (Read\ f) & = Reuse\ (\lambda e.\, prd_\ell\,(f(e))) \\ prd_\ell\ (Write\ (e,t)) & = \textbf{if}\ \ell = lvl(e) \\ & \quad\ \textbf{then}\ Write\ (e, prd_\ell(t)) \\ & \quad\ \textbf{else}\ Step\ (prd_\ell(t)) \\ prd_\ell\ (Step\ t) & = Step\ (prd_\ell(t)) \end{array}$$

Interaction trees of the kind *Produce* are isomorphic to *React*. Moreover, *Receive* can be embedded into *React*. In this light, functions *rcv* and prd_ℓ could be defined to simply produce *React* interaction trees instead. However, we have chosen to introduce new types of interaction trees in order to make the presentation more intuitive.

The following relationships are self-explanatory and they are useful to prove the propositions in Section 4. As done with the evaluation relation for interaction trees, we define evaluation relations for producers and receivers.

Definition TR 3 (Evaluation relation for producers)**.** Let $p : Produce$. The evaluation relation $\rightrightarrows \subseteq (Produce \times \mathcal{I}) \times \mathcal{O}$ is defined coinductively as follows

$$\frac{}{(Reuse(f), []) \rightrightarrows []} \qquad \frac{(f(e), i) \rightrightarrows o}{(Reuse(f), e :: i) \rightrightarrows o}$$

$$\frac{(t, i) \rightrightarrows o}{(Write(e,t), i) \rightrightarrows e :: o} \qquad \frac{(t, i) \rightrightarrows o}{(Step(t), i) \rightrightarrows \bullet :: o}$$

Definition TR 4 (Evaluation relation for receivers)**.** Let p : *Receive*. The evaluation relation $\xhookrightarrow{\star} \subseteq (\mathit{Receive} \times \mathcal{I}) \times \mathit{Colist}_{\{\star,\bullet\}}$ is defined coinductively as follows

$$\frac{}{(\mathit{RnQ}(f), []) \xhookrightarrow{\star} []} \qquad \frac{(f(e), i) \xhookrightarrow{\star} s}{(\mathit{RnQ}(f), e :: i) \xhookrightarrow{\star} \star :: s}$$

$$\frac{(t, i) \xhookrightarrow{\star} s}{(\mathit{Write}(e, t), i) \xhookrightarrow{\star} \bullet :: s} \qquad \frac{(t, i) \xhookrightarrow{\star} s}{(\mathit{Step}(t), i) \xhookrightarrow{\star} \bullet :: s}$$

If we replace *Receive* by *React* and *RnQ* by *Read* in Definition TR 4, we obtain an evaluation relation or *React* interaction trees that takes into account *when* an input event is consumed (marked with $\star$). This information is not present in the evaluation relation $\Rightarrow$. Definition TR 5 formally defines such a relation for *React*.

Definition TR 5. Let t : *React*. The evaluation relation $\hookrightarrow \subseteq (\mathit{React} \times \mathcal{I}) \times \mathit{Colist}_{\{\star,\bullet\}}$ is defined coinductively as follows

$$\frac{}{(\mathit{Read}(f), []) \hookrightarrow []} \qquad \frac{(f(e), i) \hookrightarrow s}{(\mathit{Read}(f), e :: i) \hookrightarrow \star :: s}$$

$$\frac{(t, i) \hookrightarrow s}{(\mathit{Write}(e, t), i) \hookrightarrow \bullet :: s} \qquad \frac{(t, i) \hookrightarrow s}{(\mathit{Step}(t), i) \hookrightarrow \bullet :: s}$$

The following lemmas relate evaluation of interaction trees with evaluation of producers and receivers, respectively.

Lemma TR 6. *Let t : React, $\ell \in \mathcal{L}$, and i an input colist. If $(t, i) \Rightarrow o_1$ and $(\mathit{prd}_\ell(t), i) \rightrightarrows o_2$, then $o_1 \simeq_\ell^{ID} o_2$. Moreover, $\mathit{fin}(o_1)$ iff $\mathit{fin}(o_2)$; where $\mathit{fin}(s)$ is true if and only if s is a finite colist.*

Proof. As $\simeq^{CP}$ implies $\simeq^{ID}$, this result is a direct consequence of Lemma TR 9. $\square$

Lemma TR 7. *Let t : React, and i be an input colist. If $(t, i) \hookrightarrow s$ and $(\mathit{rcv}(t), i) \xhookrightarrow{\star} s'$, then $s \equiv s'$.*

Proof. By coinduction on t. $\square$

In order to define similarity notions on colist having the special event $\star$, we assume $\forall \ell, \mathit{visible}_\ell(\star)$.

3.2 Obtaining ID-secure interaction trees

Once we have obtained the receiver and the producers, we proceed to join them into a single interaction tree *React*. We do this by choosing commands from different trees, as dictated by a scheduler. Schedulers are modelled as streams of elements in $\mathcal{L} \cup \{\star\}$, where the symbol $\star$ accounts for the receiver, while each $\ell \in \mathcal{L}$ accounts for the producer at level ℓ. We denote with *Sch* the set of all schedulers.

In order to receive data from the receiver, we equip each producer with a queue of input events. We model this *system of queues* by a function from levels to queues of input events, i.e., $Q = \mathcal{L} \to \mathit{Queue}\,(E_I)$. As dictated by secure multi-execution, input events are distributed to producers only when the security level of the producer is equal or higher than the security level of the event. We define the operator $\oplus$ responsible for the proper distribution of input events to producers via queues. More precisely, given $q \in Q$ and an event $e \in E_I$, we define

$$q \oplus e = \lambda\ell.\ \textbf{if}\ \mathit{lvl}(e) \sqsubseteq \ell \\ \qquad\qquad \textbf{then}\ \mathit{enqueue}\,(q(\ell), e) \\ \qquad\qquad \textbf{else}\ q(\ell)$$

where the function *enqueue* just adds an event to a queue. Observe that function $\oplus$ modifies a system of queues q by appending a new event e to each queue corresponding to a level ℓ such that $\mathit{lvl}(e) \sqsubseteq \ell$.

Function *join*, the most interesting function of the transformation, takes a scheduler, a receiver, a producer for each security level, and a system of queues. As a result, it calculates an ID-secure interaction tree. More precisely, the type signature of *join* is as follows.

$$\mathit{join} : \mathit{Sch} \times \mathit{Receive} \times (\mathcal{L} \to \mathit{Produce}) \times Q \to \mathit{React}$$

The function is defined by pattern-matching on the scheduler. If the scheduler dictates that it is the turn of the receiver ($\star$), *join* reproduces each *Step* in the receiver's interaction tree until it finds a *RnQ*. At this point *join* should perform a *Read*. However, in order to avoid leaking secrets through the termination channel, it must first check that all producers have consumed their input queues and that they are waiting for more input. This situation is verified by the predicate $\mathit{sync}(p, q)$, which holds iff the next interaction in every producer is *Reuse*, and $q(\ell)$ is empty for all ℓ. Hence, when the producers are synched, a *Read* can be performed and the input event added to the corresponding queues. If producers are not synched (for example, because there is a producer whose first action is to do a *Step*) then *join* should execute producers in order to try to synchronise them. This execution of producers is performed by the function *next*.

$$
\begin{array}{l}
\mathit{join}\ \ (\star :: s, r, p, q) = \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Step}\ r' \ \ \rightarrow\ \ \mathit{Step}\ (\mathit{join}\ \ (\star :: s, r', p, q)) \\
\quad \mathit{RnQ}\ f \ \ \rightarrow\ \ \textbf{if}\ \mathit{sync}(p,q)\ \textbf{then} \\
\qquad\qquad\qquad \mathit{Read}\ (\lambda e.\, \mathit{join}\ \ (s, f(e), p, q \oplus e)) \\
\qquad\qquad\qquad \textbf{else let}\ (p', q') = \mathit{next}(p,q) \\
\qquad\qquad\qquad\quad \textbf{in}\ \mathit{Step}\ \ (\mathit{join}\,(\star :: s, r, p', q'))
\end{array}
$$

Here, $\mathit{next}(p,q) = (\lambda\ell.\,(\mathit{step}(p,q,\ell))_1, \lambda\ell.\,(\mathit{step}(p,q,\ell))_2)$ is the function that tries to make a single step at every security level ℓ using the function *step*. The function *step*, in turn, tries to make a single step at a single level. The subindices in the definition of *next* denote pair projections. The function *step* is defined as follows.

$$
\mathit{step}(p,q,\ell) = \begin{cases} (p_\ell, q(\ell)) & \text{if } p(\ell) = \mathit{Step}\ p_\ell \\ (f(e), q_\ell) & \text{if } \quad p(\ell) = \mathit{Reuse}\ f \\ & \qquad \wedge\, q(\ell) = e :: q_\ell \\ (p(\ell), q(\ell)) & \text{otherwise} \end{cases}
$$

This function simply makes a computation step on a producer unless there is a *Write* or a *Reuse* with an empty queue.

