2014 IEEE 27th Computer Security Foundations Symposium

Compositional Information-flow Security for Interactive Systems

Willard Rafnsson Andrei Sabelfeld
Chalmers University of Technology, Gothenburg, Sweden

***Abstract*—To achieve end-to-end security in a system built from parts, it is important to ensure that the composition of secure components is itself secure. This work investigates the compositionality of two popular conditions of possibilistic noninterference. The first condition, progress-insensitive noninterference (PINI), is the security condition enforced by practical tools like JSFlow, Paragon, sequential LIO, Jif, FlowCaml, and SPARK Examiner. We show that this condition is not preserved under fair parallel composition: composing a PINI system fairly with another PINI system can yield an insecure system. We explore constraints that allow recovering compositionality for PINI. Further, we develop a theory of compositional reasoning. In contrast to PINI, we show what PSNI behaves well under composition, with and without fairness assumptions. Our work is performed within a general framework for nondeterministic interactive systems.**

I. Introduction

Modularity and compositionality are essential for the design and construction of modern computing systems. A major challenge is *secure composition*: to achieve end-to-end security in a system built from parts, it is important to ensure that the composition of secure components is itself secure. Secure composition is particularly intricate because security conditions are often fragile under system behavior modifications. Adding, removing, or modifying a single trace or event can break the security of a system [32].

This paper studies the foundations of secure composition. Our focus is on specifying *confidentiality* (or dual flavors of integrity [9], [10]) by defining what constitutes secure *information flow* through computing systems.

a) State of the art in security for interactive systems: Given the importance of the subject, it is not surprising that literature has explored security of communicating systems and their composition in assorted settings, discussed in detail in Section VI. Unfortunately, the models underlying previous frameworks on formalising and enforcing secure information flow [4], [12], [21], [34], [37], [42], [51] are all very different, and largely lacking relative comparison. Models range from memory-to-memory transformers, memory-to-output-stream transformers, input-to-output-stream transformers, and input-output-trace emitters. The frameworks make different fundamental assumptions, e.g. determinism, synchronous communication, environment totality, and fixed interaction pattern, leading to different notions of observation and environment.

For example, security has been addressed in *reactive* systems (e.g., [2], [12], [43]). The reactive system models considered exercise a restrictive pattern of communication, where the system waits for an input, and once an input has been received it proceeds with executing an appropriate handler until completion, possibly producing some output on the way. *Event* systems have been a focus of several previous approaches (e.g., [28], [29], [54], [57]). Events in the early work have different levels of sensitivity with the goal of protecting both presence and content of events but not distinguishing between the two. Interaction patterns in some of this work are fixed; Wittbold and Johnson [54] assume the pattern of receiving a secret and a public event, followed by sending a secret and public event. With *process calculi* as the underlying models, a line of work (e.g., [19], [22], [23], [40], [46], [47]) studies secure interaction, inheriting concrete features from the process calculus and not distinguishing between the sensitivity of presence and content of events. More general, and closer to our work, are formalizations that operate on *labeled transition systems* [16], [37], [42]. However, these model environments as strategies, which are separate from the computational model. Strategies are total, i.e., can always receive and send messages.

Thus, key questions that have not been addressed by previous work are: *(i) what is an appropriate general model of security of interactive systems*, *(ii) how to distinguish sensitivity of message presence and content in such a model*, *(iii) how do we model environments as part of the system*, and *(iv) how do we provide flexible ways for composing systems.*

b) Progress-sensitive and progress-insensitive security: The focus of our work is two popular security conditions. The first condition, *progress-insensitive noninterference (*PINI*)* [3], [4], [12], prevents information leaks from secret sources to public sinks, but allows secrets to affect progress of public computation. Thanks to the liberty it provides for handling loop constructs, this condition is a popular target for such practical security tools as JSFlow [21], Paragon [14], sequential LIO [51], Jif [34], FlowCaml [50], and SPARK Examiner [6]. The second condition is *progress-sensitive noninterference (*PSNI*)* [4], [12], [37], [43], which does not allow leaks via progress. The advantage of PSNI is that it provides stronger security guarantees that is not susceptible to laundering secrets by brute-force attacks [3] or re-running programs [11]. An important question that has not been answered by previous work is *(v) how do* PINI *and* PSNI *behave under composition?*

c) Contributions: Motivated by questions (i)-(v), this paper delivers the following contributions. The first contribution is a *general security framework* to model security of interactive systems. Given the diversity of previous work, our goal is to avoid "another information-flow model" with no relation to previous approaches. This motivates us to *systematize and generalize* the work in the area so far. We obtain full generality by adopting labeled transition systems as the underlying model. Our asynchronous input-output-stream emitters (Definition II.6) can model nondeterminism, non-blocking input, and arbitrary interaction patterns. In contrast

DOI 10.1109/CSF.2014.27

to previous work, environments are *interactive systems*, a significant generalization of previous work. The expressiveness of the environment model is key for compositionality results. Unifying the assumptions on environments and systems provides us with a generic system model. More importantly, it also paves the way for secure composition thanks to the possibility that systems and environments can be manipulated interchangeably. The second contribution is *combinators* for composing systems. The combinators allow flexibility in how exactly composed components can interact with each other. This is in contrast to previous work where composition is typically restricted to a single way. The third contribution allows us further generality in the modeling of interaction: we distinguish between the sensitivity of message *presence* and message *content* without restricting communication paradigms. The fourth contribution is the study of *compositionality for* PINI *and* PSNI. We find that PINI is not preserved under fair parallel composition: composing a PINI system fairly with another PINI system can yield an insecure system, and thus, compositionality of PINI relies fundamentally on unfairness. We explore constraints that allow recovering compositionality for PINI. Further, we develop a theory of compositional reasoning. In contrast to PINI, we show what PSNI behaves well under composition, with and without fairness assumptions.

d) Organization: The rest of the paper proceeds as follows. Section II presents the general setting of interactive systems, as specified by labeled transition systems. Section III presents the security definitions. Section IV establishes compositionality properties for a core of combinators, with and without fairness assumptions. Section V demonstrates the generality of our results by providing a rich language of secure combinators. Section VI reports on related work. Section VII offers conclusions and points to worthwhile future work.

II. Interactive Systems

We present a language-independent framework for reasoning about the behavior of interactive programs. The framework functions as the foundation for our technical contribution, and unifies several previous frameworks for interactive program security [12], [16], [37], [42], [43].

A. Computation Model

Our model of computation is a *labeled transition system* (LTS). An LTS is a triple $(\mathbb{P}, \mathbb{A}, \rightarrow)$; $\mathbb{P}$ is a set (of *processes*), ranged by p, $\mathbb{A}$ is a set (of *action labels*), ranged by a, and $\rightarrow \subseteq \mathbb{P} \times \mathbb{A} \times \mathbb{P}$ (a *labeled transition relation*). Computation occurs in discrete steps (transitions). The label on a step is the *effect* of said step. These effects are the *only* external interface to our processes; they are "black boxes" in every other respect. $p \xrightarrow{a} p'$ iff $(p, a, p') \in \rightarrow$, and $p \xrightarrow{a}$ iff $p \xrightarrow{a} p'$ for some p'.

The processes we consider interact with their environment through channel-based message-passing. They have two kinds of effects: (message-)input, denoted i, and (message-)output, denoted o. A message, m, is a value v on a channel c.

$$a ::= i \mid o \quad i ::= ?m \quad o ::= !m \quad m ::= cv \qquad \text{(act)}$$

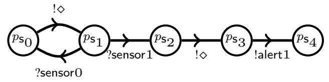
Here, $!cv$ (resp. $?cv$) denotes a message sent (resp. received) on channel c carrying value v. We let i, o, m, c and v range over $\mathbb{I}$, $\mathbb{O}$, $\mathbb{M}$, $\mathbb{C}$ and $\mathbb{V}$, respectively.

Definition II.1. An *input-output* LTS (LTS_{IO}) is an LTS with action labels ranged by a as defined in (act). ◊

This notion of computation applies to a wide range of programs and systems. For instance, Bohannon et al. give the semantics of a JavaScript-like language as an LTS_{IO} in [12], [13], and Rafnsson et al. give the semantics of an imperative language with I/O as an LTS_{IO} in [42], demonstrating that LTS_{IO} are rich enough to model anything from batch-style programs which receive one i, produce one o and terminate, to reactive systems which continuously interact. Since all of our results apply to LTS_{IO}, our contributions are general.

To illustrate, the following program reads binary sensor values until receiving 1, after which it outputs an alert message. Next to it is a graph of an LTS_{IO} describing its behavior.

```
repeat
  in sensor $b$
until $b = 1$
out alert 1
```



To model noninteraction (e.g. evaluating a loop condition), we distinguish a channel for internal actions, and denote a message on it by $\diamond$. The program receiving value 0 on channel sensor, then receiving a 1, and subsequently outputting 1 on alert, is then represented by the following transition sequence.

$$p_{\mathsf{s}_0} \xrightarrow{!\diamond} p_{\mathsf{s}_1} \xrightarrow{?\mathsf{sensor0}} p_{\mathsf{s}_0} \xrightarrow{!\diamond} p_{\mathsf{s}_1} \xrightarrow{?\mathsf{sensor1}} p_{\mathsf{s}_2} \xrightarrow{!\diamond} p_{\mathsf{s}_3} \xrightarrow{!\mathsf{alert0}} p_{\mathsf{s}_4}$$

B. Behaviors

To study the behavior of an LTS_{IO}, we consider sequences of actions performed by it.

Traces $\mathbb{T}$, ranged over by t, are (finite) lists of actions. For instance, with "ϵ" denoting the empty trace and "." the *constructor* ("cons") operator, $t_{\mathsf{s}} = !\diamond.?\mathsf{sensor1}.!\diamond.!\mathsf{alert1}.\epsilon$ is a trace. Alternatively, $\mathbb{T}$ could be defined as $\mathbb{T} = \mathbb{A}^*$, i.e., the Kleene closure of $\mathbb{A}$, or *inductively*, by asserting that $\epsilon \in \mathbb{T}$, and specifying, given an action a and a trace $t \in \mathbb{T}$, how to obtain a new trace in $\mathbb{T}$ ($a.t \in \mathbb{T}$). For simplicity, we usually omit the ϵ trailing a nonempty trace, use "." to also denote trace concatenation ($\epsilon.t = t$ and $(a.t').t = a.(t'.t)$), and write t^n for the concatenation of t n times ($t^0 = \epsilon$ and $t^{n+1} = t.t^n$). We write $t \leq t'$ when $t' = t.t''$ for some t''.

Streams $\mathbb{S}$, ranged over by s, are (strictly) infinite lists of actions ($\mathbb{S}$ exists and is unique [7]). For instance, with "." denoting the *deconstructor* operator, s_{s} for which $(.)\ s_{\mathsf{s}} = (!\diamond, \hat{s}_{\mathsf{s}})$ and $(.)\ \hat{s}_{\mathsf{s}} = (?\mathsf{sensor0}, s_{\mathsf{s}})$ is a stream, i.e., the infinite repetition of action sequence $!\diamond.?\mathsf{sensor0}$. Alternatively, $\mathbb{S}$ could be defined as $\mathbb{S} = \mathbb{A}^\omega$, i.e., the language of all infinite words over $\mathbb{A}$, or *coinductively*, by specifying, given a stream $s \in \mathbb{S}$, how to obtain an action and a new stream in $\mathbb{S}$ (apply "." on s). For simplicity, we let $a.s$ denote any stream s' for which $(.)\ s' = (a, s)$, and let $t^\omega = t.(t^\omega)$. Define $t \leq s$ and $t.s$ the same way as for traces. For more on coinduction, see [24].

The behaviors of processes are as follows. For each p, $p \xrightarrow{\epsilon} p$, and $p \xrightarrow{a.t}$ if $p \xrightarrow{a} p' \xrightarrow{t}$. Likewise, $p \xrightarrow{a.s}$ only if $p \xrightarrow{a} p' \xrightarrow{s}$. Let $\mathbb{T}(p) = \{t \mid p \xrightarrow{t}\}$ and $\mathbb{S}(p) = \{s \mid p \xrightarrow{s}\}$ be the set of traces and streams of p, i.e., the *trace- and stream-semantics* of p. For instance, $t_{\mathsf{s}} \in \mathbb{T}(p_{\mathsf{s}_0})$ and $s_{\mathsf{s}} \in \mathbb{S}(p_{\mathsf{s}_0})$, since $p_{\mathsf{s}_0} \xrightarrow{t_{\mathsf{s}}}$ and $p_{\mathsf{s}_0} \xrightarrow{s_{\mathsf{s}}}$. Traces and streams can *themselves* be regarded as LTS_{IO}, simply by defining $a.t \xrightarrow{a} t$ and $a.s \xrightarrow{a} s$.

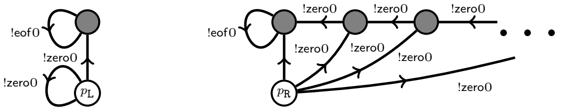
Due to the nature of PINI and due to the observation in Section III-D, we focus on stream semantics in this paper, which, under the right assumptions, are at least as expressive as trace semantics. First, the following is easy to see.

$$\forall s \,.\, s \in \mathbb{S}(p) \implies (\forall t \leq s \,.\, t \in \mathbb{T}(p)). \qquad (s\text{-all-}t)$$

However, the following does *not* hold in general.

$$\forall t \,.\, t \in \mathbb{T}(p) \implies (\exists t \leq s \,.\, s \in \mathbb{S}(p)). \qquad (t\text{-some-}s)$$

For instance, $\mathbb{S}(p_0^{\mathsf{s}}) = \{s_{\mathsf{s}}\}$ and $t_{\mathsf{s}} \not\leq s_{\mathsf{s}}$. So in general, stream semantics are less expressive than trace semantics. However, this is only the case for $\mathrm{LTS}_{\mathrm{IO}}$ with *terminal states* (e.g. p_{s_4}). For $\mathrm{LTS}_{\mathrm{IO}}$ with no terminal states, i.e., *productive* $\mathrm{LTS}_{\mathrm{IO}}$ (formalized in Definition VI.1), we have (t-some-s). However, even if we assume $\mathrm{LTS}_{\mathrm{IO}}$ are productive, trace- and stream-semantics are not equally expressive; the reverse implication of (s-all-t) does *not* hold in general. Consider these $\mathrm{LTS}_{\mathrm{IO}}$.



Here, $\mathbb{T}(p_{\mathrm{L}}) = \mathbb{T}(p_{\mathrm{R}})$. However, $\mathbb{S}(p_{\mathrm{L}}) \neq \mathbb{S}(p_{\mathrm{R}})$; in particular, for $s_{0\mathsf{s}} = \mathsf{!zero0}^{\omega}$, $p_{\mathrm{L}} \xrightarrow{s_{0\mathsf{s}}}$ but $p_{\mathrm{R}} \not\xrightarrow{s_{0\mathsf{s}}}$, since p_{R} always performs an arbitrary *finite* number of !zero0 actions. So there are scenarios where trace semantics is less expressive than stream semantics; trace semantics, depending on perspective, hides the eventuality of !eof0 in p_{R}, or hides its possible non-eventuality in p_{L}. However, in productive $\mathrm{LTS}_{\mathrm{IO}}$, this difference in expressiveness occurs *only* in the presence of *unbounded nondeterminism* (by König's lemma) [45], present in p_{R} as it nondeterministically picks a number n from the full range of natural numbers, in a single computation step.

We assume $\mathrm{LTS}_{\mathrm{IO}}$ are productive, wlg. since any $\mathrm{LTS}_{\mathrm{IO}}$ can be modeled as a productive $\mathrm{LTS}_{\mathrm{IO}}$, as follows. We model termination as an action by distinguishing a termination channel, and denoting any message on said channel by $\star$. The *obituary* wrapper O announces termination of the wrapped process.

$$\frac{p \not\rightarrow}{\mathsf{O}(p) \xrightarrow{!\star} p} \qquad \frac{p \xrightarrow{a} p'}{\mathsf{O}(p) \xrightarrow{a} \mathsf{O}(p')}$$

The *zombie* wrapper Z keeps terminated processes productive by enabling the $!\diamond$ action in terminal states.

$$\frac{p \not\rightarrow}{\mathsf{Z}(p) \xrightarrow{!\diamond} \mathsf{Z}(p)} \qquad \frac{p \xrightarrow{a} p'}{\mathsf{Z}(p) \xrightarrow{a} \mathsf{Z}(p')}$$

Now any property ϕ on $\mathbb{T}(p)$ can be stated as a property ϕ' on $\mathbb{S}(\,\mathsf{Z}(\mathsf{O}(p))\,)$ such that $\phi(\mathbb{T}(p))$ iff $\phi'(\mathbb{S}(\,\mathsf{Z}(\mathsf{O}(p))\,))$, since $\mathbb{T}(p) = \{t \mid (\,\exists s \in \mathbb{S}(\,\mathsf{Z}(\mathsf{O}(p))\,) \,.\, t \leq s\,) \wedge (\forall t' \,.\, t'.!\star \not\leq t\,)\}$.