Continuing with the definition of *join*, if the scheduler dictates that it is the turn of the producer at level ℓ, *join* will inspect the producer tree corresponding to level ℓ and execute it until it finds a write. If that producer would perform a write, a *Write* is added to the resulting tree. If the producer tries to reuse an event when there is none, it just yields the execution; if, on the other hand, there is an event, it gets consumed. If the producer makes a *Step*, *join* will replicate it.

$$
\begin{array}{l}
\mathit{join}\ \ (\ell :: s, r, p, q) = \textbf{case}\ p(\ell)\ \textbf{of} \\
\quad \mathit{Write}\,(o, p_\ell) \rightarrow\ \mathit{Write}\,(o, \mathit{join}\ \ (s, r, p[\ell \mapsto p_\ell], q)) \\
\quad \mathit{Step}\ p_\ell \qquad\ \rightarrow \mathit{Step}\,(\mathit{join}\ \ (\ell :: s, r, p[\ell \mapsto p_\ell], q)) \\
\quad \mathit{Reuse}\ f \qquad \rightarrow \textbf{case}\ q(\ell)\ \textbf{of} \\
\qquad e :: es \rightarrow \mathit{Step}\,(\mathit{join}\ \ (\ell :: s, r, p[\ell \mapsto f(e)], q[\ell \mapsto es])) \\
\qquad [] \qquad \rightarrow \mathit{Step}\,(\mathit{join}\ \ (\ell :: s, r, p, q))
\end{array}
$$

Given a scheduler, secure multi-execution for interaction trees is defined by the following function.

$$
\begin{array}{lcl}
\mathit{sme} & : & \mathit{Sch} \times \mathit{React} \rightarrow \mathit{React} \\
\mathit{sme}\ (s,t) & = & \mathit{join}\,(s, \mathit{rcv}(t), \lambda\ell.\, \mathit{prd}_\ell(t), \lambda\ell.\, [])
\end{array}
$$

The next proposition states the security of *sme*.

Proposition 7. *Let $t \in \mathit{React}$ and $s \in \mathit{Sch}$. Then, the interaction tree $\mathit{sme}\ (s,t)$ is ID-secure.*

Similarly to [7, 9, 17, 24], we can prove that the transformation preserves the semantics of ID-secure programs when the interleaving of events at different security levels is not relevant. Proposition 7 and Corollary 25 in Appendix B, quantify over all schedulers. This might seem surprising, as clearly one can choose a bad scheduler. For instance, we could choose the scheduler that always chooses the receiver. Such a scheduler would never issue a *Write*, and therefore would always diverge. As discussed in Section 2, a silent infinite colist is ID-similar to every other one. In particular, the output colist $\perp = \bullet :: \perp$ is infinite and silent at every level, and therefore ID-similar to every output colist. Therefore, the trivial transformation $bad(t) = diverge$ satisfies the security and transparency propositions. After all, *diverge* is ID-secure and produces an output ($\perp$) ID-similar to any other output. We believe that secure multi-execution can do better than the trivial diverging transformation, but in order to show it we need to state better formal guarantees.

In the next section, we present one of the main contributions of this paper. We show how the program under execution and its input induce a scheduler that significantly improves the transparency guarantees of secure multi-execution.

4 Order-preserving Scheduler

The definition of secure multi-execution in the previous section is parameterised by a scheduler. However, if we are interested in the order in which events from different levels are produced, the choice of scheduler is of paramount importance. The standard precision result for secure multi-execution [7, 9, 17, 24] ensures that the order of output events is preserved only when looking at a given security level in isolation. However, in certain scenarios (such as the monitor in Section 6) one needs to take into account the interleaving of events from different security levels. Therefore, a stronger guarantee is required. Example 8, although very simple, illustrates this point.

Example 8. We define program p with just one handler as follows:

$$
\begin{aligned}
p = \texttt{L?(x)}\{\ & \texttt{out(L!,x);} \\
& \texttt{if x > 10 } \{\texttt{out(H!,x)}\} \\
& \qquad\qquad\quad \{\texttt{skip}\}; \\
& \texttt{out(L!,1) } \};
\end{aligned}
$$

This program is non-interferent. Since there is no handler for channel `H?`, every secret input is ignored.

It is easy to see that, for all inputs, every event on channel `H!` is preceded by an event on channel `L!` with exactly the same value. However, the secure multi-execution of p with a $\texttt{select}_{\text{lowprio}}$ scheduler has a different behaviour for a high observer: every event on channel `H!` is preceded by an event with constant 1 on `L!`.

In order to preserve the order of events we will look at the order of events generated by the original program. That is, we will use the execution of the original program to guide secure multi-execution. If the original execution issues a *Read* command, the scheduler chooses the receiver, identified as $\star$, to run. Observe that this is the only interaction tree under secure multi-execution capable of issuing such command. Instead, if the original execution issues a *Write* command to a channel at level ℓ, then the producer $p(\ell)$ is run. Observe that $p(\ell)$ is the only interaction tree under secure multi-execution that is able to perform *Write*s into channels at level ℓ. However, if the original execution issues a *Step*, there is no information from which to decide what to schedule next. To account for this situation, we extend our definition of schedulers (*Sch*) in Section 3.2 with the element $\circ$. Finally, if read commands are issued by the execution of the original program under an empty colist of input events, it does not really matter which program under secure multi-execution gets scheduled. After all, the execution of the original program has stopped (see rule (R_1) in Figure 2.1). More precisely, the order-preserving scheduler, called *ops*, is defined as follows.

$$\begin{array}{lcl}
ops & : & React \times \mathcal{I} \to Sch \\
ops\ \ (Read\ f, e :: i) & = & \star :: (ops\ \ (f(e), i)) \\
ops\ \ (Read\ f, []) & = & \circ :: (ops\ \ (Read\ f, [])) \\
ops\ \ (Write\ (e, t), i) & = & lvl(e) :: ops\ \ (t, i) \\
ops\ \ (Step\ p, i) & = & \circ :: ops\ \ (p, i)
\end{array}$$

The scheduler takes the interaction tree of the program to be executed under secure multi-execution (*React*), the colist of inputs ($\mathcal{I}$), and returns the scheduling policy (*Sch*). Observe how reads in the presence of inputs are mapped to the receiver ($\star$), while writes are mapped into producers with the same security level as the channel ($lvl(e)$).

As a consequence of adding symbol $\circ$ to the scheduler, we need to extend the definition of *join* with the following additional case.

$$join\ \ (\circ :: s, p, r, q) \quad = \quad Step\ (join\ \ (s, p, r, q))$$

When *join* finds the symbol $\circ$ in the scheduling policy, it simply makes a *Step*.

4.1 Transparency guarantees for the scheduler *ops*

The order-preserving scheduler *ops* allows secure multi-execution to provide better transparency guarantees than the ones previously shown.

Theorem 9 (Transparency for ID-secure trees)**.** *Let t be an ID-secure interaction tree, and i an input colist such that $(t, i) \Rightarrow o$. If $(sme(ops(t, i), t), i) \Rightarrow o'$ then $\forall \ell.\, o \sim^{ID}_{\ell} o'$.*

Proof. Since t is an ID-secure interaction tree, it follows from Lemma 16 that every input is secure for it. In particular, i is ID-secure for t. Hence, by Theorem 18 we have the desired result. $\square$

The theorem above states that the output of the original program and its secure multi-execution are ID-similar for ID-secure interaction trees. This means that for any observer at some level ℓ, the order of ℓ-visible events is preserved. This is an improvement over previous results since it considers the interleaving of events. Nevertheless, satisfying transparency for ID-secure programs does not guarantee that secure multi-execution performs any progress, e.g., *sme* might always diverge (see discussion at the end of Section 3.)

In order to guarantee progress, we consider a stronger notion of non-interference for reactive systems called CP-security [11].

4.1.1 CP-security

We coinductively define when a colist of events is not visible (silent) for an observer at level ℓ as follows.

$$\frac{\neg visible_\ell(e) \qquad silent_\ell(s)}{silent_\ell(e :: s)} \qquad\qquad \frac{}{silent_\ell([])}$$

We define a relation that identifies the next event that it is visible to an observer at level ℓ (if exists). Intuitively, we say that $s \triangleright_\ell e :: s'$ when e is the next event in s visible at level ℓ. The following rules inductively define the relation $\triangleright_\ell$.

$$\frac{visible_\ell(e)}{e :: s \triangleright_\ell e :: s} \qquad\qquad \frac{\neg visible_\ell(e) \qquad s \triangleright_\ell e' :: s'}{e :: s \triangleright_\ell e' :: s'}$$

Note that the relation is inductively defined, which means that when $s \triangleright_\ell e :: s'$, the next ℓ-visible event e of the colist s must come after a finite sequence of ℓ-invisible events.

Definition 10 (CP-similarity)**.** CP-similarity between colists is defined coinductively by the following rules.

$$\frac{silent_\ell(s) \qquad silent_\ell(s')}{s \sim_\ell^{CP} s'}$$

$$\frac{s \triangleright_\ell e :: s_1 \qquad s' \triangleright_\ell e :: s_1' \qquad s_1 \sim_\ell^{CP} s_1'}{s \sim_\ell^{CP} s'}$$

As opposed to ID-similarity, CP-similarity is an equivalence relation. Moreover, CP-similarity guarantees progress by asking one colist to be as productive as the other, i.e. the two colists either produce the same visible event in a finite number of steps, or both become silent. We now define when a program is CP-secure.

Definition 11 (CP-security)**.** An interaction tree t is CP-secure iff, for every level ℓ and input colists i, i' such that $(t, i) \Rightarrow o$ and $(t, i') \Rightarrow o'$, $i \sim_\ell^{CP} i'$ implies $o \sim_\ell^{CP} o'$.

CP-security is strictly stronger than ID-security: any CP-secure program is ID-secure [11].

Level by level CP-security

In order to define level-by-level CP-similarity, we start by defining a predicate $sil_{=\ell}$ to represent the fact that a stream s has no events at a particular security level ℓ:

$$\frac{lvl(e) \neq \ell \qquad sil_{=\ell}(s)}{sil_{=\ell}(e :: s)} \qquad\qquad \frac{}{sil_{=\ell}([])}$$

The relation $\trianglerighteq_\ell$ is inductively defined as follows.

$$\frac{lvl(e) = \ell}{e :: s \trianglerighteq_\ell e :: s} \qquad\qquad \frac{lvl(e) \neq \ell \qquad s \trianglerighteq_\ell e' :: s'}{e :: s \trianglerighteq_\ell e' :: s'}$$

Intuitively, we say that $s \trianglerighteq_\ell e :: s'$ if e is the first event in s such that $lvl(e) = \ell$, and s' is obtained from s removing all events up to e.

Definition TR 8 (level-by-level CP-similarity)**.** The following relation coinductively defines CP-similarity for events at security level ℓ.

$$LS_1^{CP}\,\frac{sil_{=\ell}(s) \qquad sil_{=\ell}(s')}{s \simeq_\ell^{CP} s'} \qquad\qquad LS_2^{CP}\,\frac{s \trianglerighteq_\ell e :: s_1 \qquad s' \trianglerighteq_\ell e :: s_1' \qquad s_1 \simeq_\ell^{CP} s_1'}{s \simeq_\ell^{CP} s'}$$

We have the following result about the producer tranformation from Sect. **??**. It relates the behaviour of a reactive interaction tree with the behaviour of its producers.

Lemma TR 9. *Let t : React, $\ell \in \mathcal{L}$, and i an input colist. If $(t, i) \Rightarrow o_1$ and $(prd_\ell(t), i) \rightrightarrows o_2$, then $o_1 \simeq_\ell^{CP} o_2$. Moreover, $fin(o_1)$ iff $fin(o_2)$.*

Proof. By coinduction on t. $\square$

We show that secure multi-execution is transparent with respect to CP-secure programs when using the order preserving scheduler.

Theorem 12 (Transparency for CP-secure trees)**.** *Let t be CP-secure interaction tree, and i an input colist such that $(t, i) \Rightarrow o$. If $(sme(ops(t, i), t), i) \Rightarrow o'$ then $\forall \ell.\, o \sim_\ell^{CP} o'$.*

Proof. Since t is a CP-secure interaction tree, it follows from Lemma 15 that every input is secure for it. In particular, i is CP-secure for t. Hence, by Theorem 19 we have the desired result. $\square$

The transparency theorem for CP-secure trees is a significant improvement over previous results for secure multi-execution in reactive systems. Previous results did not show that secure multi-executions approaches fulfilling the security and transparency properties were any better than the transformation that always produces diverging runs. The above theorem, however, is able to guarantee progress for CP-secure programs as well as event-order preservation. Hence, if a CP-secure program produces a visible event, its secure multi-execution is forced to produce it too ($o \sim_\ell^{CP} o'$).

5 Secure Inputs

We want to precisely detect leaks of secret information in reactive systems. Non-interference, a property of programs, cannot be precisely enforced by an execution monitor [29, 39]. More importantly, it may not be a desirable property to enforce. For instance, many web applications deployed on the web might be harmless most of the time, but leak information only in certain situations. That is, they may leak information in certain runs, but not on others. In this light, it is not surprising that some information-flow monitors accept runs of interferent programs as long as they do not leak information. For instance, monitors in [2, 5, 19, 34, 37, 42] accept the runs of program `if public = 42 then public := secret else skip` when the public input is different from 42. With this in mind, we define a security condition on runs (rather than on programs), by characterising the inputs for which programs do not leak secret information (we ignore leaks due to covert channels.) In order to define this notion, we need to present an auxiliary relation.

We coinductively define the relation $\blacktriangleright_\ell$ responsible for removing all the events unobservable at level ℓ.

$$\frac{\mathit{silent}_\ell(s)}{s \blacktriangleright_\ell []} \qquad\qquad \frac{s \triangleright_\ell e :: s' \qquad s' \blacktriangleright_\ell s''}{s \blacktriangleright_\ell e :: s''}$$

Observe that given a colist s, there is a unique colist s' such that $s \blacktriangleright_\ell s'$. We will write $s_{\blacktriangleright_\ell}$ for this unique colist, and refer to it as the restriction of s at level ℓ.

Let us assume a level-indexed similarity relation $\sim_\ell$ between colists. Two inputs for a program reveal the same secrets at a given security level ℓ if, for an observer at level ℓ, they are similar and induce similar outputs.

Definition 13 ($\approx_{\ell,t}$)**.** Let $t \in \mathit{React}$, $\ell \in \mathcal{L}$, and i, i' input colists such that $i \sim_\ell i'$, $(t, i) \Rightarrow o$ and $(t, i') \Rightarrow o'$. We say that the program t reveals the same ℓ-secrets when given the inputs i, i', noted $i \approx_{\ell,t} i'$ iff $o \ \sim_\ell \ o'$.

Similarly to [9], we consider an input to be *secure* for a program t if it reveals the same information about the secrets as the input where secrets have been erased.

Definition 14 (Secure input)**.** Let t be an interaction tree. An input colist i is *secure for* t iff $\forall \ell.\, i \approx_{\ell,t} i_{\blacktriangleright_\ell}$. We say that the input i is ID-secure (CP-secure) for t when $\sim_\ell$ is instantiated to $\sim_\ell^{\scriptscriptstyle ID}$ ($\sim_\ell^{\scriptscriptstyle CP}$) in Definition 13.

It is desirable to establish a connection between programs and their secure inputs, so as to be able to transfer security properties from one notion to the other. Fortunately, there is a close relationship between CP-secure programs and CP-secure inputs.

Lemma 15 (Secure inputs and CP-security)**.** *A reactive interaction tree $t \in \mathit{React}$ is CP-secure iff $\forall i \in \mathcal{I}.\ i$ is CP-secure for t.*

$$p = \texttt{H?(x)}\{\ \texttt{r := x}\ \}; \tag{1}$$

$$\begin{aligned}&\texttt{L?(x)}\{\ \texttt{if r} \geq \texttt{1}\\ &\qquad\qquad \{\texttt{out(L!,r)}\}\\ &\qquad\qquad \{\texttt{while 1 do skip}\ \}\};\end{aligned}$$

Figure 4: ID-insecure program with ID-secure inputs

Proof.

$\Rightarrow$) Since t is CP-secure, we have that $\forall \ell, i, i'$ such that $i \sim_\ell^{CP} i'$,

$$(t, i) \Rightarrow o \quad \text{and} \quad (t, i') \Rightarrow o' \qquad \text{imply} \quad o \sim^{CP} o'.$$

If we substitute $i_{\blacktriangleright_\ell}$ for i' above, we obtain that i and $i_{\blacktriangleright_l}$ produce CP-similar outputs, and therefore i is CP-Secure for t.

$\Leftarrow$) Let i, i' be input colists such that $i \sim_\ell^{CP} i'$. The relation $\sim^{CP}$ is transitive, and thefore $i'_{\blacktriangleright_\ell} \sim^{CP} i' \sim_\ell^{CP} i \sim_\ell^{CP} i_{\blacktriangleright_\ell}$. Note that $i_{\blacktriangleright_\ell} \sim_\ell^{CP} i'_{\blacktriangleright_\ell} \iff i_{\blacktriangleright_\ell} = i'_{\blacktriangleright_\ell}$, and therefore all of these input colists will produce CP-similar results.

□

When it comes to ID-security, the relationship between secure programs and secure inputs is not that strong.

Lemma 16 (Secure inputs and ID-security)**.** *If a reactive interaction tree $t \in$ React is ID-secure, then $\forall i \in \mathcal{I}$. i is ID-secure for t.*

Proof. Since t is ID-secure, we have that $\forall \ell, i, i'$ such that $i \sim_\ell^{ID} i'$,

$$(t, i) \Rightarrow o \quad \text{and} \quad (t, i') \Rightarrow o' \qquad \text{imply} \quad o \sim^{ID} o'.$$

If we substitute $i_{\blacktriangleright_\ell}$ for i' above, we obtain that i and $i_{\blacktriangleright_l}$ produce ID-similar outputs, and therefore i is ID-Secure for t. □

It is easy to prove that all inputs are secure for a secure program, in both the ID-security and the CP-security case. However, it might not be obvious that a program such that every input is CP-secure is a CP-secure program. Note that whenever two inputs i, i' are similar at some level ℓ they have exactly the same restriction at that level.

$$i \sim_\ell^{CP} i' \quad \Longrightarrow \quad i_{\blacktriangleright_\ell} = i'_{\blacktriangleright_\ell}$$

By definition of secure input and transitivity of $\sim_\ell^{CP}$, we can conclude that the output streams produced by t with i and i' must be CP-similar, provided that i and i' are CP-secure.