C. Comparing Behaviors

To reason about the behavior of processes, we need ways to compare behaviors. Comparing traces is easily done component-wise, and a proof of trace equality is inductive (thus finite). What about infinite streams? One way is to define stream *equivalence* as non-difference (nonexistence of a derivation in an inductively defined difference relation). We instead use a coinductive definition which captures the idea of component-wise trace equality (note that $\forall s \,.\, \exists! a, s' \,.\, s \xrightarrow{a} s'$).

Definition II.2. $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a *strong stream relation* iff

$$\begin{aligned} &\forall s_1, s_2 \,.\, s_1 \,\mathcal{R}\, s_2 \implies (s_1 \not\rightarrow \wedge\, s_2 \not\rightarrow) \vee \\ &\exists a, s_1', s_2' \,.\, s_1 \xrightarrow{a} s_1' \wedge s_2 \xrightarrow{a} s_2' \wedge s_1' \,\mathcal{R}\, s_2'. \end{aligned}$$

s_1 and s_2 are *strongly stream related*, $s_1 = s_2$, iff there exists a strong stream relation $\mathcal{R}$ such that $s_1 \,\mathcal{R}\, s_2$. ◊

This is a strong bisimulation on streams, and thus an equivalence relation, stipulating component-wise equality. Roughly, the inductive definition of $t_1 = t_2$ is Definition II.2 with the implication reversed ($t_1 = t_2$ if $t_1 \not\rightarrow \wedge t_2 \not\rightarrow$ or $t_1 \xrightarrow{a} t_1' \wedge t_2 \xrightarrow{a} t_2' \wedge t_1' = t_2'$). For $s_{0\mathsf{e}} = \mathsf{!zero0.!eof0}^{\omega}$, $s_{0\mathsf{s}} \neq s_{0\mathsf{e}}$. An inductive proof of difference would use the fact that $\mathsf{!zero0.!zero0} \leq s_{0\mathsf{s}}$ and $\mathsf{!zero0.!zero0} \not\leq s_{0\mathsf{e}}$; this leads us to the following useful observation, proven by Park in [39].

$$s_1 = s_2 \quad \text{iff} \quad \forall t \,.\, t \leq s_1 \iff t \leq s_2. \qquad (=\text{-and-}t\text{s})$$

Ultimately, we need to compare *observable* behavior to reason about information-flow security of processes. We define observables in Section III; for now, let $\bullet$ range over unobservable actions. Furthermore, we let $\bullet$ function as a wildcard (multiple occurrences of $\bullet$ in the same context can represent different mathematical objects). *Observational equivalence* then becomes component-wise equality of observables. To obtain the next observable action, we use the following "weak" labeled transition relation: $p \overset{a}{\twoheadrightarrow} p'$ iff $a \neq \bullet \wedge (p \xrightarrow{a} p' \vee (p \xrightarrow{\bullet} \hat{p} \wedge \hat{p} \overset{a}{\twoheadrightarrow} p'))$.

Definition II.3. $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a *weak stream relation* iff

$$\begin{aligned} &\forall s_1, s_2 \,.\, s_1 \,\mathcal{R}\, s_2 \implies (s_1 \not\twoheadrightarrow \wedge\, s_2 \not\twoheadrightarrow) \vee \\ &\exists a, s_1', s_2' \,.\, s_1 \overset{a}{\twoheadrightarrow} s_1' \wedge s_2 \overset{a}{\twoheadrightarrow} s_2' \wedge s_1' \,\mathcal{R}\, s_2'. \end{aligned}$$

s_1 and s_2 are *weakly stream related*, $s_1 \simeq s_2$, iff there exists a weak stream relation $\mathcal{R}$ such that $s_1 \,\mathcal{R}\, s_2$. ◊

This is CP-similarity [12]; a weak bisimulation on streams, and thus an equivalence relation. For instance, $s_{\mathsf{s}} \simeq \mathsf{?sensor0}^{\omega}$ (with $!\diamond$ unobservable). Again, the inductive definition of $t_1 \simeq t_2$ is Definition II.3 with reversed implication. Let $t \preceq t'$ when $t \simeq t''$ for some $t'' \leq t'$. Define $t_1 \preceq s_2$ similarly. Note that $s_1 \not\simeq s_2$ does *not* necessarily have a proof by induction on its derivation; this is the case when the observables of s_1 strictly prefix the observables in s_2; for some t, $s_1 \overset{t}{\twoheadrightarrow} \bullet^{\omega}$, $s_2 \overset{t}{\twoheadrightarrow} s_2'$, $s_2' \overset{a}{\twoheadrightarrow}$ and $\bullet^{\omega} \not\twoheadrightarrow$, but $\bullet^{\omega} \not\twoheadrightarrow$ is an assertion about all elements in an infinite sequence. A proof of $t \preceq s$ *is* inductive, since t is finite. The following analog of (=-and-ts) is thus of interest.

$$s_1 \simeq s_2 \quad \text{iff} \quad \forall t \,.\, t \preceq s_1 \iff t \preceq s_2. \qquad (\simeq\text{-and-}t\text{s})$$

The other means we consider of comparing observables stipulates that there is never a disagreement in which observable comes next in two streams. That is, component-wise comparison of observables does not eventually yield two observables which differ. This holds e.g. when $s_1 \simeq s_2$ or when the observables in s_1 prefix the observables in s_2.

Definition II.4. $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a *feeble stream relation* iff

$$\begin{aligned} &\forall s_1, s_2 \,.\, s_1 \,\mathcal{R}\, s_2 \implies s_1 \not\twoheadrightarrow \vee\, s_2 \not\twoheadrightarrow \vee \\ &\exists a, s_1', s_2' \,.\, s_1 \overset{a}{\twoheadrightarrow} s_1' \wedge s_2 \overset{a}{\twoheadrightarrow} s_2' \wedge s_1' \,\mathcal{R}\, s_2' \end{aligned}$$

s_1 and s_2 are *feebly stream related*, $s_1 \approx s_2$, iff there exists a feeble stream relation $\mathcal{R}$ such that $s_1 \,\mathcal{R}\, s_2$. ◊

This is a coinductive definition of NC-similarity [12]. It is reflexive and symmetric, but not transitive, thus not an equivalence relation; $?\mathsf{sensor}0.\bullet^{\omega} \approx \bullet^{\omega}$, $!\mathsf{zero}0.\bullet^{\omega} \approx \bullet^{\omega}$, but $?\mathsf{sensor}0.\bullet^{\omega} \not\approx !\mathsf{zero}0.\bullet^{\omega}$ (indeed, for all s, $s \approx \bullet^{\omega}$). To contrast, $!\mathsf{zero}0.\bullet^{\omega} \not\preceq \bullet^{\omega}$. Again, the inductive definition of $t_1 \approx t_2$ is Definition II.4 with reversed implication. On traces, $t_1 \approx t_2$ iff $t_1 \preceq t_2$ or $t_2 \preceq t_1$. $s_1 \not\approx s_2$ *has* a proof by induction on its derivation, because eventuality of a component-wise difference can be represented as a trace t for which $t \preceq s_1$ but $t \not\preceq s_2$ (or vice versa). With $t \approx s$ defined in the obvious way ($t \approx s$ iff $t.\bullet^{\omega} \approx s$), $t \not\approx s$ also has an inductive proof.

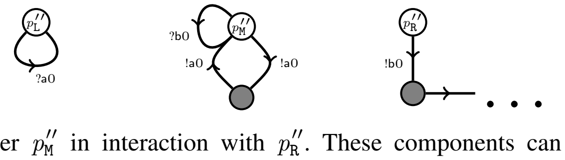
$$s_1 \approx s_2 \quad \text{iff} \quad \forall t \,\text{.}\, t \not\approx s_1 \iff t \not\approx s_2. \qquad (\approx\text{-and-}t\text{s})$$

Lemma II.5. $(=) \subsetneq (\simeq) \subsetneq (\approx)$.

We close with a justification for focusing on stream semantics as a basis for comparing the behavior or processes. As noted earlier, actions are the *only* external interface to our processes. Thus, when reasoning about security, we model attacker knowledge obtained via *extensional* observations. Here, not allowing the attacker to observe the state of the process is natural. This makes branching-time equivalence (bisimulation on *processes*) too strict for our purposes, since it makes distinctions on the *internal* branching structure of processes [35]. We therefore opt for stream equivalence: a linear-time behavioral equivalence which subsumes trace equivalence

D. Interaction

The input-output behavior in the LTS_{IO} semantics of a program implies a particular model of interaction used by said program. Typically, secure information flow frameworks in the reactive, process algebra, and interactive program setting, adopt *synchronous* communication on channels as the model of communication [12], [16], [19], [22], [37], [40], [42], [43], [47]. Here, if we take the processes in these computation models and put them in interaction, then processes can *block* on both input *and output*. We argue that synchronous communication is not a good fit for general system composition, as the exhibited blocking behavior makes (compositional) reasoning about process behavior nontrivial. Progress of synchronously interactive programs is highly context sensitive, as nonwillingness of one component to receive can halt the progress of a sender component. For instance, consider the following LTS_{IO}.



Consider $p''_{\mathtt{M}}$ in interaction with $p''_{\mathtt{R}}$. These components can synchronize b0, and under an inert environment, this synchronization *will* (eventually) occur. However, merely by adding $p''_{\mathtt{L}}$, a component which sends no messages, the eventuality of b0 is no longer guaranteed, as there is a (*fair*) scheduling which starves it: let $(p''_{\mathtt{L}}, p''_{\mathtt{M}})$ synchronize first, and then let $(p''_{\mathtt{M}}, p''_{\mathtt{R}})$ synchronize if they can; otherwise synchronize $(p''_{\mathtt{L}}, p''_{\mathtt{M}})$ twice. The same situation arises if we reverse the direction of b.

In practice, output blocking is typically avoided by buffering channels, making communication *asynchronous*. Then, $p''_{\mathtt{R}}$ would put !b0 into a buffer without delay and move on with its computation, while $p''_{\mathtt{M}}$ would subsequently read from the buffer whenever $p''_{\mathtt{M}}$ is ready to do so. However, this effect is also achieved by requiring that *there is always a receiver ready for each send*. This holds if each component in a composition is always ready to receive input, i.e., *input total* [27], [30]. Input totality simplifies reasoning about composed systems considerably [29], [57], as an input-total process cannot control its environment by not desiring certain input. Input totality abstracts from *how* asynchrony is achieved. Thus, our framework generalizes more concrete approaches e.g. input queues [52] and buffered channels [48]. To make fairness visible in an interaction stream, we ensure each component is always able to perform an action, regardless of which environment it is in. We get this by assuming that processes are *output productive*, i.e., always capable of producing output. This makes our processes similar to Input/Output Automata [27], in that processes are autonomous; a process can always make progress on its outputs. This is in contrast to asynchrony in process algebra (e.g. [23]), where output is only sent when a receiver is ready. Like in [27], we say a stream is *fair* if it contains an infinite number of outputs, i.e., is in the set

$$\mathbb{S}_{\mathsf{F}} = \{s \in \mathbb{S} \mid \forall t \leq s \,\text{.}\, \exists t', o' \,\text{.}\, t.t'.o' \leq s\}.$$

Let $\mathbb{S}_{\mathsf{F}}(p) = \mathbb{S}(p) \cap \mathbb{S}_{\mathsf{F}}$. Since $!\diamond$ models noninteraction, receivers of $!\diamond$ should not react to it. Such receivers are *atemporal*. For further justification and merits of this model of concurrency, see Section VI.

Definition II.6 (Interactive LTS_{IO})**.** p is

1) *input total* iff $\forall t, p' \,\text{.}\, p \xrightarrow{t} p' \implies \forall i \,\text{.}\, p' \xrightarrow{i}$
2) *output productive* iff $\forall t, p' \,\text{.}\, p \xrightarrow{t} p' \implies \exists o \,\text{.}\, p' \xrightarrow{o}$.
3) *atemporal* iff $\forall t, p' \,\text{.}\, p \xrightarrow{t} p' \implies \forall ?\diamond, p'' \,\text{.}\, p' \xrightarrow{?\diamond} p'' \implies p' = p''$.

p is *interactive* iff p satisfies 1), 2) and 3). ◊

Putting two interactive LTS_{IO} in nonblocking interaction is now a simple matter of making all output of one process become the only input to the other (and vice versa). We write p *under* p' as $p' \models p$. p *produces* s *under* p', $p' \models p \xrightarrow{s}$, iff, $p \xrightarrow{s}$ and $p' \xrightarrow{s^{-1}}$. Similarly for traces. Function $\cdot^{-1}$ reverses direction of messages, that is, $\epsilon^{-1} = \epsilon$, $(!m.t)^{-1} = ?m.t^{-1}$, and $(?m.t)^{-1} = !m.t^{-1}$, and similarly for streams. We refer to p' here as the *environment* of p, and we will study the behavior of p under different environments when reasoning about security of p. This is in stark contrast with previous work on security of LTS_{IO} [16], [37], [42] which considers (classes of) *strategies* as environments, i.e., functions of type $\mathbb{T} \to \mathbb{C} \to \mathcal{P}(\mathbb{V})$. One noteworthy feature which secure p-environments have over secure strategy-environments is p can force a secret input to occur before a public input, as input streams can in the reactive systems setting [12]; indeed, if p' and p are deterministic, and the interaction pattern is fixed, p' will behave like an input stream. Our framework thus unifies several previous frameworks for interactive program security.

We assume p is interactive throughout our development, unless stated otherwise. While these are strong restrictions to impose on an LTS_{IO}, any LTS_{IO} can be modeled as an interactive LTS_{IO}. For instance, for p which can discriminate on which channel to receive on next, like the $\textsc{lts}_{\text{IO}}$ in [16], [37], [42], the *buffer* wrapper $\mathsf{B}^{?}$ associates an input queue with each channel, which p can then receive on at its leisure. $\mathsf{B}^{?}(p)$ is input total, for any p.

$$\frac{p \xrightarrow{o} p'}{\mathsf{B}^{?}(t,p) \xrightarrow{o} \mathsf{B}^{?}(t,p')} \qquad \frac{p \xrightarrow{?cv} p' \qquad \nexists(t''.?cv') \leq t}{\mathsf{B}^{?}(t.?cv.t',p) \xrightarrow{!\diamond} \mathsf{B}^{?}(t.t',p')}$$

$$\frac{-}{\mathsf{B}^{?}(t,p) \xrightarrow{i} \mathsf{B}^{?}(t.i,p)} \qquad \mathsf{B}^{?}(p) = \mathsf{B}^{?}(\epsilon,p)$$

For programs which never discriminate on which channel to receive from, like the $\textsc{lts}_{\text{IO}}$ in [12], [43], [58], the *FIFO* wrapper $\mathsf{F}^{?}$ buffers input and delivers it to p on demand, in FIFO order. For such p, $\mathsf{F}^{?}(p)$ is input total.

$$\frac{p \xrightarrow{o} p'}{\mathsf{F}^{?}(t,p) \xrightarrow{o} \mathsf{F}^{?}(t,p')} \qquad \frac{p \xrightarrow{i} p'}{\mathsf{F}^{?}(i.t,p) \xrightarrow{!\diamond} \mathsf{F}^{?}(t,p')}$$

$$\frac{-}{\mathsf{F}^{?}(t,p) \xrightarrow{i} \mathsf{F}^{?}(t.i,p)} \qquad \mathsf{F}^{?}(p) = \mathsf{F}^{?}(\epsilon,p)$$

Any p which is not output productive has the potential to block on input. The *wait* wrapper W empowers any such p with the ability to, instead of block, wait as an internal action. For any p, $\mathsf{W}(p)$ is output productive.

$$\frac{\exists i \,.\, p \xrightarrow{i} \qquad \nexists o \,.\, p \xrightarrow{o}}{\mathsf{W}(p) \xrightarrow{!\diamond} \mathsf{W}(p)} \qquad \frac{p \xrightarrow{a} p'}{\mathsf{W}(p) \xrightarrow{a} \mathsf{W}(p')}$$

At last, the *atemporal* wrapper T ignores any $?\diamond$ actions. $\mathsf{T}(p)$ is atemporal, for any p.

$$\frac{-}{\mathsf{T}(p) \xrightarrow{?\diamond} \mathsf{T}(p)} \qquad \frac{p \xrightarrow{a} p' \qquad a \neq ?\diamond}{\mathsf{T}(p) \xrightarrow{a} \mathsf{T}(p')}$$

III. Security of Interactive Systems

Equipped with the tools from the previous section, we develop notions of information-flow security in our setting. We present two popular conditions of *possibilistic noninterference*: PSNI and PINI. While PSNI is well studied in our setting [16], [37], [42], we give the first formalization of PINI in a general interactive setting; PINI has thus far only been presented in settings with restricted forms of interaction [3], [12].

A. Observables

The observables of a process are its effects. We assume a lattice $(\mathcal{L}, \sqsubseteq)$, with $\mathcal{L}$ ranged by ℓ, of security levels expressing levels of *confidentiality*. Each channel c is labeled with two security levels; $\pi(c)$ is the level of the *presence* of a message on c, and $\kappa(c)$ is the level of the *content*, or value, of a message on c. In examples, we frequently represent a channel by its security levels, writing $\kappa(c)^{\pi(c)}$ in place of c. A classic example is the lattice $\mathcal{L} = \{\mathtt{L}, \mathtt{H}\}$ ("low" (public) and "high" (secret)) and $\sqsubseteq = \{(\mathtt{L},\mathtt{L}), (\mathtt{L},\mathtt{H}), (\mathtt{H},\mathtt{H})\}$. We let L, M and H denote $\mathtt{L}^{\mathtt{L}}$, $\mathtt{H}^{\mathtt{L}}$ and $\mathtt{H}^{\mathtt{H}}$, respectively. We let $\top$ resp. $\bot$ denote the top resp. bottom element in the security lattice. Let $\$ ::= ?\ |\ !$ and define $\pi(\$cv) = \pi(c)$ and $\kappa(\$cv) = \kappa(c)$. Since timing-sensitive reasoning is beyond the scope of this paper, we set $\pi(\$\diamond) = \kappa(\$\diamond) = \top$. For termination-sensitive reasoning in this framework, set $\pi(\$\star) = \kappa(\$\star) = \bot$ and impose a restriction similar to atemporal for termination actions.

The security labels express *who* can observe *what*. An observer is associated a security level ℓ. An ℓ-observer is capable of observing the presence (resp. content) of messages on c if $\pi(c) \sqsubseteq \ell$ (resp. $\kappa(c) \sqsubseteq \ell$). Let $s\upharpoonright_{\ell}$ be the stream where, component-wise, each action has been replaced with what an ℓ-observer observes in the action:

$$(\$cv.s)\upharpoonright_{\ell} = \begin{cases} \bullet.s\upharpoonright_{\ell}, \text{ if } \pi(c) \not\sqsubseteq \ell \\ \$c\mathtt{d}.s\upharpoonright_{\ell}, \text{ if } \pi(c) \sqsubseteq \ell \wedge \kappa(c) \not\sqsubseteq \ell \\ \$cv.s\upharpoonright_{\ell}, \text{ otherwise.} \end{cases}$$

For instance, $(?\mathsf{H}0.?\mathsf{M}1.!\mathsf{L}2.!\diamond^{\omega})\upharpoonright_{\mathtt{L}} = \bullet.?\mathsf{M}\mathtt{d}.!\mathsf{L}2.\bullet^{\omega}$. Here (and later), $\mathtt{d}$ is a constant. Let $s_1 \simeq_{\ell} s_2$ iff $s_1\upharpoonright_{\ell} \simeq s_2\upharpoonright_{\ell}$, and $s_1 \approx_{\ell} s_2$ iff $s_1\upharpoonright_{\ell} \approx s_2\upharpoonright_{\ell}$. Similarly for traces.

B. Noninterference

The idea behind noninterference is as follows. Assume an ℓ-observer observes all he is privileged to observe. A process is noninterfering if, based on ℓ-observables, the ℓ-observer learns nothing he is not privileged to learn, i.e., unobservable input does *not interfere* with observable behavior. Noninterfering processes are thereby not responsible for leaks (in our possibilistic setting, this means any difference in observable behavior must be attributable to nondeterministic choices).

To attribute a detected insecurity to the process under scrutiny, we study its behavior under secure environments. Typically, definitions of noninterference state that a process exhibits observably equivalent behavior, under any pair of noninterfering observably equivalent environments. In our setting, this leads to a circularity, since environments are processes. Previous work avoids this circularity by

a) using simpler environments for which noninterference and observational equivalence are trivial [3], [19], [37],
b) defining observational equivalence on processes, and noninterference as self-equivalence [12], [49], or
c) defining noninterference as invariance of observable behavior to insertion/deletion of unobservable input [25].

We find that none of these approaches can be applied directly to our setting. Since compositionality is a main concern in this paper, we need environments to be part of the computation model, ruling out a). Since self-equivalences are bisimulation relations, they are branching-time equivalences, rejecting e.g. the following program p_{linear}, since it *can* enter the "else" branch, where it can leak information, even through it can also always take the "then" branch on x, where no information leaks. We therefore find b) too strict.

```
$x = 0|1$ ; **out** H $x$ ; **poll** H $h$
**if** $h =$ UNDEFINED **then** $h = 0$ **end if**
**if** $x \bmod 2 = 1$ **then out** L $(0|1)$
**else out** L $(h \bmod 2)$ **end if**
```

Here, **poll** c x is a nonblocking input interacting with a buffering context, similar to if-receive in [48]. If a c-input is waiting in the buffer, consume it. Otherwise, write UNDEFINED to x. Formally à la [42] (with $X =$ UNDEFINED),

$$\frac{-}{\langle \mu, \textbf{poll}\ c\ x \rangle \xrightarrow{¿cv} \langle \mu[x \mapsto v], \textbf{skip} \rangle}$$

$$\frac{p \xrightarrow{¿cv} p' \qquad \nexists(t''.?cv') \leq t}{\mathsf{B}^{?}(t.?cv.t',p) \xrightarrow{!\diamond} \mathsf{B}^{?}(t.t',p')} \qquad \frac{p \xrightarrow{¿cX} p' \qquad \nexists(t'.?cv) \leq t}{\mathsf{B}^{?}(t,p) \xrightarrow{!\diamond} \mathsf{B}^{?}(t,p')}$$

Approaches based on c) are defined on traces, and have similar problems as b) in that they reject the above program; inserting $?\mathtt{H1}$ immediately after $!\diamond.!\mathtt{H0}$ in $!\diamond.!\mathtt{H0}.!\diamond^k.!\mathtt{L0}$ makes the subsequent $!\mathtt{L0}$ impossible (since $!\mathtt{H0}$ fixes x to 0).

What we desire is a property which stipulates that a process *can* (by making the appropriate nondeterministic choices) *preserve the possibility* of a sequence of observables, under insertion of unobservable input during execution. To ensure that a stream is not only possible due to the presence of unobservable input, we require the above for streams which contain no unobservable input, i.e., streams over

$$\mathbb{A}_\ell = \mathbb{O} \cup \{?cv \mid \pi(c) \sqsubseteq \ell \wedge (\kappa(c) \not\sqsubseteq \ell \implies v = \mathsf{d})\}.$$

Since a process cannot leak information if denied the opportunity to produce output, we focus on fair runs of processes. We define the above as a coinductive predicate as follows.

Definition III.1. *p $\mathcal{R}_\ell$-preserves the possibility of s_0 after t through s*, $\mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(t, s)$, is the largest predicate satisfying each of the following.

1) $\forall o \leq s \,\textbf{.}\, \exists s' \in \mathbb{A}^\omega_\ell \textbf{.}$

   $s_0 \,\mathcal{R}_\ell\, t.o.s' \quad \wedge\ p \xrightarrow{t.o.s'} \quad \wedge\, \mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(t.o, s')$

2) $\forall i \preceq_\ell s \,\textbf{.}\, \exists s' \in \mathbb{A}^\omega_\ell \textbf{.}$

   $s_0 \,\mathcal{R}_\ell\, t.i.s' \quad \wedge\ p \xrightarrow{t.i.s'} \quad \wedge\, \mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(t.i, s')$

3) $\forall \tilde{i} \simeq_\ell s \,\textbf{.}\, \exists s' \in \mathbb{A}^\omega_\ell, \bar{i} \leq \tilde{i}, o \textbf{.}$

   $s_0 \,\mathcal{R}_\ell\, t.\bar{i}.o.s' \ \wedge\ p \xrightarrow{t.\bar{i}.o.s'} \ \wedge\, \mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(t.\bar{i}.o, s')$ ◊

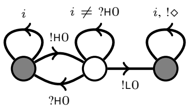
How is this definition invoked to state that p preserves the possibility of the observables in a fair stream s_0 for which $p \xrightarrow{s_0}$, under insertion of $\mathtt{H}$ input? If p is secure, then there is a fair $s \in \mathbb{A}^\omega_\ell$ for which $p \xrightarrow{s}$ and $s_0 \,\mathcal{R}_\ell\, s$. We then invoke the definition with $\mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(\epsilon, s)$.

Definition III.2. p is *$\mathcal{R}$-noninterfering* iff $\forall \ell \,\textbf{.}\, \forall s \in \mathbb{S}_\mathsf{F}(p) \textbf{.}$

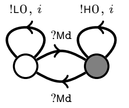
$\exists s' \in \mathbb{S}(p) \cap \mathbb{A}^\omega_\ell \,\textbf{.}\, s \,\mathcal{R}_\ell\, s' \wedge \mathsf{preserve}^{\ell,\mathcal{R}}_{p,s}(\epsilon, s').$ ◊

How does $\mathsf{preserve}^{\ell,\mathcal{R}}_{p,s_0}(\epsilon, s)$ state that p preserves the possibility of the observables in s_0? 1) corresponds to the scenario where the environment feeds no input into p; to match s_0, p merely emits s, since we already have that $s_0 \ \mathcal{R}_\ell \ s$ ($s = o.s'$). 2) states what must hold when the environment provides $\mathtt{H}$ input at any time ($?cv \preceq_\ell \hat{s}$ holds for all $\hat{s}$, v and c for which $\pi(c) \not\sqsubseteq v$) and when the next observable action is an input; p must be able to choose a $s' \in \mathbb{A}^\omega_\ell$ matching the rest of s_0. 3) states that when there are no more observable output remaining to be matched, p can be scheduled fairly while still preserving possibility of s_0; when p is being fed a stream of input, p can at some point cut into the (otherwise infinite) stream of input $\tilde{i}$ after a (finite) trace of inputs $\bar{i} \leq \tilde{i}$ and produce an unobservable output o.

When devising our properties, we encountered three challenging scenarios which our properties needed to deal with to guarantee eventuality of actions: i) *scheduling*, ii) *high interaction loops*, and iii) *high output starvation.* In i), since both p and its environment can be the producer behind the kth action in an interaction, p might only be able to choose a matching s' if p produces the next message. However, the demand p places on its possibilistic scheduling may conflict with the demands of the environment. This is the case when both p and its environment only preserve confidentiality if they each are the producer behind the first message in the interaction. Having one yield fully to the demands of the other makes the security properties stipulate security for all schedulers, which implies security under scheduler *refinement* [59]. While interesting, we find this to be too conservative; we wish for our scheduler to be possibilistic (and therefore a source of nondeterminism which can conceal information leaks). In ii), future observables can be starved as a result of either unfortunate scheduling, or $\mathtt{H}$ input. To illustrate, consider e.g. the following process p_{loop}.



After receiving $?\mathtt{H0}$, this process *insists* on outputting $\mathtt{H0}$ before outputting $\mathtt{L0}$. If the environment insists (to not leak) on outputting $\mathtt{H0}$ before receiving $\mathtt{L0}$, action $!\mathtt{L0}$ is deferred, possibly indefinitely, as is possible when p_{loop} interacts with a variant of itself which first outputs $\mathtt{H0}$. The process and its environment here are engaged in a *livelock*, exhibiting behavior reminiscent of a "hallway dance", or more accurately, an Alphonse-Gaston routine ("After you", followed by a back-and-forth "No, you first." ad infinitum) [38]. In the presence of high interaction loops, security must ensure that p can eventually produce its next observable, regardless of what environment p is run under. Dually, the environment must tolerate receiving the next observable at any point in time, as different processes under it can demand high interaction loops terminated at different times. Thus 2) additionally guarantees possible eventual termination of high interaction loops between a secure p interacting with a secure environment (at the whim of the producer of the next observable), addressing i) and ii). Finally, 3) is designed to address iii), e.g., to ensure that after p has produced all its observables, it can still be scheduled fairly. Consider this example of high output starvation, p_{starve}.



Consider environments $p_1 = \mathsf{F}^?(!\mathtt{Md}^\omega)$ and $p_2 = \mathsf{F}^?(!\mathtt{M0}^\omega)$. Then $p_1 \models p_{\text{starve}}$ can match $?\mathtt{M0}^\omega$ fairly, while $p_2 \models p_{\text{starve}}$ cannot; while $p_{\text{starve}} \xrightarrow{(?\mathtt{Md}.!\mathtt{H0})^\omega}$, the first $i \preceq_\ell (?\mathtt{Md}.!\mathtt{H0})^\omega$ fed to p_{starve} by p_2 is $i = ?\mathtt{M0}$, making the matching stream with i as first action $?\mathtt{M0}.(?\mathtt{Md}.!\mathtt{H0})^\omega$; compared to $(?\mathtt{Md}.!\mathtt{H0})^\omega$, the first output has been deferred. The only way for $p_2 \models p_{\text{starve}}$ to match $?\mathtt{M0}^\omega$ is to not produce output at all, thus starving p_{starve}. 3) rejects programs of this nature.

Our "*preservation-based noninterference*" can be viewed as a hybrid of b) and c). In contrast to b), we do not require security to hold in all reachable states. In contrast to c), we do not require that each $s \in \mathbb{S}(p)$ satisfies insert/delete conditions; only that some $s' \ \mathcal{R}_\ell \ s$ does, i.e. that p *can* make the right nondeterministic choices to preserve the possibility of $\mathcal{R}_\ell s$.

Definition III.3. $p \in \text{PSNI}$ iff p is $\simeq$-noninterfering. ◊

Definition III.4. $p \in \text{PINI}$ iff p is $\approx$-noninterfering. ◊

Definition III.4 is the first definition of PINI in a general nondeterministic interactive setting. It differs from previous PINI formalizations [3], [4], [12] in that input to the process is not fixed before the process is run; rather, the environment is permitted to adapt its input based on prior process output. Our definition can be improved, however; consider p_{echo}, a process which outputs anything it receives in FIFO order (outputs $!\diamond$ when its FIFO is empty), except when $?\mathsf{H}0$ is received; then p_{echo} immediately becomes $\mathsf{F}^?(!\diamond^\omega)$. Consider environments $p_1 = !\mathsf{L}0^\omega$ and $p_2 = !\mathsf{H}0.!\mathsf{L}0^\omega$. Then $p_1 \models p_{\text{echo}} \xrightarrow{(?\mathsf{L}0.!\mathsf{L}0)^\omega}$, which is $\approx_\ell$-matched by $p_2 \models p_{\text{echo}} \xrightarrow{?\mathsf{H}0.!\diamond^\omega}$. However, the moment $p_2 \models p_{\text{echo}}$ inputs twice after $?\mathsf{H}0$, $\approx_\ell$-equivalence with $(?\mathsf{L}0.!\mathsf{L}0)^\omega$ is lost. Thus, in general, a p satisfying Definition III.4 might *have* to starve the environment to $\approx_\ell$-match a stream. This is not unlike ID-security in [12], which allows a reactive system to ignore an observable input ready in the environment by diverging silently while reacting to a previous input. While we could adjust our definition of $\approx_\ell$ to cover the above scenario, the main reason we introduce PINI is to study how it behaves under composition, and the adjusted PINI would fail to compose in the same ways our PINI does.

PSNI is strictly stronger than PINI. This follows from Definition III.2, Lemma II.5 and this program which, wrapped in $\mathsf{W} \circ \mathsf{B}^? \circ \mathsf{Z}$, is in PINI $\setminus$ PSNI.

in $\mathsf{H}\ h$
if $(h \bmod 2) = 0$ **then out** $\mathsf{L}\ 0$ **end if**

Lemma III.5. $p \in \text{PSNI} \implies p \in \text{PINI}$

Pt. 2) in Definition III.1 enables a simple *proof technique* for guaranteeing eventuality of observable actions in an interaction: *schedule the producer of the next observable*. To see this in action, see the proofs of Theorems III.10 and IV.3.

C. Noninterference Under Environments

To facilitate evaluation of the relative merits of our preservation-based formalization of progress-(in)sensitive noninterference, and to demonstrate the generality of our framework, we give more conventional definitions of PSNI and PINI under environments à la [16], [37], [42].

Definition III.6. p_2 *$\mathcal{R}_\ell$-simulates* p_1, iff

$$\forall s_1 \in \mathbb{S}_{\mathsf{F}}(p_1) \,.\, \exists s_2 \in \mathbb{S}_{\mathsf{F}}(p_2) \,.\, s_1\ \mathcal{R}_\ell\ s_2.$$

p_1, p_2 are *$\mathcal{R}_\ell$-equivalent*, $p_1\ \mathcal{R}_\ell\ p_2$, iff, p_1 *$\mathcal{R}_\ell$-simulates* p_2 and p_2 *$\mathcal{R}_\ell$-simulates* p_1. ◊

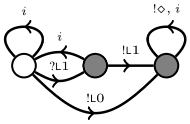
Definition III.7. p_2 *$\mathcal{R}_\ell$-E-simulates* p_1, iff

$$\begin{aligned}
&\forall p_1', p_2' \in \text{PSNI} \,.\, p_1' \simeq_\ell p_2' \implies \\
&\forall s_1 \in \mathbb{S}_{\mathsf{F}} \,.\, p_1' \models p_1 \xrightarrow{s_1} \implies \\
&\exists s_2 \in \mathbb{S}_{\mathsf{F}} \,.\, p_2' \models p_2 \xrightarrow{s_2} \wedge\, s_1\ \mathcal{R}_\ell\ s_2.
\end{aligned}$$ ◊

Definition III.8. $p \in \text{PSNI}_{\mathsf{E}}$ iff $\forall \ell \,.\, p \simeq_\ell$-$\mathsf{E}$-simulates p. ◊

Definition III.9. $p \in \text{PINI}_{\mathsf{E}}$ iff $\forall \ell \,.\, p \approx_\ell$-$\mathsf{E}$-simulates p. ◊

We use PSNI environments in the definition of PINI_{E} because otherwise, PINI_{E} becomes too conservative. To see this, say we defined PINI_{E} by relaxing $\simeq_\ell$ to $\approx_\ell$ and environment assumption PSNI to PINI in the definition of PSNI_{E}. Now consider the following process $p_{\text{PINI}_{\mathsf{E}}} \in \text{PSNI}$.