Lemma 16 indicates that if a program is ID-secure, then every input for that program is ID-secure, i.e. running the program under those inputs leaks the same amount of information as if secrets had been erased. The converse, however, does not hold. We illustrate this point with the following example.

Example 17. Consider the program p in Figure 4. Every input is ID-secure for p but p is not an ID-secure program, as the following two ID-similar inputs show.

$$i = [(\mathtt{H?}, 1), [(\mathtt{L?}, 0)] \quad \sim_L^{ID} \quad i' = [(\mathtt{H?}, 2), (\mathtt{L?}, 0)]$$

Let μ_0 be the initial memory where every variable is initialised to 0, and let $t = [\![p]\!](\mu_o)$, the interaction tree obtained from p and μ_0. Then, we see that t is not ID-secure, since i and i' are ID-similar at level L, but their outputs are not ID-similar at level L.

$$\begin{array}{ccc} (t, i) & \Rightarrow & [\bullet, \bullet, \bullet, \bullet, (\mathtt{L!}, 1)] \\ & & \not\sim_L^{ID} \\ (t, i') & \Rightarrow & [\bullet, \bullet, \bullet, \bullet, (\mathtt{L!}, 2)] \end{array}$$

Nevertheless, all inputs i are ID-secure for t (Definition 14). The key observation here is that t diverges for $i_{\blacktriangleright L}$ (which coincides with $i'_{\blacktriangleright L}$), since r was initially zero. In other words, the input colist without events on channel `H?` produces an output which is silent and infinite, and hence ID-similar to every other output.

Secure inputs and producers The next two lemmas provide useful invariants to prove the transparency results for ID- and CP-secure inputs.

Lemma TR 10 (ID-secure inputs and producers)**.** *Let t : React, $\ell \in \mathcal{L}$, and i an ID-secure input for t such that $(t, i) \Rightarrow o$. Then,*

$$(prd_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq_\ell^{ID} o_\ell$$

Proof. Different from the proof of Lemma TR-11, ID-similarity is not transitive, so we need to proceed carefuly. As i is ID-secure for t, we have $(t, i_{\blacktriangleright \ell}) \Rightarrow o'$, and $o \sim_\ell^{ID} o'$. Therefore $o \simeq_\ell^{ID} o'$. From Lemma TR- 9 we have $o' \simeq_\ell^{CP} o_\ell$, and we know that o' is finite if and only if o_ℓ is finite. Using these facts, it is easy to conclude that $o \simeq_\ell^{ID} o_\ell$.

□

The next lemma is similar to the previous one but considers CP-secure inputs.

Lemma TR 11 (CP-secure inputs and producers)**.** *Let t : React, $\ell \in \mathcal{L}$ and i a CP-secure input for t such that $(t, i) \Rightarrow o$. Then,*

$$(prd_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq_\ell^{CP} o_\ell$$

Proof. As i is CP-secure for t, we have $(t, i_{\blacktriangleright \ell}) \Rightarrow o'$, and $o \sim_\ell^{CP} o'$. Therefore $o \simeq_\ell^{CP} o'$. From Lemma TR- 9 we have $o' \simeq_\ell^{CP} o_\ell$. By transitivity of $\simeq_\ell^{CP}$, we conclude $o \simeq_\ell^{CP} o_\ell$. □

The following theorems state the transparency of secure inputs for secure multi-execution with our order-preserving scheduler. That is, the theorems show that outputs of a given program under secure multi-execution are not observably different when provided with a secure input.

In order to prove the transparency theorems, we will first prove two technical lemmas that establish the key invariant needed in the proofs.

Lemma TR 12. *Let t be a reactive interaction tree, i an input colist CP-secure for t such that $(t, i) \Rightarrow o$, r : Receive, $p : \mathcal{L} \to$ Produce, $q : Q$, and $sch = ops(t, i)$. Suppose that*

$$\forall \ell, (p(\ell), q(\ell) +\!\!+\, i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq^{CP}_\ell o_\ell \tag{2}$$

$$(r, i) \xhookrightarrow{\star} s \wedge (t, i) \hookrightarrow s' \Rightarrow s \simeq^{CP} s' \tag{3}$$

If $(join(sch, r, p, q), i) \Rightarrow o'$, then $\forall \ell, o \sim^{CP}_\ell o'$.

Proof. Let $\ell \in \mathcal{L}$. We have to show that $o \sim^{CP}_\ell o'$. The proof is by coinduction on the proof of $(t, i) \Rightarrow o$, by showing that all rules preserve the invariants (2) and (3). □

We prove a similar result for ID-secure inputs.

Lemma TR 13. *Let t be a reactive interaction tree, i an input colist ID-secure for t such that $(t, i) \Rightarrow o$, r : Receive, $p : \mathcal{L} \to$ Produce, $q : Q$, and $sch = ops(t, i)$. Suppose that*

$$\forall \ell, (p(\ell), q(\ell) +\!\!+\, i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq^{ID}_\ell o_\ell \tag{4}$$

$$(r, i) \xhookrightarrow{\star} s \wedge (t, i) \hookrightarrow s' \Rightarrow s \simeq^{CP} s' \tag{5}$$

If $(join(sch, r, p, q), i) \Rightarrow o'$, then $\forall \ell, o \sim^{ID}_\ell o'$ [3].

Proof. Analogous to Lemma TR 12. Only case $t = Write(e, t')$ has a different proof; since we need to take into account the case in wich the evaluation of t with input i generates a visible event al level ℓ but $prod_\ell(t)$ silently diverges. □

Theorem 18 (Transparency for ID-secure inputs)**.** *Let t be an interaction tree, and let i be an input colist ID-secure for t such that $(t, i) \Rightarrow o$. If $(sme(ops(t, i), t), i) \Rightarrow o'$ then $\forall \ell. o \sim^{ID}_\ell o'$.*

Proof. We have

$$sme(ops(t, i), t) = join(ops(t, i), rcv(t), \lambda\ell.prd_\ell(t), \lambda\ell.[])$$

By Lemma TR 13, it is enough to show that

$$\forall \ell, (prd_\ell(t), [] +\!\!+\, i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq^{ID}_\ell o_\ell \tag{6}$$

[3] Note that (5) is exactly the same property than (3).

$$(rcv(t), i) \overset{\star}{\hookrightarrow} s \wedge (t, i) \hookrightarrow s' \Rightarrow s \simeq^{CP} s' \tag{7}$$

Property (6) is valid from Lemma TR 10, and Property (7) is a consequence of Lemma TR-7.

□

The theorem above uses $\sim_\ell^{ID}$ and therefore assures transparency under this notion of observation, i.e. differences in outputs due to divergence are not observable and therefore not captured. If one wants to distinguish productive outputs from divergence, then CP-similarity is the right notion of observation.

Theorem 19 (Transparency for CP-secure inputs)**.** *Let t be an interaction tree, and let i be an input colist CP-secure for t such that $(t, i) \Rightarrow o$. If $(sme(ops(t, i), t), i) \Rightarrow o'$ then $\forall \ell.\, o \sim_\ell^{CP} o'$.*

Proof. Analogous to Theorem 18 using Lemmas TR 11 and TR 7. □

6 Multi-execution Monitor

An important problem of the secure multi-execution approach is that it makes programs non-interferent by modifying its semantics. Consequently, it is difficult to detect if, and when, programs behave maliciously. To remedy this situation, we present a monitor capable of precisely detecting when an input is insecure for a program.

Our monitor executes the original program and its secure multi-execution in parallel, checking at each step that both executions would produce the same output command. If outputs differ, we are in presence of information leaks, and thus execution is aborted. If, on the other hand, executions remain synchronised, the output command is safe to be executed. It is crucial for the monitor to work that secure multi-execution is run under a scheduler that preserves the order of output events such as the scheduler *ops* from Section 4.

At any point during execution, the monitor might need to signal an alarm. Hence, we define a new kind of datatype that can represent the outputs of the monitor. This datatype, written $\mathcal{O}_\epsilon$, is similar to a colist of output events, but it may end with an alarm ϵ. More formally, we coinductively define $\mathcal{O}_\epsilon$ as follows.

$$\mathcal{O}_\epsilon = [] \mid \epsilon \mid O :: \mathcal{O}_\epsilon$$

At a first glance, it seems enough to simply run the interaction tree of the program under consideration in parallel with the one obtained from the *sme* transformation. However, in order to precisely detect violations of the security policy a new relation is needed. The monitor is expressed by the relation $\Downarrow \subseteq \mathit{React} \times \mathcal{I} \times (\mathcal{L} \to \mathit{Produce}) \times Q \times \mathcal{O}_\epsilon$ defined in Figure 5, where we write $(t, i, p, q) \Downarrow o$ whenever $(t, i, p, q, o) \in \Downarrow$. The intuition is that whenever $(t, i, p, q) \Downarrow o$, the evaluation of the interaction tree t, together with the input i, a producer for each security level p, and a system of queues q, results in an output o. Note that the first two components (t and i) pertain to the execution

$$
(W_1)\ \frac{lvl(e) = \ell \qquad p(\ell) = \mathit{Write}(e, p_\ell) \qquad (t, i, p[\ell \mapsto p_\ell], q) \Downarrow o}{(\mathit{Write}\ (e,t), i, p, q) \Downarrow e :: o}
$$

$$
(W_2)\ \frac{lvl(e) = \ell \qquad p(\ell) = \mathit{Write}(e', p_\ell) \qquad e \neq e'}{(\mathit{Write}\ (e,t), i, p, q) \Downarrow \epsilon}
$$

$$
(W_3)\ \frac{lvl(e) = \ell \qquad p(\ell) = \mathit{Reuse}\ f_\ell \qquad q(\ell) = e' :: q_\ell \qquad (\mathit{Write}\ (e,t), i, p[\ell \mapsto f_\ell(e')], q[\ell \mapsto q_\ell]) \Downarrow o}{(\mathit{Write}\ (e,t), i, p, q) \Downarrow \bullet :: o}
$$

$$
(W_4)\ \frac{lvl(e) = \ell \qquad p(\ell) = \mathit{Reuse}\ f_\ell \qquad q(\ell) = []}{(\mathit{Write}\ (e,t), i, p, q) \Downarrow \epsilon}
$$

$$
(W_5)\ \frac{lvl(e) = \ell \qquad p(l) = \mathit{Step}\ p_\ell \qquad (\mathit{Write}\ (e,t), i, p[\ell \mapsto p_\ell], q) \Downarrow o}{(\mathit{Write}\ (e,t), i, p, q) \Downarrow \bullet :: o}
\qquad
(S)\ \frac{(t, i, p, q) \Downarrow o}{(\mathit{Step}\ t, i, p, q) \Downarrow \bullet :: o}
$$

$$
(R_1)\ \frac{(f(e), i, p, q) \Downarrow o}{(\mathit{Read}\ f, e :: i, p, q) \Downarrow \bullet :: o}
\qquad
(R_2)\ \frac{}{(\mathit{Read}\ f, [], p, q) \Downarrow go(p, q)}
$$

Figure 5: Semantics for the multi-execution monitor

of the original program, while the third and fourth one (p and q) pertain to the secure multi-execution of the program.

Rules (W_1) to (W_5) concern the case where the monitored interaction tree wants to do a *Write* of an event at security level ℓ. In rule (W_1), the producer at level ℓ ($p(\ell)$) tries to write the same event, hence the event is performed ($e :: o$). In rule (W_2), on the other hand, the producer at level ℓ tries to write a different event ($e \neq e'$), so the monitor raises an alarm (ϵ) and aborts execution. In rule (W_3) and (W_4), the producer at level ℓ tries to reuse some input events. If there is an event available ($q(\ell) = e' :: q_\ell$), rule (W_3) provides the information to the producer ($p[\ell \mapsto f_\ell(e')]$). In contrast, if there is no event available ($q(\ell) = []$), rule (W_4) aborts execution (ϵ). Finally, in rule (W_5), the producer at level ℓ makes a silent step, so a silent event is perform by the monitor ($\bullet$).

In rule (S), the interaction tree tries to make a silent step (*Step*), in which case the monitor outputs a silent event ($\bullet$). Rules (R_1) and (R_2) are related to consuming input data. If there is an input event ($e :: i$), rule (R_1) consumes it ($f(e)$). If, on the other hand, there is no event, rule (R_2) determines the result of the evaluation depending on the state of system of queues, as determined by the function go.

The function go examine, when possible, if the termination behaviour of the interaction tree matches its secure multi-execution version. There are three possible situations. In the first one, every producer has consumed its input queue and the next command to be executed is a *Reuse*, thus matching the *Read* event. We indicate this situation on the producer at level ℓ by the predicate *end* ($end(p(\ell), q(\ell))$). Since at this point the interaction tree and producers are in sync, *go* just returns the empty colist. In the second situation, if there is one *Write* event as the next command to be executed by a producer at level ℓ, noted $writer(p(\ell))$, then an alarm is raised, since executions are out of sync. Lastly, it may happen that none of the two conditions above apply, e.g., there are no writes but some of the producers are on *Step* commands. In this situation, *go* should make progress on the producers until some of the two first situations occur. This search, however, may lead to divergence. More specifically, we define *go* as follows.

$$go(p,q) = \begin{cases} [] & \text{if } \forall \ell,\, end(p(\ell), q(\ell)) \\ \epsilon & \text{if } \exists \ell,\, writer(p(\ell)) \\ \bullet :: go(next(p,q)) & \text{otherwise} \end{cases}$$

where the predicate *sync* and the function *next* are those defined in Section 3. It may seem odd to go through the complication of defining the function *go* when the program was going to end anyway, but the following example shows why this is needed:

Example 20. Consider the following reactive program:

```
p =  H?(x){ r := x };

     L?(x){ if r = 0
               {out(L!,1)}
               {skip} };
```

Let t be the interaction tree obtained from it for the initial memory μ_0, and let $i = [(H,1),(L,0)]$. The evaluation of t with input i, will result in a finite sequence of invisible events.

$$(t,i) \Rightarrow [\bullet, \bullet, \bullet, \bullet, \bullet]$$

On the other hand, the evaluation of t when fed with the restriction of i at level L, i.e. $i_{\blacktriangleright L} = [(L,0)]$, has an observable event.

$$(t, i_{\blacktriangleright L}) \Rightarrow [\bullet, \bullet, (L,1)]$$

Since the outputs are distinguishable at level L, we conclude that i is not ID-secure for t.

In order to detect cases like this one, where the normal execution ends, but its execution with the secrets erased would produce an output, the monitor needs to make sure that the execution of the producers with the available inputs will not produce an output, as done by rule (R_2).

6.1 Properties of multi-execution monitoring

This section describes the most important contribution of this work. We establish the security policy enforced by our monitor as well as the transparency guarantees. To simplify notation, given an interaction tree t and input colist i, we define $monitor(t, i) = (t, i, \lambda\ell.\, prd_\ell(t), \lambda\ell. i_{\blacktriangleright_\ell})$ as the initial configuration of our monitor for auditing program t under the input events i. We define the predicate $ok \subseteq \mathcal{O}_\epsilon$ identifying outputs colists where the monitor has not raised an alarm. The predicate is defined coinductively by the following rules.

$$\frac{}{ok([])} \qquad\qquad \frac{ok(s)}{ok(e :: s)}$$

It is easy to see that an output is not ok precisely when it ends with an alarm ϵ. Clearly, we can embed any output $o \in \mathcal{O}_\epsilon$ such that $ok(o)$ in $\mathcal{O}$. To avoid additional notation, we perform such embedding transparently.

The following theorem states that the multi-execution monitor is able to precisely detect ID-secure inputs.

The next lemma is useful to show the precision result for ID-secure runs. It states a sufficient condition (in the form of an invariant) to ensure that the monitor will not raise an alarm with secure inputs.

Lemma TR 14. *Let t : React, i an input colist such that $(t, i) \Rightarrow o$, $p : \mathcal{L} \to$ Produce, and $q : Q$. Assume*

$$\forall \ell\, (p(\ell), q(\ell)) \rightrightarrows o_\ell \Rightarrow o \simeq^{ID}_\ell o_\ell \tag{8}$$

If $(t, i, p, q) \Downarrow o'$, then $ok(o')$.

Proof. We have to show that an alarm is not raised in the derivation of $(t, i, p, q) \Downarrow o'$. Note that, if Property (8) is valid on (t, i, p, q), then the rules of $\Downarrow$ raising an alarm (W_2, W_4 and R_2 in case that some producer generates a visible event) cannot be applied to this tuple. Formal proof is by coinduction on the definition of $\Downarrow$, proving that property (8) is preserved by all rules of the monitor that can be applied.

(W_1) We have

- $t = Write(e, t')$,
- $lvl(e) = \ell$,
- $p(\ell) = Write(e, p_\ell)$,
- $o = e :: o'$, and
- $(t', i, p[\ell \mapsto p_\ell], q) \Downarrow o'$.

We have to show that property (8) holds for $(t', i, p[\ell \mapsto p_\ell], q)$.

Let $\ell_1 \in \mathcal{L}$. We distinguish two cases

1. $\ell_1 = \ell$. In this case, we have to prove that

$$(t', i) \Rightarrow o'_1 \wedge (p_\ell, q(\ell)) \rightrightarrows o'_2 \Rightarrow o'_1 \simeq^{ID}_\ell o'_2$$

By definition of $\Rightarrow$, we know that $o_1 = e :: o'_1$ and by definition of $\rightrightarrows$, $o_2 = e :: o'_2$. Using the hypothesis $(o_1 \simeq^{ID}_\ell o_2)$, and rule LS_2 in definition of $\simeq^{ID}_\ell$, we conclude that $o'_1 \simeq^{ID}_\ell o'_2$.