Consider environments $p_1 = \mathsf{F}^?(!\mathsf{L}1.!\diamond^\omega)$ and $p_2 = \mathsf{F}^?(!\diamond^\omega)$, for which we have $p_1, p_2 \in \text{PINI}$ and $p_1 \approx_{\mathtt{L}} p_2$. However, while $p_1 \models p_{\text{PINI}_{\mathsf{E}}} \xrightarrow{?\mathsf{L}1.!\mathsf{L}1.!\diamond^\omega}$, $p_2 \models p_{\text{PINI}_{\mathsf{E}}}$ cannot fairly either match these observables or remain silent; eventually, $p_{\text{PINI}_{\mathsf{E}}}$ performs $!\mathsf{L}0$, and so, for any stream $s \in \mathbb{S}_{\mathsf{F}}$ for which $p_2 \models p_{\text{PINI}_{\mathsf{E}}} \xrightarrow{s}$, $s \not\approx_\ell ?\mathsf{L}1.!\mathsf{L}1.!\diamond^\omega$. So, while replacing $\simeq_\ell$ in PSNI_{E} with $\approx_\ell$ weakens the security definition, relaxing the assumptions on the environments to PINI strengthens it immensely; the behavior of a process would need to be invariant to $\mathtt{L}$ input as well as $\mathtt{H}$ input to satisfy PINI_{E}.

For our preservation-based definitions to be useful on their own, they need to be stronger than the standard environment-based definitions; then we will know processes satisfying our preservation-based definitions are safe against all attacks which processes satisfying the environment-based definition are safe against. This is the case; the proof is in the appendix.

Theorem III.10. $p \in \text{PSNI} \implies p \in \text{PSNI}_{\mathsf{E}}$

Theorem III.11. $p \in \text{PINI} \implies p \in \text{PINI}_{\mathsf{E}}$

We suspect the reverse implication of these theorems to be false. To show that p satisfies the constraint imposed by Definition III.1 pt. 2) using assumption $p \in \text{PINI}_{\mathsf{E}}$, it seems we need to propose an environment which outputs a different observable if it receives an unobservable first. However, such an environment is not PSNI (resp. PINI). These theorems give us a sense of assurance, however; if a property is too weak, say, $\text{P} = \{p \in \text{LTS}_{\mathsf{IO}} \mid p \text{ is interactive}\}$, then $p \in \text{P} \centernot\implies p \in \text{P}_{\mathsf{E}}$, since P_{E} places demands on p beyond $p \in \text{P}$.

For similar reasons as for Lemma III.12, PSNI_{E} is strictly stronger than PINI_{E}.

Lemma III.12. $p \in \text{PSNI}_{\mathsf{E}} \implies p \in \text{PINI}_{\mathsf{E}}$

Finally, we consider to which extent PSNI_{E} (resp. PINI_{E}) permits processes to starve the environment to preserve confidentiality. A process starving the environment is exerting control over the possibilistic scheduling of processes, which violates our desire for processes to be autonomous. Therefore, ideally, PSNI_{E} and PINI_{E} should reject processes which might *need* to starve the environment to preserve confidentiality. We say *p produces s fairly under p'*, $p' \models_{\mathsf{F}} p \xrightarrow{s}$, iff $p' \models p \xrightarrow{s}$ and $s^{-1} \in \mathbb{S}_{\mathsf{F}}$. Now let $\text{PSNI}_{\mathsf{EF}}$ (resp. $\text{PINI}_{\mathsf{EF}}$) be defined as PSNI_{E} (resp. PINI_{E}) with "$\models$" replaced by "$\models_{\mathsf{F}}$". It turns out that $\text{PSNI}_{\mathsf{EF}}$ and PSNI_{E} are equivalent, and thus, that $p \in \text{PSNI}_{\mathsf{E}}$ preserves interaction fairness when matching behaviors. However, as hinted at earlier, due to the way we formalized PINI_{E}, $p \in \text{PINI}_{\mathsf{E}}$ may need to starve $p_2 \simeq_\ell p_1$ to match a behavior that $p_1 \models p$ can perform.

Theorem III.13. $p \in \text{PSNI}_{\mathsf{EF}} \iff p \in \text{PSNI}_{\mathsf{E}}$.

Theorem III.14. $p \in \text{PINI}_{\mathsf{EF}} \centernot\iff p \in \text{PINI}_{\mathsf{E}}$.

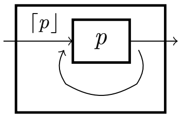


Fig. 1. Loop Combinator

D. Contrast to Trace-based Properties

Finally, to emphasize the novelty of our security properties, we demonstrate that they rule out classes of attacks which trace-based security properties, classically considered in work on compositionality in event systems [28], [32], [57], do *not* guarantee protection from. Consider the following program, which we refer to as the *extortionist*.

```
**repeat poll** H $h$ **until** $h \neq$ UNDEFINED
**out** L 0
```

This program, turned into the process p_{extort} by buffering input, will repeatedly attempt a read on H in a nonblocking manner from its buffering context; if no H input is available in the context, then the program outputs L0.

We have $\mathsf{F}^?(!\mathsf{H}^\omega) \models p_{\text{extort}} \xrightarrow{(!\diamond.?\mathsf{H}0)^\omega}$. However, we have that $\mathsf{F}^?(!\diamond) \models p_{\text{extort}}$ cannot match this behavior; thus, $p_{\text{extort}} \notin \text{PSNI}_{\mathsf{E}}$, thus $p_{\text{extort}} \notin \text{PSNI}$. Alternatively, since $\mathbb{A}_\ell^\omega$ has no fair stream matching this behavior, $p_{\text{extort}} \notin \text{PSNI}$.

However, p_{extort} *does*, for instance, satisfy *forward correctability* [25]. This is due to the fact that forward correctability is defined in terms of a trace semantics, and therefore cannot properly deal with the *definite (non)eventuality* of !L0. Pick a $t \in \mathbb{T}(p_{\text{extort}})$. If t has the $\mathtt{L}$ output, then for any $t = t'.t''$ and $?\mathsf{H}v$, there will be a $\hat{t}''$ with no H input for which $t \simeq_{\mathtt{L}} t'.?\mathsf{H}v.\hat{t}''$. Similarly if a H input is deleted. If t does not have the $\mathtt{L}$ output, then regardless of whether a H input is inserted or deleted, setting $\hat{t}'' = \epsilon$ will $\simeq_{\mathtt{L}}$-match t.

IV. Compositional Security

To study how our security properties behave under composition, we present a minimal combinator language for building systems from parts. The core of this language is *complete* in the sense that arbitrary wirings between components can be constructed, yet *structured* in the sense that the possible routes that data can take in the composed system are clearly defined by the combinators used (as opposed to being partially defined by the (un)willingness of components to synchronize on certain channels at different times).

A. First Attempt

A minimal approach to enable two processes in a composed system to interact is to introduce a *loop* combinator $\lceil\rfloor$, illustrated in Figure 1. This is the approach taken in functional reactive programming [36], where loops and simple products (e.g. $\oplus$ in Figure 3) enable modeling of arbitrary wirings.

$$p ::= \ldots \mid \lceil p \rfloor$$

In $\lceil p \rfloor$, output from p is sent to the environment, and immediately, a copy of said output is sent into p as input. Any input to $\lceil p \rfloor$ is handed to p. The semantics of $\lceil\rfloor$ is as follows.

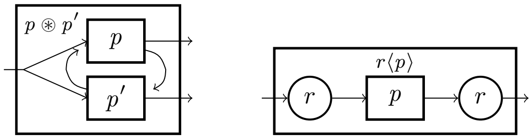


Fig. 2. Core Combinators

$$\frac{p \xrightarrow{o} p' \qquad p' \xrightarrow{o^{-1}} p''}{\lceil p \rfloor \xrightarrow{o} \lceil p'' \rfloor}\lceil\rfloor^! \qquad \frac{p \xrightarrow{i} p'}{\lceil p \rfloor \xrightarrow{i} \lceil p' \rfloor}\lceil\rfloor^?$$

Convenient as this combinator is, it enables a process to engage in a high interaction loop *with itself*. While it is possible to schedule $p_{\text{loop}} \in \text{PSNI}$ under an arbitrary environment s.t. !L0 eventually occurs, this is not the case for $\lceil p_{\text{loop}} \rfloor$; while $\mathsf{F}^?(!\mathsf{L}0.!\diamond^\omega) \models \lceil p_{\text{loop}} \rfloor \xrightarrow{?\mathsf{L}0.!\mathsf{L}0.!\diamond^\omega}$, $\mathsf{F}^?(!\mathsf{H}0.!\mathsf{L}0.!\diamond^\omega) \models \lceil p_{\text{loop}} \rfloor$ cannot match this behavior; to do ?L0 without producing !L0, p_{loop} must consume ?H0. This sends p_{loop} to its leftmost state, whereto it returns *directly* after performing !H0, as the next action of p_{loop} is invariably !H0. While this poses no problems for PINI (since a high-interaction-looping process emits infinite silence), $\lceil\rfloor$ removes the degree of control a wrapped process needs to have for PSNI to be preserved under $\lceil\rfloor$.

Theorem IV.1. $p \in \text{PINI} \implies \lceil p \rfloor \in \text{PINI}$.

Theorem IV.2. $p \in \text{PSNI} \not\implies \lceil p \rfloor \in \text{PSNI}$.

B. Core Combinators

The issue with $\lceil\rfloor$ is that it can prevent the source of an output from making progress on its observable productions, by immediately following each output it makes with an input sent directly to the source. Therefore, if our combinators are to compose under PSNI, they need to at most enable output to reach any part of the system *except* its source, in one step of the whole system. We provide two combinators which we deem to be *core* combinators, sufficient to construct arbitrary such wirings: "and", and "route".

$$p ::= \ldots \mid p \circledast p \mid r\langle p \rangle$$

These each take (possibly compound) interactive LTS_{IO} as parameter and yield a compound interactive LTS_{IO}. The structure that they impose is illustrated in Figure 2.

1) and: The *and* combinator produces a composite system from parts. An input to $p_1 \circledast p_2$ is sent to both p_1 and p_2, while output from $p_1 \circledast p_2$ comes from exactly one of p_1 and p_2, and is copied into the other as input, thus exhibiting *feedback*. This combinator is the *enabler* of communication, functioning as the "glue" with which we wire together larger systems from parts. Our $\circledast$ combinator most closely resembles the (full, arbitrary, hook-up) binary composition typically used in event-based formalisms [28], or alternatively, a broadcasting variant [41] of parallel composition in process algebra [19]. The semantics of $\circledast$ is as follows. It is clear that $\circledast$ is associative and commutative.

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o^{-1}} p'_{\mathrm{R}}}{p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \circledast p'_{\mathrm{R}}}\circledast^!_{\mathrm{L}} \qquad \frac{p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}} \qquad p_{\mathrm{L}} \xrightarrow{o^{-1}} p'_{\mathrm{L}}}{p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \circledast p'_{\mathrm{R}}}\circledast^!_{\mathrm{R}}$$

$$\frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{R}}}{p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \circledast p'_{\mathrm{R}}}\circledast^?$$

2) router: The *router* combinator $\langle\rangle$ wraps an $\textsc{lts}_{\textsc{io}}$ in a context which routes messages. The routing is defined by a router function $r : \mathbb{A} \to \mathbb{A}$ satisfying $r(\mathbb{I}) \subseteq \mathbb{I}$, $r(\mathbb{O}) \subseteq \mathbb{O}$ and $r(\$\diamond) = \$\diamond$. Then in $r\langle p\rangle$, an output o leaving p is replaced with $r(o)$, and when an input i arrives at $r\langle p\rangle$, $r(i)$ is received by p. Whereas $\circledast$ is an enabler of flows, where a composed system wires each output to each component (save the output source), $\langle\rangle$ can be used to *control* which underlying component receive which input, and to hide certain output from certain components. One example of the use of r is to map input actions on a particular channel (i.e., carrying high data) to $?\diamond$ to put the channel out of scope of the wrapped process. The semantics of $\langle\rangle$ are the following.

$$\frac{p \xrightarrow{r(i)} p'}{r\langle p\rangle \xrightarrow{i} r\langle p'\rangle}\langle\rangle^{?} \qquad \frac{p \xrightarrow{o} p'}{r\langle p\rangle \xrightarrow{r(o)} r\langle p'\rangle}\langle\rangle^{!}$$

C. Point-to-Point Variant

Instead of our broadcasting message-passing semantics, one could alternatively opt for one which, instead of sending a message along both branches of a split arrow, sends it along exactly one of them. This yields a point-to-point message-passing semantics (which is still asynchronous), represented by the "xor" combinator.

$$p ::= \ldots \mid p \boxast p$$

1) xor: The *xor* combinator $\boxast$ is similar to parallel composition in point-to-point message-passing formalisms [19]. While the component wiring in $\boxast$ is the same as for $\circledast$, the message-passing semantics differs notably: Input goes to exactly one component, and output goes exclusively to either the other component, or the environment. The semantics of $\boxast$ is as follows. It is clear that $\boxast$ is associative and commutative.

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p_{\mathrm{L}}' \quad p_{\mathrm{R}} \xrightarrow{o^{-1}} p_{\mathrm{R}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{!\diamond} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{L}}^{!\mathrm{R}} \qquad \frac{p_{\mathrm{L}} \xrightarrow{o} p_{\mathrm{L}}' \quad p_{\mathrm{R}} \xrightarrow{?\diamond} p_{\mathrm{R}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{L}}^{!\mathrm{E}}$$

$$\frac{p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{R}}' \quad p_{\mathrm{L}} \xrightarrow{o^{-1}} p_{\mathrm{L}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{!\diamond} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{R}}^{!\mathrm{L}} \qquad \frac{p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{R}}' \quad p_{\mathrm{L}} \xrightarrow{?\diamond} p_{\mathrm{L}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{R}}^{!\mathrm{E}}$$

$$\frac{p_{\mathrm{L}} \xrightarrow{i} p_{\mathrm{L}}' \quad p_{\mathrm{R}} \xrightarrow{?\diamond} p_{\mathrm{R}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{i} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{L}}^{?} \qquad \frac{p_{\mathrm{R}} \xrightarrow{i} p_{\mathrm{R}}' \quad p_{\mathrm{L}} \xrightarrow{?\diamond} p_{\mathrm{L}}'}{p_{\mathrm{L}} \boxast p_{\mathrm{R}} \xrightarrow{i} p_{\mathrm{L}}' \boxast p_{\mathrm{R}}'}\boxast_{\mathrm{R}}^{?}$$

Using $\boxast$ and $\langle\rangle$ as the combinator core would be viable in a synchronous concurrency model; there, input is only delivered to an *intended* receiver. However, in a setting where each component always waits for input on every channel, $\boxast$ nondeterministically picks a component to receive the input. The input can therefore be sent along a branch in the composition which is not intended to receive the input (a constraint modeled using a router which maps it to $?\diamond$) and therefore never reach the intended target. We therefore find that $\boxast$ is not a good fit in our framework – at least not as a *replacement* for $\circledast$. There are some merits to including $\boxast$ in a language based on our combinator core, e.g. if, at the combinator level, one wishes to model nondeterministic dispatching of input to servers.

D. Compositionality

We now explore the compositionality of our security properties, to then give a *language of secure combinators* for building secure systems from secure parts. While security properties are known to be fragile under composition [32], the proof technique arising from the design of our security properties yields positive results.

We consider compositionality of both $\circledast$ and $\boxast$ since they cannot be easily defined in terms of each other; $\circledast$ and $\langle\rangle$ have no source of nondeterminism needed to model the nondeterministic behavior of $\boxast$ (although this can be supplied by an $\textsc{lts}_{\textsc{io}}$), and no combination of $\boxast$ and $\langle\rangle$ can guarantee eventual delivery between any pair of components like $\circledast$ can.

1) and: We begin with the most important combinator, $\circledast$. It composes under PSNI.

Theorem IV.3. $p_{\mathrm{L}}, p_{\mathrm{R}} \in \mathrm{PSNI} \implies p_{\mathrm{L}} \circledast p_{\mathrm{R}} \in \mathrm{PSNI}$.