2. $\ell_1 \neq \ell$. In this case, we have to prove that

$$(t', i) \Rightarrow o'_1 \wedge (p(\ell_1), q(\ell_1)) \rightrightarrows o'_2 \Rightarrow o'_1 \simeq^{ID}_{\ell_1} o'_2$$

By definition of $\Rightarrow$, we know that $o_1 = e :: o'_1$ and by definition of $\rightrightarrows$, $o_2 = o'_2$. Using the hypothesis $(o_1 \simeq^{ID}_\ell o_2)$, the fact that $lvl(e) = l \neq \ell_1$, we can apply rule LS_4 in definition of $\simeq^{ID}_{\ell_1}$ to conclude that $o'_1 \simeq^{ID}_{\ell_1} o'_2$.

► The proof for the other rules are analogous.

□

Theorem 21 (Precision for ID-secure runs)**.** *Let t : React and let i be an input colist such that $monitor(t, i) \Downarrow o$. Then*

$$i \textit{ is ID-secure for } t \iff ok(o)$$

Proof.

($\Longrightarrow$) By Lemma TR-14, it is enough to show that

$$\forall \ell((prd_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq^{ID}_\ell o_\ell$$

This property is true by Lemma TR 10.

($\Longleftarrow$) By contraposition. Assume i not ID-secure for t, i.e. there exists a level ℓ such that $(t, i_{\blacktriangleright \ell}) \Rightarrow o_1$, and $o \not\simeq^{ID}_\ell o_1$. Let o_ℓ such that $(prod_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell$. Applying Lemma TR 9, we know that $o_\ell \simeq^{CP}_\ell o_1$ and $fin(o_\ell) \iff fin(o_1)$. Then, it is easy to show that $o \not\simeq^{ID}_\ell o_\ell$.

Therefore, one of the following cases occur

1. $silent_\ell(o_\ell)$, $fin(o_\ell)$ and $o \rhd_\ell e :: o'$
2. $o_\ell \rhd_\ell e :: o'_\ell$, $silent_\ell(o)$ and $fin(o)$
3. $o_\ell \rhd_\ell e' :: o'_\ell$, $o \rhd_\ell e :: o'$, and $e \neq e'$.

We show that in all cases the monitor raises an alarm.

Case (1) raises an alarm by rule (W_4). Since e is an event in o, after a finite number of evaluation steps, the tree t will issue a *Write*. Note that, if the producer al level ℓ generates a silent and finite colist (o_ℓ), then after a finite number of silent steps it will be of the form *Reuse*(f) and $q(\ell)$ will be the empty colist. So rule (W_4) will be applied after a finite number of steps and an alarm is raised.

Case (2) raises an alarm by rule (R_2). If o is finite and silent at level ℓ, then after a finite number of steps will be in the case of rule (R_2). As there exists a producer that generates an event visible at ℓ (e), function *go* will detect this situation and an alarm is raised.

Case (3) raises an alarm by rule (W_2). When the execution of t tries to write event e, the producer at level ℓ is inspected and after a finite number of steps the preconditions of rule (W_2) are valid and a alarm is raised.

□

This theorem states that, if and only if the monitor raises an alarm $(\neg ok(o))$, the run i is not ID-secure. Consequently, the run has tried to leak more information than the one observed in a run where secrets are not present in the system (Definition 14). In this case, having detected such condition, the monitor assures that the program under surveillance is not ID-secure (contrapositive of Lemma 16). Differently from most of the dynamic monitoring techniques for confidentiality (e.g. [2, 5, 19, 28, 37, 40, 42]), our monitor does not raise false alarms due to some imprecision in the analysis of information flow inside a program.

When our monitor does not raise an alarm, we cannot infer the ID-security of the program, only the ID-security of the observed run. It could be the case that the program is ID-secure, in which case the monitor will never raise an alarm (Lemma 16), but also that the program is interferent but the input was ID-secure. This last case is common in dynamic monitors which accept non-leaking runs of interferent programs [2, 5, 19, 34, 37, 42].

To sum up, our monitor can precisely detect if a run is ID-secure. However, we can only assert that a program is not ID-secure when an alarm is raised. Otherwise, the monitor has not enough information to determine if the program is ID-secure or we are in presence of an ID-secure input. In either case, it is guaranteed that the output will be ID-similar to the output obtained without secrets.

The invariant in Lemma TR 14 ensures that the monitor will be transparent for secure inputs.

Lemma TR 15. *Let t : React, i an input colist such that $(t, i) \Rightarrow o$, $p : \mathcal{L} \rightarrow$ Produce, and $q : Q$. Assume*

$$\forall \ell((p(\ell), q(\ell)) \rightrightarrows o_\ell) \Rightarrow o \simeq_\ell^{ID} o_\ell \tag{9}$$

If $(t, i, p, q) \Downarrow o'$, then $\forall \ell, o \sim_\ell^{ID} o'$.

Proof. Similar to Lemma TR 14. By coinduction on the derivation of $(t, i, p, q) \Downarrow o'$. □

The next theorem establishes that the monitor is transparent for ID-secure runs. In particular, the interleaving of events from different security levels is not altered ($\sim_\ell^{ID}$).

Theorem 22 (Transparency for ID-secure runs)**.** *Let $t : React$ and i an input colist such that $(t, i) \Rightarrow o$ and $monitor(t, i) \Downarrow o'$. Then,*

$$i \text{ is ID-secure for } t \quad \Longrightarrow \quad ok(o') \ \wedge \ \forall \ell.\, o \sim_\ell^{ID} o'.$$

Proof. From Theorem 21, we know that $ok(o')$. To prove $\forall \ell.\, o \sim_\ell^{ID} o'$, applying Lemma TR 15, it is enough to show that

$$\forall \ell, (prd_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq_\ell^{ID} o_\ell$$

This property is true by Lemma TR 10. ☐

When considering CP-security, we can guarantee that the monitor will not raise an alarm and be transparent for CP-secure inputs. However, a CP-insecure input may cause the monitor to diverge without raising an alarm, as the monitor cannot predict if another visible event will be found.

Theorem 23 (CP-precise monitor)**.** *Let $t : React$ and i an input colist such that $(t, i) \Rightarrow o$ and $monitor(t, i) \Downarrow o'$. Then,*

$$i \text{ is CP-secure for } t \quad \Longrightarrow \quad ok(o') \ \wedge \ \forall \ell\, o \sim_\ell^{CP} o'.$$

Proof.

We proceed analougsly to Theorem 21 and Theorem22, taking

$$\forall \ell\, (prd_\ell(t), i_{\blacktriangleright \ell}) \rightrightarrows o_\ell \Rightarrow o \simeq_\ell^{CP} o_\ell \tag{10}$$

as an invariant in the monitor evaluation; and then proving that this invariant is a sufficient condition to ensure $ok(o')$ [4] $\forall \ell\, o \sim_\ell^{CP} o'$ [5], provided that i is CP-secure for t.

Note that (10) is valid from Lemma TR 11. ☐

7 Related Work

Precise dynamic enforcement of non-interference A series of work characterises the security policies enforceable by execution monitoring [18, 25, 39]. As a result of that, it is known that non-interference is not a safety property (see [29, 39] for a proof), and therefore not enforceable by execution monitors. The main argument for that claim relies on the fact that non-interference relates a pair of execution traces, while safety properties refer to a single one. Despite being inherently imprecise, researchers propose execution monitors to

[4] The proof proceed as in Lemma TR 14, ($\Longrightarrow$) part
[5] The proof proceed as in Lemma TR 15

enforce, in the shape of a safety property, a stronger version of non-interference (e.g. [2, 4, 5, 34, 37].) Although these monitors stop the execution of potentially dangerous programs, they still reject some non-interferent ones. Motivated by theorem proving techniques, Darvas, Hähnle, and Sands [15] show how to cast non-interference into a safety property by composing programs with a copy of themselves. This technique is known as self-composition (term coined in [8]) and it has been used to exploit known techniques for program verification [8, 45]. It is an open question if some sort of self-composition could precisely, and dynamically, detect when programs violate confidentiality. This work shows that it is possible to precisely, and dynamically, detect when the notion of non-interference ID-security [11] gets violated.

Secure multi-execution The closest related work to ours is secure multi-execution. From a systems perspective, Capizzi et al. [12] describe the idea of secure multi-execution for two security levels using the term *shadow executions*. Similarly to this work, the authors do not cover timing covert channels. However, they do not give any transparency guarantees for secure programs when using shadow executions. Similarly to Capizzi et al., but looking to obtain a secure Linux kernel, Cristiá and Mata [14] consider similar ideas as secure multi-execution using two security levels and ignoring timing channels. Differently from this work, their method is formalised for a specific programming language. Devriese and Piessens [17] introduce secure multi-execution. In that work, authors evaluate the practicality of their ideas in the Google Chrome web browser. Our work, instead, focus on theoretical results. Barthe et al. [7] show how to achieve secure multi-exection by code transformation. While their transformation is defined over an specific programming language, ours in Section 3 is described for interaction trees and thus more general. Under the same scheduler and security condition as Devriese and Piessens, Barthe et al. prove the soundness and precision of the transformed code. Focusing only on implementation issues, Jaskelioff and Russo [23] provide secure multi-execution for Haskell programs via a library. In that work, the authors propose a pure description of the I/O operations of programs that influenced the adoption of interaction trees in this paper. In order to deal with timing leaks, authors in [7, 17] require a total order of the lattice and choose a specific scheduler. Instead, authors in [24] describe a range of schedulers capable of preventing timing leaks which depends on the comparability of the elements in the lattice. In this work, we ignore the external timing covert channel for the sake of simplicity and precision of our enforcement. However, due to the modularity of our approach, we could easily apply practical black-box techniques [3, 50] to mitigate timing leaks. Bielova et al. [9] adapt secure multi-execution for web browsers. Similarly to this work, they consider a notion of secure runs for which they can provide transparency guarantees, i.e., that the behaviour of those runs is not altered by secure multi-execution. None of the works described above [7, 9, 12, 14, 17, 24] can preserve the interleaving of events generated at different security levels as well as report when insecurities occur. Instead, at the price of not considering timing covert channels, our work describes a scheduling strategy capable of preserving such order and detecting insecure actions violating ID-security. Focusing on extending secure multi-execution, and independent of this work, Rafnsson and Sabelfeld [33] propose a scheduler that is also able to preserve the order of events. Their work lifts the totality assumptions on input channels, i.e. that inputs are always available, and introduce means for declassification. Our monitor might be able to benefit from these results since it uses secure-multi execution underneath.

Faceted values Focusing on gaining performance, Austin and Flanagan [6] proposed a semantics based on faceted values that simulates multiple executions in one run. Differently from this work, execution with faceted values requires a full description of the underlying programming language semantics. They provide no formal guarantees that the interleaving of output events at different security levels is preserved. Similarly to secure multi-execution, this approach is not capable of detecting when insecurities occur during the execution of programs.

Non-interference for reactive systems Bohannon et al. [11] define several non-interference notions for reactive systems including ID- and CP-security. While CP-security provides stronger guarantees, it is more difficult to enforce by information-flow techniques. As noticed by Rafnsson and Sabelfeld [32], ID and CP-security are termination-insensitive, making possible to leak secret values by brute force attacks. Modern information-flow tools like Jif [30] (based on Java), SPARK Examiner [13] (based on Ada), and the sequential version of LIO [42] (based on Haskell) are not strong enough to avoid these leaks. For deterministic systems like the ones we consider, the bandwidth of leaking information by exploiting outputs in combination with termination is logarithmic in the size of the secret, i.e. it takes exponential time in the size of the secret to leak its whole value [1]. Nevertheless, the bandwidth can be reduced by applying buffering techniques [32]. We could easily adapt our multi-execution monitor to do that.

Interaction trees The interaction trees used to model reactive systems in this work are based on the coalgebraic view of systems [21]. Swierstra and Altenkirch [43, 44] use interaction trees to provide a functional model of some of the features of Haskell's I/O monad such as mutable state, interactive programming and concurrency. Differently from us, they do not consider reactive systems.

8 Summary

We propose multi-execution monitoring, a novel technique combining execution monitoring and secure multi-execution. This technique precisely detects actions that reveal information under the notion of ID-security. Consequently, we keep alarms to the minimum. We also prove that the monitor provides good transparency guarantees for the progress-sensitive non-interference notion of CP-security. To achieve these results, we rely on a scheduling strategy for secure multi-execution which allows us to preserve the interleaving of events, and the notions of ID-secure and CP-secure inputs.

Having the foundations for our multi-execution monitor, we can take our work into several future directions. Interaction trees can be easily adapted to model interactive programs [43, 44] (by adding one constructor *Stop* and modifying the *Read* constructor to let the program, instead of the environment, choose the channel.) However, interactive programs would require the modification of the secure multi-execution mechanism in order to consider default values for producers reading data from higher security levels. Modelling non-determinism is another interesting direction to explore. To model non-determinism, instead of considering one interaction with the environment at the time, one could consider finite sets of them [21]. This modification demands a change in the notion of similarity depending on attackers' power [31, 41, 47]. Clearly, this extension requires to extend secure multi-execution to account for non-determinism while being permissive, which is an open challenge. Declassification [38], or intended release of information, is a desirable feature of any practical information-flow system. Taking the declassification policy of delimited release [36], Askarov and Sabelfeld [2] show techniques to dynamically enforce it. We believe that our monitor could enforce declassification permissively. For that, we would extend the interaction tree model with a special constructor indicating a declassification action. Every time that the original program reads a secret data that would eventually be declassified, our monitor forwards it to the public producers. Then, when reaching a declassification point, the original program and the public producers need to be sync: they should both release the same values; otherwise, the original program might be releasing more information than expected.

References

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08. Springer-Verlag, 2008.

[2] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, Washington, DC, USA, 2009. IEEE Computer Society.

[3] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM conference on Computer and Communications Security*. ACM, 2010.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow

analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10. ACM, 2010.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12. ACM, 2012.

[7] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, June 2012.

[8] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04. IEEE Computer Society, 2004.

[9] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011),*, Sept. 2011.

[10] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10. USENIX Association, 2010.

[11] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09. ACM, 2009.

[12] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08. IEEE Computer Society, 2008.

[13] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. In *Proc. of the 2004 annual ACM SIGAda international conference on Ada*. ACM, 2004.

[14] M. Cristiá and P. Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *Workshop de Seguridad Informática WSEGI 2009, Argentina*, 38 JAIIO, 2009.

[15] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05. Springer-Verlag, 2005.

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

[17] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[18] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes

for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, Jan. 2006.

[19] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Sec. Foundations Symposium.* IEEE Computer Society, 2012.

[20] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06. ACM, 2006.

[21] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[22] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10. ACM, 2010.

[23] M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.

[24] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy.* IEEE, 2011.

[25] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, Feb. 2005.

[26] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[27] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10. ACM, 2010.

[28] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the International Information Security Conference*, SEC, 2010.

[29] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, SP '94. IEEE Computer Society, 1994.

[30] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[31] W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing interactive programs. In *Proc. IEEE Computer Sec. Foundations Symposium.* IEEE Computer Society, 2012.

[32] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based

communication. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11. ACM, 2011.

[33] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. Submitted, Feb. 2013.

[34] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, July 2009.

[35] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[36] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[37] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[38] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 255–269, June 2005.

[39] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 2000.

[40] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

[41] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[42] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[43] W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.

[44] W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 25–36, 2007.

[45] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th international conference on Static Analysis*, SAS'05. Springer-Verlag, 2005.

[46] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.

[47] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent

language. *J. Computer Security*, 7(2–3), Nov. 1999.

[48] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[49] Williams and D. Wichers. OWASP Top 10 2010. `http://www.owasp.org/index.php/Top_10_2010`, 2010.

[50] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM conference on Computer and Communications Security*. ACM, 2011.

A Interaction trees for JavaScript-like Programs

We show how to obtain the corresponding interaction tree for programs in the JavaScript-like language of Figure 2. We assume that every handler uses variable x to store the message received, and we extend the set of commands $\mathbb{C}$ with an special symbol $\cent$. Making these assumptions, we can reinterpret p as mapping from channels to the set $\mathbb{C} \cup \{\cent\}$ as follows.

$$\begin{array}{lcl} (.)_m & = & \lambda ch.\, \cent \\ (ch(x)\{c\};p)_m & = & \lambda ch'.\,(\mathbf{if}\ ch = ch'\ \mathbf{then}\ c\ \mathbf{else}\ (p)_m) \end{array}$$

The definition above simplifies our transformation, since we can treat uniformly all input events. We note $(p)_m$ as p_m for simplicity.

A state s is a tuple (p_m, μ, c), where

- p_m is a mapping from channels to $\mathbb{C} \cup \{\cent\}$ (the re-interpretation of a reactive program p as a mapping);
- μ is a mapping from variables to values, the memory;
- $c \in \mathbb{C} \cup \{\cent\}$ is the command being executed by the machine in response to an input event, or $\cent$ if the system is ready to receive an input.

We name *State* the set of states for our JavaScript-like reactive program. Given a state $s = (p_m, \mu, c)$, we can compute the next interaction of s with the environment. An interaction is an element in the set $Signal = \{\downarrow, \uparrow^{ch}_{v}, \odot\}$. Interaction $\downarrow$ is raised if s is a consumer state, i.e. if $c = \cent$. Signal $\uparrow^{ch}_{v}$ is raised when c will produce an output message (ch, v). Otherwise, the next step of s is silent, represented by signal $\odot$. Figure 6 defines a function $step : State \to Signal \times State$. We write $s \xrightarrow{i} s'$ for $step(s) = (i, s')$.