The proof of this is as follows. For any $p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{s}$, we must show existence of a $p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{s'}$ for which $s' \in \mathbb{A}_\ell^\omega$, $s \simeq_\ell s'$ and $\mathsf{preserve}_{p_{\mathrm{L}} \circledast p_{\mathrm{R}}, s}(\epsilon, s')$. Since $p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{s}$, there are some s_{L}, s_{R} for which $p_{\mathrm{L}} \xrightarrow{s_{\mathrm{L}}}$, $p_{\mathrm{R}} \xrightarrow{s_{\mathrm{R}}}$ and $s_{\mathrm{L}} \circledast s_{\mathrm{R}} \xrightarrow{s}$. Indeed, for any nth action a_{L}, a_{R} and a in s_{L}, s_{R} and s respectively, if $a = i$ (environment input) for some i, then $a_{\mathrm{L}} = i = a_{\mathrm{R}}$, and if $a = o$ (component output), then either $a_{\mathrm{L}} = o$ and $a_{\mathrm{R}} = o^{-1}$, or vice versa. Since $p_{\mathrm{L}}, p_{\mathrm{R}} \in \mathrm{PSNI}$, there exist $p_{\mathrm{L}} \xrightarrow{s_{\mathrm{L}}'}$ and $p_{\mathrm{R}} \xrightarrow{s_{\mathrm{R}}'}$ for which $s_{\mathrm{L}}', s_{\mathrm{R}}' \in \mathbb{A}_\ell^\omega$, $s_{\mathrm{L}} \simeq_\ell s_{\mathrm{L}}'$, $s_{\mathrm{L}} \simeq_\ell s_{\mathrm{L}}'$, $\mathsf{preserve}_{p_{\mathrm{L}}, s_{\mathrm{L}}}(\epsilon, s_{\mathrm{L}}')$ and $\mathsf{preserve}_{p_{\mathrm{R}}, s_{\mathrm{R}}}(\epsilon, s_{\mathrm{R}}')$.

We obtain s' by "zipping" s_{L}' and s_{R}' in a manner guided by the observables in s_{L} and s_{R} (observables in both are the same as observables in s modulo direction) as follows:

Assume for $\hat{s}_{\mathrm{L}}, \hat{s}_{\mathrm{R}} \in \mathbb{A}_\ell^\omega$, t_{L}, t_{R}, t that $s_{\mathrm{L}} \simeq_\ell t_{\mathrm{L}}.\hat{s}_{\mathrm{L}}$, $s_{\mathrm{R}} \simeq_\ell t_{\mathrm{R}}.\hat{s}_{\mathrm{R}}$, $p_{\mathrm{L}} \xrightarrow{t_{\mathrm{L}}.\hat{s}_{\mathrm{L}}}$, $p_{\mathrm{R}} \xrightarrow{t_{\mathrm{R}}.\hat{s}_{\mathrm{R}}}$, $\mathsf{preserve}_{p_{\mathrm{L}}, s_{\mathrm{L}}}(t_{\mathrm{L}}, \hat{s}_{\mathrm{L}})$, $\mathsf{preserve}_{p_{\mathrm{R}}, s_{\mathrm{R}}}(t_{\mathrm{R}}, \hat{s}_{\mathrm{R}})$, $t \in \mathbb{A}_\ell^*$, $t \preceq_\ell s$, and $t_{\mathrm{L}} \circledast t_{\mathrm{R}} \xrightarrow{t}$.

We show existence of $\hat{s}_{\mathrm{L}}', \hat{s}_{\mathrm{R}}' \in \mathbb{A}_\ell^\omega$, t_{L}', t_{R}', t' for which $s_{\mathrm{L}} \simeq_\ell t_{\mathrm{L}}'.\hat{s}_{\mathrm{L}}'$, $s_{\mathrm{R}} \simeq_\ell t_{\mathrm{R}}'.\hat{s}_{\mathrm{R}}'$, $p_{\mathrm{L}} \xrightarrow{t_{\mathrm{L}}'.\hat{s}_{\mathrm{L}}'}$, $p_{\mathrm{R}} \xrightarrow{t_{\mathrm{R}}'.\hat{s}_{\mathrm{R}}'}$, $\mathsf{preserve}_{p_{\mathrm{L}}, s_{\mathrm{L}}}(t_{\mathrm{L}}', \hat{s}_{\mathrm{L}}')$, $\mathsf{preserve}_{p_{\mathrm{R}}, s_{\mathrm{R}}}(t_{\mathrm{R}}', \hat{s}_{\mathrm{R}}')$, $t' \in \mathbb{A}_\ell^*$, $t' \preceq_\ell s$, $t_{\mathrm{L}}' \circledast t_{\mathrm{R}}' \xrightarrow{t'}$, $t_{\mathrm{L}} < t_{\mathrm{L}}'$, $t_{\mathrm{R}} < t_{\mathrm{R}}'$, $t < t'$, and $\hat{s}_{\mathrm{L}}, \hat{s}_{\mathrm{R}} \not\simeq_\ell \epsilon \implies t \not\simeq_\ell t'$. (*).

Assume $o_{\mathrm{L}} \preceq_\ell \hat{s}_{\mathrm{L}}$ for some $o_{\mathrm{L}} \not\simeq_\ell \epsilon$ (proof for $o_{\mathrm{R}} \preceq_\ell \hat{s}_{\mathrm{R}}$ case obtained by swapping L and R). Then $\hat{s}_{\mathrm{L}} = \bar{o}_{\mathrm{L}}.o_{\mathrm{L}}.\hat{s}_{\mathrm{L}}''$ for some $\bar{o}_{\mathrm{L}} \simeq_\ell \epsilon$ and $\hat{s}_{\mathrm{L}}''$. Through repeated application of Def. III.1 pt. 1), $s_{\mathrm{L}} \simeq_\ell t_{\mathrm{L}}.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}.\hat{s}_{\mathrm{L}}''$, $p_{\mathrm{L}} \xrightarrow{t_{\mathrm{L}}.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}.\hat{s}_{\mathrm{L}}''}$, and $\mathsf{preserve}_{p_{\mathrm{L}}, s_{\mathrm{L}}}(t_{\mathrm{L}}.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}, \hat{s}_{\mathrm{L}}'')$. Through repeated application of Def. III.1 pt. 2), we have some $\hat{s}_{\mathrm{R}}''$ for which $s_{\mathrm{R}} \simeq_\ell t_{\mathrm{R}}.\bar{o}_{\mathrm{L}}{}^{-1}.o_{\mathrm{L}}{}^{-1}.\hat{s}_{\mathrm{R}}''$, $p_{\mathrm{R}} \xrightarrow{t_{\mathrm{R}}.\bar{o}_{\mathrm{L}}{}^{-1}.o_{\mathrm{L}}{}^{-1}.\hat{s}_{\mathrm{R}}''}$, and $\mathsf{preserve}_{p_{\mathrm{R}}, s_{\mathrm{R}}}(t_{\mathrm{R}}.\bar{o}_{\mathrm{L}}{}^{-1}.o_{\mathrm{L}}{}^{-1}, \hat{s}_{\mathrm{R}}'')$. Further, we have that $t_{\mathrm{L}}.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}} \circledast t_{\mathrm{R}}.\bar{o}_{\mathrm{L}}{}^{-1}.o_{\mathrm{L}}{}^{-1} \xrightarrow{t.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}}$ and $p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{t.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}}$. Set $\hat{s}_{\mathrm{L}}' = \hat{s}_{\mathrm{L}}''$, $\hat{s}_{\mathrm{R}}' = \hat{s}_{\mathrm{R}}''$, $t_{\mathrm{L}}' = t_{\mathrm{L}}.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}$, $t_{\mathrm{R}}' = t_{\mathrm{R}}.\bar{o}_{\mathrm{L}}{}^{-1}.o_{\mathrm{L}}{}^{-1}$, and $t' = t.\bar{o}_{\mathrm{L}}.o_{\mathrm{L}}$, and we get (*).

Assume $i \preceq_\ell \hat{s}_{\mathrm{L}}$ and $i \preceq_\ell \hat{s}_{\mathrm{R}}$ for some $i \not\simeq_\ell \epsilon$ and $i \in \mathbb{A}_\ell$. Through single application of Def. III.1 pt. 2), we have for some $\hat{s}_{\mathrm{L}}''$ that $s_{\mathrm{L}} \simeq_\ell t_{\mathrm{L}}.i.\hat{s}_{\mathrm{L}}''$, $p_{\mathrm{L}} \xrightarrow{t_{\mathrm{L}}.i.\hat{s}_{\mathrm{L}}''}$, and $\mathsf{preserve}_{p_{\mathrm{L}}, s_{\mathrm{L}}}(t_{\mathrm{L}}.i, \hat{s}_{\mathrm{L}}'')$. Through single application of Def. III.1 pt. 2), we have for some $\hat{s}_{\mathrm{R}}''$ that $s_{\mathrm{R}} \simeq_\ell t_{\mathrm{R}}.i.\hat{s}_{\mathrm{R}}''$, $p_{\mathrm{R}} \xrightarrow{t_{\mathrm{R}}.i.\hat{s}_{\mathrm{R}}''}$, and $\mathsf{preserve}_{p_{\mathrm{R}}, s_{\mathrm{R}}}(t_{\mathrm{R}}.i, \hat{s}_{\mathrm{R}}'')$. Further, we have that $t_{\mathrm{L}}.i \circledast t_{\mathrm{R}}.i \xrightarrow{t.i}$ and $p_{\mathrm{L}} \circledast p_{\mathrm{R}} \xrightarrow{t.i}$. Set $\hat{s}_{\mathrm{L}}' = \hat{s}_{\mathrm{L}}''$, $\hat{s}_{\mathrm{R}}' = \hat{s}_{\mathrm{R}}''$, $t_{\mathrm{L}}' = t_{\mathrm{L}}.i$, $t_{\mathrm{R}}' = t_{\mathrm{R}}.i$, and $t' = t.i$, and we get (*).

Assume $\hat{s}_{\mathrm{L}} \simeq_\ell \hat{s}_{\mathrm{R}} \simeq_\ell \epsilon$ (equally valid proof obtained by swapping L and R in the following). Then $\hat{s}_{\mathrm{L}} = o_{\mathrm{L}}.\hat{s}_{\mathrm{L}}''$

for some $o_\mathtt{L} \simeq_\ell \epsilon$ and $\hat{s}_\mathtt{L}''$. Through single application of Def. III.1 pt. 1), we have that $s_\mathtt{L} \simeq_\ell t_\mathtt{R}.o_\mathtt{L}.\hat{s}_\mathtt{L}''$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}.o_\mathtt{L}.\hat{s}_\mathtt{L}''}$, and $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}.o_\mathtt{L}, \hat{s}_\mathtt{L}'')$. Through single application of Def. III.1 pt. 2), we have for some $\hat{s}_\mathtt{R}''$ that $s_\mathtt{R} \simeq_\ell t_\mathtt{R}.{o_\mathtt{L}}^{-1}.\hat{s}_\mathtt{R}''$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}.{o_\mathtt{L}}^{-1}.\hat{s}_\mathtt{R}''}$, and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}.{o_\mathtt{L}}^{-1}, \hat{s}_\mathtt{R}'')$. Further, $t_\mathtt{L}.o_\mathtt{L} \circledast t_\mathtt{R}.{o_\mathtt{L}}^{-1} \xrightarrow{t.o_\mathtt{L}}$ and $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t.o_\mathtt{L}}$. Set $\hat{s}_\mathtt{L}' = \hat{s}_\mathtt{L}''$, $\hat{s}_\mathtt{R}' = \hat{s}_\mathtt{R}''$, $t_\mathtt{L}' = t_\mathtt{L}.o_\mathtt{L}$, $t_\mathtt{R}' = t_\mathtt{R}.{o_\mathtt{L}}^{-1}$, and $t' = t.o_\mathtt{L}$, and we get (*).

Recall $p_\mathtt{L}, p_\mathtt{R} \in \text{PSNI}$, $p_\mathtt{L} \xrightarrow{s_\mathtt{L}'}$, $p_\mathtt{R} \xrightarrow{s_\mathtt{R}'}$, $s_\mathtt{L}', s_\mathtt{R}' \in \mathbb{A}_\ell^\omega$, $s_\mathtt{L} \simeq_\ell s_\mathtt{L}'$, $s_\mathtt{L} \simeq_\ell s_\mathtt{L}'$, $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(\epsilon, s_\mathtt{L}')$ and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(\epsilon, s_\mathtt{R}')$. Now (*) gives us traces $t_0 < t_1 < \ldots$ s.t. $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t_j}$ and $t_j \in \mathbb{A}_\ell^*$ for all $j \geq 0$. Let s' be the fixed point of these traces. Then $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{s'}$ and $s' \in \mathbb{A}_\ell^\omega$, $s \simeq_\ell s'$.

To establish $\mathsf{preserve}_{p_\mathtt{L} \circledast p_\mathtt{R},s}(\epsilon, s')$, we proceed as follows. Assume for $\hat{s}_\mathtt{L}, \hat{s}_\mathtt{R} \in \mathbb{A}_\ell^\omega$, $t_\mathtt{L}$, $t_\mathtt{R}$, t that $s_\mathtt{L} \simeq_\ell t_\mathtt{L}.\hat{s}_\mathtt{L}$, $s_\mathtt{R} \simeq_\ell t_\mathtt{R}.\hat{s}_\mathtt{R}$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}.\hat{s}_\mathtt{L}}$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}.\hat{s}_\mathtt{R}}$, $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}, \hat{s}_\mathtt{L})$, $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}, \hat{s}_\mathtt{R})$, $\hat{s} \in \mathbb{A}_\ell^\omega$, $s \simeq_\ell t.\hat{s}$, and $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t.\hat{s}}$.

To prove $\mathsf{preserve}_{p_\mathtt{L} \circledast p_\mathtt{R},s}(t, \hat{s})$, we must show for each of Def. III.1 pt. 1)-3) that there exist $\hat{s}' \in \mathbb{A}_\ell^\omega$ and t' satisfying structural requirements imposed by the pt such that $s \simeq_\ell t'.\hat{s}'$ and $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t'.\hat{s}'}$, and that there exist $\hat{s}_\mathtt{L}', \hat{s}_\mathtt{R}' \in \mathbb{A}_\ell^\omega$, $t_\mathtt{L}'$, and $t_\mathtt{R}'$ for which $s_\mathtt{L} \simeq_\ell t_\mathtt{L}'.\hat{s}_\mathtt{L}'$, $s_\mathtt{R} \simeq_\ell t_\mathtt{R}'.\hat{s}_\mathtt{R}'$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}'.\hat{s}_\mathtt{L}'}$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}'.\hat{s}_\mathtt{R}'}$, $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}', \hat{s}_\mathtt{L}')$, $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}', \hat{s}_\mathtt{R}')$ and $t_\mathtt{L}' \circledast t_\mathtt{R}' \xrightarrow{t'}$. (+).

For pt. 1), assume $o \leq \hat{s}$ for some o. Then either $o \leq \hat{s}_\mathtt{L}$ or $o \leq \hat{s}_\mathtt{R}$. Assume wlg that $o \leq \hat{s}_\mathtt{L}$ (and that $o \leq \hat{s}$ came from $p_\mathtt{L}$ during construction of s'). Since $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}, \hat{s}_\mathtt{L})$, we have through single application of Def. III.1 pt. 1) that for some $\hat{s}_\mathtt{L}'$, $s_\mathtt{L} \simeq_\ell t_\mathtt{L}.o.\hat{s}_\mathtt{L}'$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}.o.\hat{s}_\mathtt{L}'}$, and $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}.o, s_\mathtt{L}'')$. Since $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}, \hat{s}_\mathtt{R})$, we get through single application of Def. III.1 pt. 2) some $\hat{s}_\mathtt{R}''$ for which $s_\mathtt{R} \simeq_\ell t_\mathtt{R}.o^{-1}.\hat{s}_\mathtt{R}''$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}.o^{-1}.\hat{s}_\mathtt{R}''}$, and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}.o^{-1}, \hat{s}_\mathtt{R}'')$. Further, we have that $t_\mathtt{L}.o \circledast t_\mathtt{R}.o^{-1} \xrightarrow{t.o}$ and $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t.o_\mathtt{L}}$. Set $\hat{s}_\mathtt{L}' = \hat{s}_\mathtt{L}''$, $\hat{s}_\mathtt{R}' = \hat{s}_\mathtt{R}''$, $t_\mathtt{L}' = t_\mathtt{L}.o$, $t_\mathtt{R}' = t_\mathtt{R}.o^{-1}$, $t' = t.o$, and use the above-described approach to obtain $\hat{s}'$ from these, and we get (+).

For pt. 2), assume $i \preceq_\ell \hat{s}$ for some i, Then $i \preceq_\ell \hat{s}_\mathtt{L}$ and $i \preceq_\ell \hat{s}_\mathtt{R}$. By $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}, \hat{s}_\mathtt{L})$ and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}, \widehat{str}_\mathtt{R})$, we get through single application of Def. III.1 pt. 2) some $\hat{s}_\mathtt{L}'$, $\hat{s}_\mathtt{R}'$ for which $s_\mathtt{L} \simeq_\ell t_\mathtt{L}.i.\hat{s}_\mathtt{L}'$, $s_\mathtt{R} \simeq_\ell t_\mathtt{R}.i.\hat{s}_\mathtt{R}'$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}.i.\hat{s}_\mathtt{L}'}$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}.i.\hat{s}_\mathtt{R}'}$, $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}.i, \hat{s}_\mathtt{L}')$, and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}.i, \hat{s}_\mathtt{R}')$. Set $\hat{s}_\mathtt{L}' = \hat{s}_\mathtt{L}''$, $\hat{s}_\mathtt{R}' = \hat{s}_\mathtt{R}''$, $t_\mathtt{L}' = t_\mathtt{L}.i$, $t_\mathtt{R}' = t_\mathtt{R}.i$, $t' = t.i$, and use the above-described approach to obtain $\hat{s}'$ from these, and we get (+).

For pt. 3), assume $\tilde{i} \simeq_\ell \hat{s}$ for some $\tilde{i}$. By $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}, \hat{s}_\mathtt{L})$ (the same argument with $\mathtt{L}$ and $\mathtt{R}$ swapped also holds), we get through single application of Def. III.1 pt. 3) some $\bar{i}$, o and $\hat{s}_\mathtt{L}'$ for which $\bar{i} \leq \tilde{i}$, $s_\mathtt{L} \simeq_\ell t_\mathtt{L}.\bar{i}.o.\hat{s}_\mathtt{L}'$, $p_\mathtt{L} \xrightarrow{t_\mathtt{L}.\bar{i}.o.\hat{s}_\mathtt{L}'}$ and $\mathsf{preserve}_{p_\mathtt{L},s_\mathtt{L}}(t_\mathtt{L}.\bar{i}.o, \hat{s}_\mathtt{L}')$. By repeated application of pt. 2) for each i in $\bar{i}$ and for o^{-1}, we have for some $\hat{s}_\mathtt{R}'$ that $s_\mathtt{R} \simeq_\ell t_\mathtt{R}.\bar{i}.o^{-1}.\hat{s}_\mathtt{R}'$, $p_\mathtt{R} \xrightarrow{t_\mathtt{R}.\bar{i}.o^{-1}.\hat{s}_\mathtt{R}'}$ and $\mathsf{preserve}_{p_\mathtt{R},s_\mathtt{R}}(t_\mathtt{R}.\bar{i}.o^{-1}, \hat{s}_\mathtt{R}')$. Set $\hat{s}_\mathtt{L}' = \hat{s}_\mathtt{L}''$, $\hat{s}_\mathtt{R}' = \hat{s}_\mathtt{R}''$, $t_\mathtt{L}' = t_\mathtt{L}.\bar{i}.o$, $t_\mathtt{R}' = t_\mathtt{R}.\bar{i}.o^{-1}$, $t' = t.\bar{i}.o$, and use the above-described approach to obtain $\hat{s}'$ from these, and we get (+).

Thus $\mathsf{preserve}_{p_\mathtt{L} \circledast p_\mathtt{R},s}(\epsilon, s')$, which completes this proof.

Furthermore, it turns out that PINI composes under $\circledast$, *if* we e.g. change the way $\hat{s}_\mathtt{L}'$ and $\hat{s}_\mathtt{R}$ are zipped to form s' such that if one runs out of observables, say, $s_\mathtt{L}'$, then the rest of s' is set to the rest of $s_\mathtt{L}'$.

Corollary IV.4. $p_\mathtt{L}, p_\mathtt{R} \in \text{PINI} \implies p_\mathtt{L} \circledast p_\mathtt{R} \in \text{PINI}.$

2) xor: The proof that PSNI and PINI compose under $\boxast$ is similar to the above "zip-and-preserve" proof of compositionality of $\circledast$. Since environment input does not enter both components, and since output from a component does not both go to the environment and the other component, the zipping procedure consults s in addition to $\hat{s}_\mathtt{L}$ and $\hat{s}_\mathtt{R}$ since s tells us *where* observables in $s_\mathtt{L}$ and $s_\mathtt{R}$ came from and went to.

Corollary IV.5. $p_\mathtt{L}, p_\mathtt{R} \in \text{PSNI} \implies p_\mathtt{L} \boxast p_\mathtt{R} \in \text{PSNI}.$

Corollary IV.6. $p_\mathtt{L}, p_\mathtt{R} \in \text{PINI} \implies p_\mathtt{L} \boxast p_\mathtt{R} \in \text{PINI}.$

3) route: Since a router can route any message to any channel, and change values in messages, a router has the capacity to reveal the presence of $\mathtt{H}$-presences message or values in $\mathtt{H}$-content messages. However, as long as a router function never moves information down in the security lattice, wrapping a secure process in it yields a secure process.

Definition IV.7. r is *ℓ-secure* iff $\forall c, c_1, v, v_1 \,\textbf{.}$

$$r(\$cv) = \$c_1v_1 \wedge \kappa(c) \not\sqsubseteq \ell \implies$$

- $\pi(c) \sqsubseteq \ell \implies \forall v', c_2, v_2 \,\textbf{.}\, r(\$cv') = \$c_2v_2 \implies c_1 = c_2$
- $\pi(c) \not\sqsubseteq \ell \implies \pi(c_1) \not\sqsubseteq \ell.$

We say r is *secure* if it is ℓ-secure for all ℓ. ◊

Compositionality of $\langle\rangle$ is immediate from Definition III.1.

Corollary IV.8. $p \in \text{PSNI} \implies r\langle p\rangle \in \text{PSNI}$ *for secure* r.

Corollary IV.9. $p \in \text{PINI} \implies r\langle p\rangle \in \text{PINI}$ *for secure* r.

E. Fairness

As we have discussed, autonomy, and therefore fairness, is a key feature of interactive LTS_IO. However, the proof we just saw does not rule out the possibility of starvation. For instance, in the zipper given in the proof of $\circledast$, when $\hat{s}_\mathtt{L}$ has an infinite number of observables and $\hat{s}_\mathtt{R}$ has no observable output, the zipper can ignore the remainder of $\hat{s}_\mathtt{R}$. Also, the PINI zipper ignores the remainder of $\hat{s}_\mathtt{R}$ when $\hat{s}_\mathtt{L}$ has no more observables. The central question here is whether our security properties *require* starvation of components to remain compositional. This prompts us to study *fair* combinators. What we find is that whereas PSNI composes freely assuming fairness, PINI relies *fundamentally* on *lack of fairness* to be compositional for even simple product compositions.

1) Fair Composition: A fair combinator permits only fair behaviors, i.e., ones which do not starve components, always allowing each to eventually make progress on its outputs.

Definition IV.10. For $p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{s}$, s is *$\circledast$-fair*, iff, $\forall t \leq s \,\textbf{.}$

$$\exists t_\mathtt{L}, t_\mathtt{R}, o_\mathtt{L}, o_\mathtt{R}, s_\mathtt{L}, s_\mathtt{R}, p_\mathtt{LL}, p_\mathtt{RL}, p_\mathtt{LR}, p_\mathtt{RR}, p_\mathtt{LL}', p_\mathtt{LR}', p_\mathtt{RL}', p_\mathtt{RR}' \,\textbf{.}$$

$$s = t.t_\mathtt{L}.o_\mathtt{L}.s_\mathtt{L} \;\wedge\; s = t.t_\mathtt{R}.o_\mathtt{R}.s_\mathtt{R} \;\wedge$$

$$p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t.t_\mathtt{L}} p_\mathtt{LL} \circledast p_\mathtt{RL} \xrightarrow{o_\mathtt{L}} p_\mathtt{LL}' \circledast p_\mathtt{RL}' \xrightarrow{s_\mathtt{L}} \;\wedge$$

$$p_\mathtt{L} \circledast p_\mathtt{R} \xrightarrow{t.t_\mathtt{R}} p_\mathtt{LR} \circledast p_\mathtt{RR} \xrightarrow{o_\mathtt{R}} p_\mathtt{LR}' \circledast p_\mathtt{RR}' \xrightarrow{s_\mathtt{R}} \;\wedge$$

$$p_\mathtt{LL} \xrightarrow{o_\mathtt{L}} p_\mathtt{LL}' \;\wedge\; p_\mathtt{RL} \xrightarrow{{o_\mathtt{L}}^{-1}} p_\mathtt{RL}' \;\wedge$$

$$p_\mathtt{RR} \xrightarrow{o_\mathtt{L}} p_\mathtt{RR}' \;\wedge\; p_\mathtt{LR} \xrightarrow{{o_\mathtt{L}}^{-1}} p_\mathtt{LR}'.$$

$p_{\mathtt{L}} \circledast_{\mathsf{F}} p_{\mathtt{R}} \xrightarrow{s}$ iff $p_{\mathtt{L}} \circledast p_{\mathtt{R}} \xrightarrow{s}$ and s is $\circledast$-fair. $\Diamond$

We let $\boxast_{\mathsf{F}}$ be defined in a similar manner.

2) PSNI *Composes Fairly:* Modifying the zip-and-preserve proof above for the compositionality of PSNI under $\circledast_{\mathsf{F}}$ and $\boxast_{\mathsf{F}}$ is easy; when $\hat{s}_{\mathtt{R}}$ runs out of observable output, we zip in such a way that $\hat{s}'$ takes turns in pulling an output from $\hat{s}_{\mathtt{R}}$ and $\hat{s}_{\mathtt{L}}$ into t, irrespective of when and how many input appears before the outputs. We therefore have the following.

Corollary IV.11. $p_{\mathtt{L}}, p_{\mathtt{R}} \in \text{PSNI} \implies p_{\mathtt{L}} \circledast_{\mathsf{F}} p_{\mathtt{R}} \in \text{PSNI}.$

Corollary IV.12. $p_{\mathtt{L}}, p_{\mathtt{R}} \in \text{PSNI} \implies p_{\mathtt{L}} \boxast_{\mathsf{F}} p_{\mathtt{R}} \in \text{PSNI}.$

3) PINI *Composes Unfairly:* However, the same cannot be said for PINI; it fails to be preserved under even simple fair products. The way in which PINI fails to compose is *not* due to the simplification we made of the definition of PINI discussed in Section III-B. We therefore maintain that, in the interactive setting, PINI relies *fundamentally* on unfairness to be preserved under composition, making it a *poor* target property for reasoning about security of autonomous processes.

Theorem IV.13. $p_{\mathtt{L}}, p_{\mathtt{R}} \in \text{PINI} \;\not\Longrightarrow\; p_{\mathtt{L}} \circledast_{\mathsf{F}} p_{\mathtt{R}} \in \text{PINI}.$

To prove this, we give two programs which satisfy PINI which fair composition fails to be PINI. It illustrates how a progress-difference in one component can translate into an *explicit flow* under composition. Consider this program $p_{\mathtt{A}}$,

```
  in M $h$
  if $h \bmod 2 = 0$ then out L 0 end if
```

and the following program $p_{\mathtt{B}}$.

```
  out L 1
```

Wrapped in $\mathsf{W} \circ \mathsf{B}^{?} \circ \mathsf{Z}$, both satisfy PINI. In fact, $p_{\mathtt{B}}$ satisfies PSNI. Now consider environments $p_1 = \mathsf{F}^{?}(!\mathsf{M0}.!\diamond^{\omega})$ and $p_2 = \mathsf{F}^{?}(!\mathsf{M1}.!\diamond^{\omega})$. Then we have that $p_1 \models p_{\mathtt{A}} \circledast_{\mathsf{F}} p_{\mathtt{B}}$ can match $?\mathsf{M0}.!\mathsf{L0}.!\mathsf{L1}$. However, $p_2 \models p_{\mathtt{A}} \circledast_{\mathsf{F}} p_{\mathtt{B}}$ cannot match this behavior; $p_{\mathtt{A}}$ produces no observables, and the composition cannot postpone the observable $p_{\mathtt{B}}$ wishes to perform indefinitely, so at best, $p_1 \models p_{\mathtt{A}} \circledast_{\mathsf{F}} p_{\mathtt{B}}$ can match $?\mathsf{M1}.!\mathsf{L1}$, which is not $\approx_{\ell}$-equivalent to $?\mathsf{M0}.!\mathsf{L0}.!\mathsf{L1}$. Thus $p_{\mathtt{A}} \circledast_{\mathsf{F}} p_{\mathtt{B}} \notin \text{PINI}_{\mathsf{E}}$. and thus, $p_{\mathtt{A}} \circledast_{\mathsf{F}} p_{\mathtt{B}} \notin \text{PINI}$.

This same counterexample shows PINI also fails to compose under $\boxast_{\mathsf{F}}$.

Corollary IV.14. $p_{\mathtt{L}}, p_{\mathtt{R}} \in \text{PINI} \;\not\Longrightarrow\; p_{\mathtt{L}} \boxast_{\mathsf{F}} p_{\mathtt{R}} \in \text{PINI}.$

While PINI may be justifiable if it composes under simpler combinators, it turns out that PINI fails to compose fairly *even* under products and cascades, as we will see in Section V-B. Finally, $p_{\mathtt{B}} \in$ PSNI. This means that even if a process is the *only* PINI process in a composition with PSNI processes, it cannot be guaranteed that the composition *even* satisfies PINI. The only way we can be sure of this in general is if each PINI process operates in a part of the security lattice *disjoint* from where all other processes operate, as then, varied presence of output by PINI components will not interfere with the behavior of the other components.

For these reasons, we deem PINI unfit for reasoning about security of autonomous processes.

V. Language of Secure Combinators

To showcase the generality of our combinator core and compositionality results, we give an rich language of combinators with which to build secure systems from secure parts. Since the combinators are implemented in terms of core combinators, they all compose under PSNI and PINI, and compose fairly under PSNI.

A. Derived Combinators

Figure 3 contains the full set of binary combinators which, for each component, has a path from input, through it, to output. Some of them appear regularly in literature on compositionality of security properties, e.g. product, (relaxed) cascade and feedback [28], [32], [56]. We show how these, and other, combinators can be implemented in terms of our core combinators. For comparison, we have placed their operational semantics in the appendix.

1) Relaxed Cascade Feedback: This combinator, denoted $\otimes$, behaves like $\circledast$, except that input to the composition is only delivered to the left component in the composition. The combinator can therefore be seen as a *relaxed* relaxed cascade. Using our combinators, we can implement $\otimes$ as

$$p_{\mathtt{L}} \otimes p_{\mathtt{R}} = r_{\mathsf{E}}\langle r_{\mathtt{L}}\langle p_{\mathtt{L}}\rangle \circledast r_{\mathtt{R}}\langle p_{\mathtt{R}}\rangle\rangle,$$

where

$$\begin{array}{lll} r_{\mathsf{E}}(i) = i_{\mathsf{E}} & r_{\mathtt{L}}(o) = o_{\mathtt{L}} & r_{\mathtt{R}}(o) = o_{\mathtt{R}} \\ r_{\mathsf{E}}(o_{\mathtt{L}}) = o & r_{\mathtt{L}}(i_{\mathsf{E}}) = i & r_{\mathtt{R}}(i_{\mathsf{E}}) = ?\diamond \\ r_{\mathsf{E}}(o_{\mathtt{R}}) = o & r_{\mathtt{L}}(i_{\mathtt{R}}) = i & r_{\mathtt{R}}(i_{\mathtt{L}}) = i. \end{array}$$

Here, a_X is a which message is labeled X (e.g. by partitioning $\mathbb{C}$ or $\mathbb{V}$). The label expresses who is the producer behind a; E is the environment, $\mathtt{L}$ is $p_{\mathtt{L}}$ and $\mathtt{R}$ is $p_{\mathtt{R}}$. Thus, for instance, $r_{\mathtt{R}}(i_{\mathsf{E}}) = ?\diamond$ expresses that $p_{\mathtt{R}}$ should not receive a message that originated from the environment.