The definition of *step* makes use of some auxiliary functions. The first one is an evaluation function $\Downarrow$, such that $(\mu, e) \Downarrow v$ iff expression e evaluates to value v in memory μ. We assume that $\Downarrow$ is a side-effect free function. Second, function parse takes an string s and returns the command denoted by it if successfully parsed. For simplicity, if there is an error when parsing s, we assume parse returns `skip`. The rules are self-explanatory and therefore we do not discuss them.

$$\frac{}{(p_m, \mu, ¢) \xrightarrow{\downarrow} (p_m, \mu, ¢)} \qquad \frac{}{(p_m, \mu, \texttt{skip}) \xrightarrow{\odot} (p_m, \mu, ¢)}$$

$$\frac{(\mu, e) \Downarrow v}{(p_m, \mu, x \texttt{ := } e) \xrightarrow{\odot} (p_m, \mu[x \mapsto v], ¢)}$$

$$\frac{}{(p_m, \mu, \texttt{new } ch(x)\{c\}) \xrightarrow{\odot} (p_m[ch \mapsto c], \mu, ¢)}$$

$$\frac{(\mu, e) \Downarrow v}{(p_m, \mu, \texttt{out(}ch\texttt{,}e\texttt{)}) \xrightarrow{\uparrow_v^{ch}} (p_m, \mu, ¢)} \qquad \frac{(\mu, e) \Downarrow 0}{(p_m, \mu, \texttt{if } e\ \{c\}\{c'\}) \xrightarrow{\odot} (p_m, \mu, c')}$$

$$\frac{(\mu, e) \Downarrow v \qquad v \neq 0}{(p_m, \mu, \texttt{if } e\ \{c\}\{c'\}) \xrightarrow{\odot} (p_m, \mu, c)} \qquad \frac{(\mu, e) \Downarrow 0}{(p_m, \mu, \texttt{while } e \texttt{ do } c) \xrightarrow{\odot} (p_m, \mu, ¢)}$$

$$\frac{(\mu, e) \Downarrow v \qquad v \neq 0}{(p_m, \mu, \texttt{while } e \texttt{ do } c) \xrightarrow{\odot} (p_m, \mu, c\texttt{;}\ \texttt{while } e \texttt{ do } c)}$$

$$\frac{(p_m, \mu, c) \xrightarrow{i} (p'_m, \mu', ¢)}{(p_m, \mu, c\texttt{;}\ c') \xrightarrow{i} (p'_m, \mu', c')} \qquad \frac{(p_m, \mu, c) \xrightarrow{i} (p'_m, \mu', c_0) \qquad c_0 \neq ¢}{(p_m, \mu, c\texttt{;}\ c') \xrightarrow{i} (p'_m, \mu', c_0\texttt{;}\ c')}$$

$$\frac{}{(p_m, \mu, \texttt{eval}(s)) \xrightarrow{\odot} (p_m, \mu, \mathrm{parse}(s))}$$

Figure 6: Next interaction for a reactive state

Given a state $s = (p_m, \mu, c)$, and using function *step*, Figure 7 shows how to map interactions from s into interaction trees. The effects on the environment (i.e. $\odot$, $\downarrow$, and $\uparrow_v^{ch}$) are simply mapped into the constructors of interaction trees. The interesting case is the one related to input events. When processing input events, the interaction tree describes that, when an event arrives, the corresponding event handler is invoked ($p_m(ch)$), and variable x gets updated with the received value.

We now define function $[\![-]\!]$ from Section 2.2 as follows.

$$\begin{array}{lcl} [\![-]\!] & : & Prg \times M \to React \\ [\![p]\!](\mu) & = & [\![(p_m, \mu, ¢)]\!]_{st} \end{array}$$

B Scheduler Independent Transparency for Secure Multi-Execution

In Sect. 3 and Sect. 4, we provide security (Proposition 7) and transparency (Theorem 9) for SME under the *ops* scheduler. These results, however, can be generalized for arbitrary schedulers when the interleaving of events from

$$
\begin{array}{l}
[\![-]\!]_{st} : \mathit{State} \to \mathit{React} \\
[\![s]\!]_{st} = \textbf{let } (i, (p_m, \mu, c)) = \mathit{step}(s) \textbf{ in} \\
\quad \textbf{case } i \textbf{ of} \\
\qquad \odot \quad \to \mathit{Step}\ ([\![(p_m, \mu, c)]\!]_{st}) \\
\qquad \uparrow^{ch}_{v} \to \mathit{Write}\ (ch, v, [\![(p_m, \mu, c)]\!]_{st}) \\
\qquad \downarrow \quad \to \mathit{Read}\ (\lambda(ch, v).\, [\![(p_m, \mu[x \mapsto v], p_m(ch))]\!]_{st})
\end{array}
$$

Figure 7: Generation of interaction trees for Javascript-like programs

different security levels is not relevant.

The key observation to achieve the generalization is to be aware of schedulers which might produce leaks depending when inputs are consumed. To illustrate this point, we assume the (malicious) scheduler $s_m = [\star, L, \star, \ldots, \star, \ldots]$, i.e., an scheduler which runs the receiver, then the public producer and then infinitly often the receiver. Assume the program p:

```
p
 read H(x) skip

 read L(x) write(L, x)
```

By inspecting the code, this program seems secure. However, when run under secure multi-execution with the s_m scheduler, it leaks information. More specifically, let us take the inputs $i = [(H, 1), (L, 1)]$ and $i' = [(L, 1)]$, where $i \sim_L^{ID} i'$. Then,

$$(sme(s_m, p), i) \Rightarrow [\bullet, (L, 1)]$$

and

$$(sme(s_m, p), i') \Rightarrow [\bullet, \bullet]$$

. where $[\bullet, \bullet] \not\sim_L^{ID} [\bullet, (L, 1)]$. The leak is produced by the scheduler which exploits the information regarding the amount of inputs in its policy $[\star, L, \star, \ldots, \star, \ldots]$. To avoid such leaks, we make schedulers not capable to distinguish when programs are done consuming inputs. For that, we redefine the evaluation relation to diverge when no inputs are present.

Definition TR 16 (New semantics for *React*)**.** The evaluation relation $\Rrightarrow \subseteq (\mathit{React} \times \mathcal{I}) \times \mathcal{O}$ is coinductively defined by the folowing rules, where we write $(t, i) \Rrightarrow o$ for $(t, i, o) \in \Rrightarrow$.

$$
(R_1')\frac{(\mathit{Read}\ f, []) \Rrightarrow o}{(\mathit{Read}\ f, []) \Rrightarrow \bullet :: o}
\qquad
(R_2')\frac{(f(e), i) \Rrightarrow o}{(\mathit{Read}\ f, e :: i) \Rrightarrow \bullet :: o}
$$

$$
(W')\frac{(t, i) \Rrightarrow o}{(\mathit{Write}\ (e, t), i) \Rrightarrow e :: o}
\qquad
(S')\frac{(t, i) \Rrightarrow o}{(\mathit{Step}\ t, i) \Rrightarrow \bullet :: o}
$$

Definition 16 ensures that t will produce a stream with any input i. It is easy to show the following lemmas, relating $\Rightarrow$ and $\Rrightarrow$.

Lemma TR 17. *Let t an interaction tree and i an input colist such that $(t, i) \Rightarrow o$ and $(t, i) \Rrightarrow o'$. If $fin(o)$ then $o' \equiv o +\!\!+ \perp$, and if $inf(o)$ then $o \equiv o'$.*

Proof. By coinduction on the derivation of $(t, i) \Rightarrow o$. $\square$

Lemma TR 18. *Let t an interaction tree and i an input colist such that $(t, i) \Rightarrow o$ and $(t, i) \Rrightarrow o'$. Then $\forall \ell, o \sim_{\ell}^{CP} o'$.*

Proof. Immediate from Lemma TR 17. $\square$

Replacing $\Rightarrow$ by $\Rrightarrow$ in the definition of ID-security, we obtain a new notion of security for reactive systems.

Definition TR 19. An interaction tree t is termination-ignoring ID-secure iff, for every level ℓ and input colists i, i' such that $(t, i) \Rrightarrow o$ and $(t, i') \Rrightarrow o'$, $i \sim_{\ell}^{ID} i'$ implies $o \sim_{\ell}^{ID} o'$.

Note that if t is ID-secure, then t is termination-ignoring ID-secure. The following proposition generalize Proposition 7 to an arbitrary scheduler.

Proposition TR 20. *Let $t \in React$ and $s \in Sch$. Then, the interaction tree $sme\ (s, t)$ is termination-ignoring ID-secure.*

We prove that *sme* preserves the semantics of ID-secure programs when the interleaving of events at different security levels is not relevant.

Theorem 24 (ID-Level-by-level semantic preservation for secure inputs)**.** *Let t be an interaction tree, i an input colist ID-secure for t, and o an output colist. For any scheduler s, if $(t, i) \Rrightarrow o$, $(sme(s, t), i) \Rrightarrow o'$ then $o \simeq^{ID} o'$.*

Since every input is secure for non-interferent programs, we have the following corollary.

Corollary 25 (ID-level-by-level semantics preservation)**.** *Let t be a ID-secure interaction tree, i an input colist, and o and output colists. For any scheduler s, if $(t, i) \Rrightarrow o$, $(sme(s, t), i) \Rrightarrow o'$ then $o \simeq^{ID} o'$.*

Proof. Since t is an ID-secure interaction tree, by lemma 16, i is ID-secure for t, and we can apply theorem 24. $\square$