2) Relaxed Cascade: This combinator, denoted $\obslash$, is a relaxation of cascade (i.e., sequential) composition often considered in work in compositionality. Like in a cascade, input to a relaxed cascade enters only the first component, and output from the second component is sent only to the environment. However, output from the first component is replicated and sent both to the environment and the second component. Using our core, we can implement $\obslash$ as

$$p_{\mathtt{L}} \obslash p_{\mathtt{R}} = r_{\mathsf{E}}\langle r_{\mathtt{L}}\langle p_{\mathtt{L}}\rangle \circledast r_{\mathtt{R}}\langle p_{\mathtt{R}}\rangle\rangle,$$

where

$$\begin{array}{lll} r_{\mathsf{E}}(i) = i_{\mathsf{E}} & r_{\mathtt{L}}(o) = o_{\mathtt{L}} & r_{\mathtt{R}}(o) = o_{\mathtt{R}} \\ r_{\mathsf{E}}(o_{\mathtt{L}}) = o & r_{\mathtt{L}}(i_{\mathsf{E}}) = i & r_{\mathtt{R}}(i_{\mathsf{E}}) = ?\diamond \\ r_{\mathsf{E}}(o_{\mathtt{R}}) = o & r_{\mathtt{L}}(i_{\mathtt{R}}) = ?\diamond & r_{\mathtt{R}}(i_{\mathtt{L}}) = i. \end{array}$$

3) Cascade: Also known as sequential composition, combinator $\ominus$ is a basic combinator typically seen as a primitive in various combinator formalisms (e.g. [36]). Input entering a cascade enters the first component only, output from the first component enters the second component only, and output from the second component becomes the output of the cascade. Using our core, we can implement $\ominus$ as

$$p_{\mathtt{L}} \ominus p_{\mathtt{R}} = r_{\mathsf{E}}\langle r_{\mathtt{L}}\langle p_{\mathtt{L}}\rangle \circledast r_{\mathtt{R}}\langle p_{\mathtt{R}}\rangle\rangle,$$

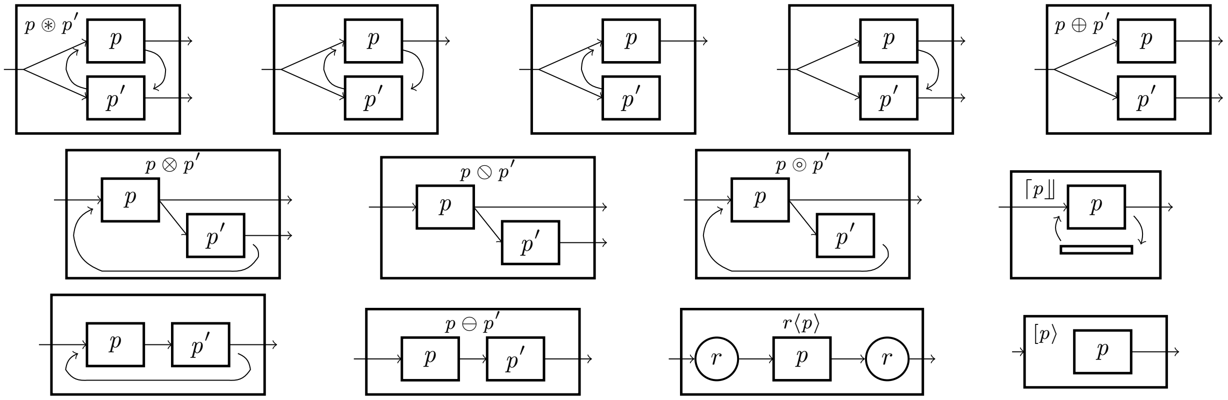


Fig. 3. Secure Combinators

where

$$\begin{array}{lll} r_{\mathsf{E}}(i) = i_{\mathsf{E}} & r_{\mathtt{L}}(o) = o_{\mathtt{L}} & r_{\mathtt{R}}(o) = o_{\mathtt{R}} \\ r_{\mathsf{E}}(o_{\mathtt{L}}) = !\diamond & r_{\mathtt{L}}(i_{\mathsf{E}}) = i & r_{\mathtt{R}}(i_{\mathsf{E}}) = ?\diamond \\ r_{\mathsf{E}}(o_{\mathtt{R}}) = o & r_{\mathtt{L}}(i_{\mathtt{R}}) = ?\diamond & r_{\mathtt{R}}(i_{\mathtt{L}}) = i. \end{array}$$

4) Product: This combinator, denoted $\oplus$, behaves like $\circledast$ without feedback (or alternatively, like parallel composition without sharing [8]). Such *product* compositions are frequently considered in work on compositionality (e.g. [28], [42]) and taken as a primitive in combinator formalisms (e.g. [36]). Using our combinators, we can implement $\oplus$ as

$$p_{\mathtt{L}} \oplus p_{\mathtt{R}} = r_{\mathsf{E}}\langle r_{\mathtt{L}}\langle p_{\mathtt{L}}\rangle \circledast r_{\mathtt{R}}\langle p_{\mathtt{R}}\rangle\rangle,$$

where

$$\begin{array}{lll} r_{\mathsf{E}}(i) = i_{\mathsf{E}} & r_{\mathtt{L}}(o) = o_{\mathtt{L}} & r_{\mathtt{R}}(o) = o_{\mathtt{R}} \\ r_{\mathsf{E}}(o_{\mathtt{L}}) = o & r_{\mathtt{L}}(i_{\mathsf{E}}) = i & r_{\mathtt{R}}(i_{\mathsf{E}}) = i \\ r_{\mathsf{E}}(o_{\mathtt{R}}) = o & r_{\mathtt{L}}(i_{\mathtt{R}}) = ?\diamond & r_{\mathtt{R}}(i_{\mathtt{L}}) = ?\diamond. \end{array}$$

5) Feedback: A specialization of $\otimes$, this combinator, denoted $\circledcirc$, isolates the right-component, making it interact only with the left component. This is useful for modeling interaction with a closed system. We implement $\circledcirc$ as

$$p_{\mathtt{L}} \circledcirc p_{\mathtt{R}} = r_{\mathsf{E}}\langle r_{\mathtt{L}}\langle p_{\mathtt{L}}\rangle \circledast r_{\mathtt{R}}\langle p_{\mathtt{R}}\rangle\rangle,$$

where

$$\begin{array}{lll} r_{\mathsf{E}}(i) = i_{\mathsf{E}} & r_{\mathtt{L}}(o) = o_{\mathtt{L}} & r_{\mathtt{R}}(o) = o_{\mathtt{R}} \\ r_{\mathsf{E}}(o_{\mathtt{L}}) = o & r_{\mathtt{L}}(i_{\mathsf{E}}) = i & r_{\mathtt{R}}(i_{\mathsf{E}}) = ?\diamond \\ r_{\mathsf{E}}(o_{\mathtt{R}}) = !\diamond & r_{\mathtt{L}}(i_{\mathtt{R}}) = i & r_{\mathtt{R}}(i_{\mathtt{L}}) = i. \end{array}$$

6) Buffered Loop: Placing a FIFO as the right component of $\circledcirc$ yields a buffered loop combinator, denoted $\lceil \cdot \rrbracket$. To see this, let $p_{\mathsf{F}} = \mathsf{W}(\mathsf{F}(\epsilon))$, where F is defined by $\mathsf{F}(\bar{o}) \xrightarrow{i} \mathsf{F}(\bar{o}.i^{-1})$ and $\mathsf{F}(o.\bar{o}) \xrightarrow{o} \mathsf{F}(\bar{o})$. Using this process, we can implement the buffered loop combinator as follows.

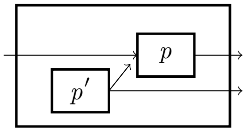
$$\lceil p \rrbracket = p \circledcirc p_{\mathsf{F}}.$$

This loop combinator avoids the compositionality issues $\lceil \rfloor$ has by storing loop messages in the FIFO p_{F} (which p can consume from at *its* leisure), instead of jamming them directly into p. This ensures that p *can* make progress on its outputs.

7) Generator: Any interactive $\mathrm{LTS}_{\mathrm{IO}}$ can be used as a *source of information* / input by never delivering input to it. We define the *generator* combinator $[\cdot\rangle$ as

$$[p\rangle = r_{\mathrm{drop}}\langle p\rangle,$$

where $r_{\mathrm{drop}}(i) = ?\diamond$ and $r_{\mathrm{drop}}(o) = o$. When used in conjunction with the binary combinators in Figure 3, we obtain *three new combinators* for introducing information into a system. These are $p \oplus [p'\rangle$ (p' streams to the environment), $p \circledcirc [p'\rangle$ (p' streams to p) and $p \otimes [p'\rangle$ (p' streams to both), the last of which can be illustrated as follows.



While for $r_{\mathrm{mute}}(i) = i$ and $r_{\mathrm{mute}}(o) = !\diamond$, one might consider including $r_{\mathrm{mute}}\langle\cdot\rangle$ as a combinator, we find that for any binary combinator $\odot$, $p \odot r_{\mathrm{mute}}\langle p'\rangle$ behaves either as p or as $r_{\mathrm{mute}}\langle p'\rangle$ (and $r_{\mathrm{mute}}\langle p'\rangle$ is semantically equivalent to $\mathsf{F}^{?}(!\diamond^{\omega})$).

B. Compositionality

Since all of the combinators presented in Figure 3 are specializations of $\circledast$ and $\langle\rangle$, we have for all of them, and their counterparts based on $\boxast$ instead of $\circledast$, the following compositionality properties.

Corollary V.1. *For each binary combinator $\odot$ in Figure 3,*

$$\begin{aligned} p_{\mathtt{L}}, p_{\mathtt{R}} \in \mathrm{PINI} &\implies p_{\mathtt{L}} \odot p_{\mathtt{R}} \in \mathrm{PINI}, \\ p_{\mathtt{L}}, p_{\mathtt{R}} \in \mathrm{PSNI} &\implies p_{\mathtt{L}} \odot p_{\mathtt{R}} \in \mathrm{PSNI}, \\ p_{\mathtt{L}}, p_{\mathtt{R}} \in \mathrm{PSNI} &\implies p_{\mathtt{L}} \odot_{\mathsf{F}} p_{\mathtt{R}} \in \mathrm{PSNI}. \end{aligned}$$

For each corresponding operator $\boxdot$ based on $\boxast$, the same holds.

For $\lceil \cdot \rrbracket$, and its counterpart based on $\boxast$,

$$\begin{aligned} p \in \mathrm{PINI} &\implies \lceil p \rrbracket \in \mathrm{PINI}, \\ p \in \mathrm{PSNI} &\implies \lceil p \rrbracket \in \mathrm{PSNI}, \\ p \in \mathrm{PSNI} &\implies \lceil p \rrbracket_{\mathsf{F}} \in \mathrm{PSNI}. \end{aligned}$$

For $[\cdot\rangle$,

$$p \in \textsc{pini} \implies [p\rangle \in \textsc{pini},$$

$$p \in \textsc{psni} \implies [p\rangle \in \textsc{psni}.$$

While the above corollary implies that any composite system consisting of PSNI components and combinators in Figure 3 satisfies PSNI regardless of fairness, the same cannot be said about PINI. When fairness is assumed, PINI fails to compose for even simple combinators; the counterexample in the proof of Theorem IV.13 also applies to products $\oplus_{\mathsf{F}}$ and $\boxplus_{\mathsf{F}}$, and we have pointed out that PINI does not behave well under cascade $\ominus_{\mathsf{F}}$ (see our rationale for considering PSNI environments when defining $\textsc{pini}_{\mathsf{E}}$ in Section III-C).

Corollary V.2. *For each* $\triangle \in \{\oplus, \boxplus, \ominus\}$,

$$p_{\mathtt{L}}, p_{\mathtt{R}} \in \textsc{pini} \;\not\Longrightarrow\; p_{\mathtt{L}} \triangle_{\mathsf{F}}\, p_{\mathtt{R}} \in \textsc{pini}.$$

C. Building Secure Systems From Parts

By the above result, we now have a rich toolset for building secure wholes from secure parts. Large systems are often developed in a modular manner, in different programming languages, and once deployed, run distributed over a network. Our combinators facilitate *end-to-end security*, and a divide-and-conquer approach to building large secure systems. Parts can be proven secure by use of language-based or language-independent enforcement mechanisms that target our security properties. Once the parts are proven secure, he have that the whole, assembled using our combinators, is secure. Combinators can be used to model the *network topology* (how the parts are "hard-wired" or nested), while routers can express data-dependent traffic routing in the network.

The combinators and our system model can also be used to formalize the concurrency semantics in a programming language, like Erlang. Furthermore, by proposing suitable primitive interactive $\textsc{lts}_{\textsc{io}}$, our combinators can *be a programming language* for writing asynchronous message-passing systems. One could, say, replace $\langle\rangle$ with $\ominus$ as a primitive, if the routing delay this introduces at the semantics level is not problematic. However, for such a language to be expressive, combinators for programmatically changing the wiring of components (e.g. switches in functional reactive programming [36] and name-passing in process algebra [22]) should be introduced.

VI. Related work

To aid in understanding the relative merits of the various models of interaction we are about to discuss, we classify $\textsc{lts}_{\textsc{io}}$ based on the interaction behavior they exhibit.

Definition VI.1 ($\textsc{lts}_{\textsc{io}}$ classification)**.** p is

1) *input value neutral* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies (\exists ?cv \bullet p' \xrightarrow{?cv}) \implies \forall v' \bullet p' \xrightarrow{?cv'}.$
2) *input neutral* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies (\exists i \bullet p' \xrightarrow{i}) \implies \forall i' \bullet p' \xrightarrow{i'}.$
3) *reactive* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies (\exists i \bullet p' \xrightarrow{i}) \implies (\nexists i', i'' \bullet p' \xrightarrow{i'.i''}) \wedge (\nexists o \bullet p' \xrightarrow{o})$
4) *productive* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies \exists a \bullet p' \xrightarrow{a}.$
5) *internally deterministic* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies \forall a, p_1', p_2' \bullet p' \xrightarrow{a} p_1' \wedge p' \xrightarrow{a} p_2' \implies p_1' = p_2'$
6) *input deterministic* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies \forall i_1, i_2 \bullet p' \xrightarrow{i_1} \wedge\, p' \xrightarrow{i_2} \implies \exists c \bullet i_1, i_2 \in \{?cv \mid v \in \mathbb{V}\}$
7) *output deterministic* iff $\forall t, p' \bullet p \xrightarrow{t} p' \implies \forall o_1, o_2 \bullet p' \xrightarrow{o_1} \wedge\, p' \xrightarrow{o_2} \implies o_1 = o_2$ $\lozenge$

Event systems [20], [28], [29], [31], [54], [57] are essentially $\textsc{lts}_{\textsc{io}}$ with no restrictions applied. Trace semantics is used as the underlying notion of behavioral equivalence. Compositionality of information flow properties, under a binary operator which implicitly wires matching communication channels internally, has been thoroughly studied in this setting in theories developed for reasoning about compositionality [28], [32], [57]. McLean [32], Zakinthinos and Lee [56] showed that *noninference*, *separability* and *perfect security* are all compositional, and McLean further showed that *generalized noninference* and *generalized noninterference* compose under product. Johnson and Thayer showed that *forward correctability* is fully compositional [25]. McCullough first demonstrated that generalized noninterference is *not* fully compositional [29]. However, Zakinthinos and Lee have shown that generalized noninterference composes under certain conditions: under a relaxed cascade [55], if every feedback loop involves at least three components [56], or if a delay component is inserted into the feedback of high events [55]. Mantel [28] derived all the above results save the last two using his modular assembly kit for security properties (MAKS). He also derived several new conditional compositionality results, and showed that a *weakened forward correctability* is compositional. Our PSNI composes under routing, product, and under cascade and feedback provided a FIFO is placed between components. Our combinators offer a structured way of composing secure systems from parts; no wiring is implicit, and the possible routes that data can take are clearly defined by structure and routers. Our properties use stream semantics as the underlying behavioral equivalence, which we have argued and demonstrated, makes more, desirable distinctions, enabling us to reject the "extortionist" program given in Section III-D.

Process calculi for security [19], [22], [23], [40], [46], [47] have $\textsc{lts}_{\textsc{io}}$ as their underlying semantics. They study the use of algebraic properties of concrete concurrency constructs. We are more abstract, providing results for $\textsc{lts}_{\textsc{io}}$ directly. We assume input totality in our framework, which induces a concurrency semantics free of output blocking, similar to mailboxes in the Actor model [1] (implemented in Erlang), message queues in JavaScript, and buffered I/O in most programming languages. Assuming input totality simplifies system composition considerably [57]. The parallel composition operator also implicitly wires channels. Bisimulation on processes is typically provided as the primary tool for behavioral reasoning. Since bisimulation is a branching-time, it makes undesired distinctions, which our behavioral equivalence avoids.

Reactive systems [12], [43], [58] are, in the sense of Definition VI.1, reactive, input neutral and productive $\textsc{lts}_{\textsc{io}}$. Bohannon et al. [12] present and contrast four stream-based possibilistic noninterference definitions, emphasizing CP-security and ID-security, and give a type-based enforcement of ID-security. While ID-security and CP-security do not exclude nondeterministic programs, ID- and CP-security are very restrictive for nondeterministic programs, rejecting programs which conceal information using nondeterministic choice, and therefore essentially becoming as strict as low observational determinism [59] or security under refinement [37]. Our definition of PINI can therefore be perceived as a more faithful generalization of PINI from [3] to nondeterministic systems, or as a generalization of ID-security to nondeterministic systems with intermediate input. We have shown that PINI does not compose under fair schedulers. The counterexamples can easily be expressed in the language of [12]; thus, ID-security does not compose fairly. While the *transducer* impression of reactive systems suggests easy composition, reactive systems are not input total; non-willingness of a component to receive can halt progress of another component that wishes to send.

Our security framework most closely resembles the LTS_{IO}- and strategy-based frameworks for possibilistic noninterference [16], [37], [42], [44]. O'Neill et al. [37] present a single-threaded programming language which LTS_{IO} is input neutral, internally- and output-deterministic. The target property is strategy-based PSNI, originally inspired by *nondeducibility on strategies* by Wittbold and Johnson [54]. Extending the language with nondeterministic choice (making their LTS_{IO} no longer deterministic), they modify PSNI to require noninterference under *refinement* (arbitrary determinization of all nondeterministic choices prior to execution). Clark and Hunt [16] instead give a *possibilistic* version of PSNI, show that it is sufficient to guarantee security under deterministic strategies to prove that a program is PSNI, and that stream strategies are sufficient if the program is internally-, input- and output-deterministc, a result used by [12]. In both of these settings, strategies are *total*; strategies are always willing to receive, and regardless of when and on which channel a program blocks on, the strategy has input available on said channel. While this may be a good fit when strategies model local memory, as demonstrated by Rafnsson et al. [42], this total strategies assumption ignores classes of realizable attacks which encode secrets in the varied presence of messages in a concurrent setting. This motivates distinguishing between sensitivity of message presence and content, as we do in the present paper, which none of the work discussed so far does, save for [42]–[44]. This idea can be traced back to Sabelfeld and Mantel [48], who study public (L), encrypted (M) and secret (H) communication channels in a concurrent setting, and Myers [33], who distinguishes between sensitivity of data structure length and content. Rafnsson et al. [42] show that PSNI composes under product. All three of [16], [37], [42] use trace semantics as a basis for behavioral equivalence. We show PSNI composes under all of our combinators, with stream semantics as the basis for behavioral equivalence, and with environments which are part of the computation model.

As an alternative to our trace- and stream-semantics, we could have chosen to express semantics as a function mapping input behaviors to output behaviors. While such a semantics would not give us a complete rule for composition, as the composition would suffer from the Brock-Ackerman anomaly [15], that issue is resolved by Widom et al. [53].

Asynchronous testing faces the same difficulties with blocking behavior as we face when putting two LTS_{IO} in interaction. Whereas Verhaard et al. [52] solve the issues by equipping a tester and the implementation under test with an input queue, our assumption of input totality effectively means our interactive systems have input queues baked in. The Input/Output Automata model [27] is very similar to our LTS_{IO} model. It has input totality as a fundamental assumption, and is designed to reason about system composition and fairness. We have not seen this model applied in research on information-flow security. The concurrency semantics induced by our combinators is similar to that employed by signal processing formalisms, like Kahn networks [26] and dataflow programming languages, e.g. Functional Reactive Programming languages [18], [36]. Indeed, our combinators are reminiscent of the signal function constructors in [36].

Finally, end-to-end security is easier to achieve if we, alongside combinators which compose secure components securely, have combinators which *repair* insecurities. Rafnsson and Sabelfeld [43] give such a combinator which puts a logarithmic bound on leaks through progress in a ID-secure program, and Askarov et al. give a combinator which puts a logarithmic bound on leaks through timing observations [5]. Secure Multi-Execution [17], [44] is a promising new technique which, through program transformation or dynamic monitoring, modifies (modestly) the semantics of any program to become that of a secure program. Devriese and Piessens [17] prove that the approach enforces timing sensitive noninterfence, while Rafnsson and Sabelfeld [44] show that, by relaxing the guarantee to PSNI, the semantics of secure programs can be modified less.

VII. Conclusion

We have presented a framework for secure composition. Coming back to the research questions from Section I, we have achieved generality along several dimensions: (i) our underlying systems are general labeled transition systems; (ii) we distinguish between the security level of message presence and content; (iii) our model incorporates environments as part of the system; (iv) our composition is facilitated by a rich set of combinators; and (v) we study both progress-sensitive and progress-insensitive security definitions. While the latter is a popular policy for practical tools, our findings point to the importance of the former in the context of secure system composition. Our findings also provide new insights on the impact of fairness for the security of system composition.

Future work includes investigation of composition in the presence of insecure components. Generalizing the results in this work and our previous results on limiting leakage by programs that satisfy PINI [43], we plan to extend our combinator set with enhanced combinators that are able to "repair" insecure components and make them readily pluggable into a secure (composed) system.

Acknowledgments: Thanks are due to Sergio Maffeis, Daniel Hedin, Daniel Schoepe, and Musard Balliu for their helpful comments. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] A. Almeida Matos, G. Boudol, and I. Castellani. Typing Noninterference for Reactive Programs. *Journal of Logic and Algebraic Programming*, 72:124–156, 2007.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, October 2008.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[6] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[7] J. Barwise and L. Moss. *Vicious Circles*. CSLI Lecture Notes. CSLI, Stanford, California, 1996.

[8] J.A. Bergstra and J.W. Klop. Fixed Point Semantics in Process Algebras. Technical Report IW 206/82, Mathematical Centre, Amsterdam, 1982.

[9] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[10] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *ICISS*, pages 48–65, 2010.

[11] A. Birgisson and A. Sabelfeld. Multi-run Security. In *ESORICS*, pages 372–391, 2011.

[12] A. Bohannon, B. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive Noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, November 2009.

[13] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Proc. of the USENIX Conference on Web Application Development*, 2010.

[14] N. Broberg, B. van Delft, and D. Sands. Paragon for Practical Programming with Information-Flow Control. In Chung chieh Shan, editor, *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.

[15] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In J. Daz and I. Ramos, editors, *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer Berlin Heidelberg, 1981.

[16] D. Clark and S. Hunt. Noninterference for Deterministic Interactive Programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, October 2008.

[17] D. Devriese and F. Piessens. Non-Interference Through Secure Multi-Execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[18] C. Elliott and P. Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, 1997.

[19] R. Focardi and R. Gorrieri. A Classification of Security Properties for Process Algebras. *J. Computer Security*, 3(1):5–33, 1995.

[20] J. D. Guttman and M. E. Nadel. What Needs Securing. In *CSFW*, pages 34–57. MITRE Corporation Press, 1988.

[21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow. Software release. Located at http://chalmerslbs.bitbucket.org/jsflow, September 2013.

[22] K. Honda, V. Vasconcelos, and N. Yoshida. Secure Information Flow as Typed Process Behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.

[23] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, January 2002.

[24] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[25] D. M. Johnson and F. J. Thayer. Security and the Composition of Machines. In *CSFW*, pages 72–89. MITRE Corporation Press, 1988.

[26] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[27] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2:219–246, 1989.

[28] H. Mantel. On the Composition of Secure Systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 81–94, May 2002.

[29] D. McCullough. Specifications for Multi-level Security and Hook-Up Property. In *Proc. IEEE Symp. on Security and Privacy*, pages 161–166, April 1987.

[30] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.

[31] D McCullough. A Hookup Theorem for Multilevel Security. *IEEE Trans. Software Eng.*, 16(6):563–568, 1990.

[32] J. McLean. A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.

[33] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[34] 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.

[35] S. Nain and M. Y. Vardi. Branching vs. Linear Time: Semantical Perspective. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, ATVA'07, pages 19–34, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[37] K. O'Neill, M. Clarkson, and S. Chong. Information-flow Security for Interactive Programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[38] Frederick Burr Opper. Alphonse a la Carte and His Friend Gaston de Table d'Hote. *New York Journal*, September 1901.

[39] D. M. R. Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[40] F. Pottier. A Simple View of Type-Secure Information Flow in the pi-Calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[41] K. V. S. Prasad. A Calculus of Broadcasting Systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995.

[42] W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing Interactive Programs. In *Proc. IEEE Computer Security Foundations Symposium*, June 2012.

[43] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2011.

[44] W. Rafnsson and A. Sabelfeld. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *Proc. IEEE Computer Security Foundations Symposium*, pages 33–48. IEEE, 2013.

[45] A. W. Roscoe. Unbounded Nondeterminism in CSP. Technical Report PRG-67, Oxford University Computing Laboratory, July 1988. in *Two papers on CSP*, Also appeared in Journal of Logic and Computation, Vol 3, No 2 pp131-172 (1993).

[46] P. Ryan. Mathematical Models of Computer Security—Tutorial Lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[47] P. Ryan and S. Schneider. Process Algebra and Non-Interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[48] A. Sabelfeld and H. Mantel. Static Confidentiality Enforcement for Distributed Programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.

[49] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[50] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/∼simonet/soft/flowcaml, July 2003.

[51] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazires. Flexible Dynamic Information Flow Control in Haskell. In *In Proceedings of the 4th Symposium on*, 2011.

[52] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On Asynchronous Testing. In Gregor von Bochmann, Rachida Dssouli, and Anindya Das, editors, *Protocol Test Systems*, volume C-11 of *IFIP Transactions*, pages 55–66. North-Holland, 1992.

[53] J. Widom, D. Gries, and F. B. Schneider. Trace-based network proof systems: Expressiveness and completeness. *ACM Trans. Program. Lang. Syst.*, 14(3):396–416, 1992.

[54] J. T. Wittbold and D. M. Johnson. Information Flow in Nondeterministic Systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 144–161, 1990.

[55] A. Zakinthinos and E. S. Lee. The Composability of Non-Interference. In *CSFW*, pages 2–8. IEEE Computer Society, 1995.

[56] A. Zakinthinos and E. S. Lee. How and Why Feedback Composition Fails. In *CSFW*, pages 95–101. IEEE Computer Society, 1996.

[57] A. Zakinthinos and E. Stewart Lee. A General Theory of Security Properties. In *IEEE Symposium on Security and Privacy*, pages 94–102, 1997.

[58] D. Zanarini, M. Jaskelioff, and A. Russo. Precise Enforcement of Confidentiality for Reactive Systems. In *Proceedings of the 26th Computer Security Foundations Symposium,* New Orleans, Louisiana, USA, 2013.

[59] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

Appendix

Theorem III.10. $p \in \textsc{psni} \implies p \in \textsc{psni}_{\mathsf{E}}$.

Proof: Assume $p \in \textsc{psni}$. Let p_1, p_2 and s_1 such that $p_1, p_2 \in \textsc{psni}$, $s_1 \in \mathbb{S}_{\mathsf{F}}$, $p_1 \simeq_\ell p_2$ and $p_1 \models p \xrightarrow{s_1}$ be arbitrary. Since $p_1 \models p \xrightarrow{s_1}$, we have $p \xrightarrow{s_1}$. Since $s_1 \in \mathbb{S}_{\mathsf{F}}$ and $p \in \textsc{psni}$, there exists a $s_{1\mathsf{P}} \in \mathbb{S}(p) \cap \mathbb{A}_\ell^\omega$ for which $s_1 \simeq_\ell s_{1\mathsf{P}}$ and $\mathsf{preserve}_{p,s_1}(\epsilon, s_{1\mathsf{P}})$. Since $p_1 \models p \xrightarrow{s_1}$, we have $p_1 \xrightarrow{{s_1}^{-1}}$. Assume ${s_1}^{-1} \in \mathbb{S}_{\mathsf{F}}$ for now (at the end of this proof, we explain how to adapt it to the scenario ${s_1}^{-1} \notin \mathbb{S}_{\mathsf{F}}$). Since $p_1 \simeq_\ell p_2$, there exists a $\hat{s}_2 \in \mathbb{S}_{\mathsf{F}}$ for which $p_2 \xrightarrow{\hat{s}_2}$ and ${s_1}^{-1} \simeq_\ell \hat{s}_2$. Since $p_2 \in \textsc{psni}$, there exists a $s_{2\mathsf{E}} \in \mathbb{S}(p_2) \cap \mathbb{A}_\ell^\omega$ for which $\hat{s}_2 \simeq_\ell s_{2\mathsf{E}}$ and $\mathsf{preserve}_{p_2,\hat{s}_2}(\epsilon, s_{2\mathsf{E}})$. To summarize, $s_{1\mathsf{P}} \simeq_\ell s_1 \simeq_\ell \hat{s}_2^{-1} \simeq_\ell {s_{2\mathsf{E}}}^{-1}$. We must show that there exists a $s_2 \in \mathbb{S}_{\mathsf{F}}$ for which $p_2 \models p \xrightarrow{s_2}$ and $s_1 \simeq_\ell s_2$. We obtain s_2 using

$$\begin{aligned}
&\mathsf{zip}(s_{\mathsf{E}}, t, \bar{o}_{\mathsf{P}}.o_{\mathsf{P}}.s_{\mathsf{P}}) \mid \bar{o}_{\mathsf{P}} \simeq_\ell \epsilon \wedge \pi(o_{\mathsf{P}}) \sqsubseteq \ell \\
&\quad = \bar{o}_{\mathsf{P}}.o_{\mathsf{P}}.\mathsf{zip}(s'_{\mathsf{E}}, t.\bar{o}_{\mathsf{P}}.o_{\mathsf{P}}, s_{\mathsf{P}}), \\
&\qquad \text{where } \hat{s}_2 \simeq_\ell t.(\bar{o}_{\mathsf{P}}.o_{\mathsf{P}})^{-1}.s'_{\mathsf{E}} \\
&\qquad\qquad \wedge p_2 \xrightarrow{(t.\bar{o}_{\mathsf{P}}.o_{\mathsf{P}})^{-1}.s'_{\mathsf{E}}} \\
&\qquad\qquad \wedge \mathsf{preserve}_{p_2,\hat{s}_2}((t.\bar{o}_{\mathsf{P}}.o_{\mathsf{P}})^{-1}, s'_{\mathsf{E}}) \\
&\mathsf{zip}(\bar{o}_{\mathsf{E}}.o_{\mathsf{E}}.s_{\mathsf{E}}, t, s_{\mathsf{P}}) \mid \bar{o}_{\mathsf{E}} \simeq_\ell \epsilon \wedge \pi(o_{\mathsf{E}}) \sqsubseteq \ell \\
&\quad = (\bar{o}_{\mathsf{E}}.o_{\mathsf{E}})^{-1}.\mathsf{zip}(s_{\mathsf{E}}, t.(\bar{o}_{\mathsf{E}}.o_{\mathsf{E}})^{-1}, s'_{\mathsf{P}}), \\
&\qquad \text{where } s_1 \simeq_\ell (t.\bar{o}_{\mathsf{E}}.o_{\mathsf{E}})^{-1}.s'_{\mathsf{P}} \\
&\qquad\qquad \wedge p \xrightarrow{t.(\bar{o}_{\mathsf{E}}.o_{\mathsf{E}})^{-1}.s'_{\mathsf{P}}} \\
&\qquad\qquad \wedge \mathsf{preserve}_{p,s_1}(t.(\bar{o}_{\mathsf{E}}.o_{\mathsf{E}})^{-1}, s'_{\mathsf{P}}) \\
&\mathsf{zip}(\tilde{o}_{\mathsf{E}}, t, \tilde{o}_{\mathsf{P}}) \qquad\qquad \mid \tilde{o}_{\mathsf{E}}^{-1} \simeq_\ell \tilde{o}_{\mathsf{P}} \simeq_\ell \epsilon \\
&\quad = \bar{o}_{\mathsf{E}}^{-1}.o_{\mathsf{P}}.\mathsf{zip}(\tilde{o}'_{\mathsf{E}}, t.\bar{o}_{\mathsf{E}}^{-1}.o_{\mathsf{P}}, \tilde{o}'_{\mathsf{P}}), \\
&\qquad \text{where } \hat{s}_2 \simeq_\ell t^{-1}.\bar{o}_{\mathsf{E}}.{o_{\mathsf{P}}}^{-1}.\tilde{o}'_{\mathsf{E}} \\
&\qquad\qquad \wedge p_2 \xrightarrow{t^{-1}.\bar{o}_{\mathsf{P}}.{o_{\mathsf{P}}}^{-1}.\tilde{o}'_{\mathsf{E}}} \\
&\qquad\qquad \wedge \mathsf{preserve}_{p_2,\hat{s}_2}(t^{-1}.\bar{o}_{\mathsf{P}}.{o_{\mathsf{P}}}^{-1}, \tilde{o}'_{\mathsf{E}}) \\
&\qquad\qquad \wedge s_1 \simeq_\ell t.\bar{o}_{\mathsf{E}}^{-1}.o_{\mathsf{P}}.\tilde{o}'_{\mathsf{P}} \\
&\qquad\qquad \wedge p \xrightarrow{t.\bar{o}_{\mathsf{E}}^{-1}.o_{\mathsf{P}}.\tilde{o}'_{\mathsf{P}}} \\
&\qquad\qquad \wedge \mathsf{preserve}_{p,s_1}(t.\bar{o}_{\mathsf{E}}^{-1}.o_{\mathsf{E}}, \tilde{o}'_{\mathsf{P}}).
\end{aligned}$$

By setting $s_2 = \mathsf{zip}(s_{2\mathsf{E}}, \epsilon, s_{1\mathsf{P}})$, we get $p_2 \models p \xrightarrow{s_2}$ and $s_1 \simeq_\ell s_2$. This can be seen by observing that the middle parameter t during each corecursive call grows to include one more observable (until it contains all observables, after which it grows with unobservables), and that $t \preceq_\ell s_1$ and $t^{-1} \preceq_\ell \hat{s}_2$.

If ${s_1}^{-1} \notin \mathbb{S}_{\mathsf{F}}$, then for the shortest prefix $t_1 \leq {s_1}^{-1}$ containing all output in ${s_1}^{-1}$, there is a $\tilde{o}_1$ for which $p_1 \xrightarrow{t_1.\tilde{o}_1}$. We then make sure to not zip beyond the last observable output in t_1 and still get the desired s_2 by replacing the last case in the definition of zip with one which evaluates to $\tilde{o}_{\mathsf{P}}$. ■

A. Combinators

1) Relaxed Cascade Feedback:

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o^{-1}} p'_{\mathrm{R}}}{p_{\mathrm{L}} \otimes p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \otimes p'_{\mathrm{R}}}\otimes^!_{\mathrm{L}} \qquad \frac{p_{\mathrm{L}} \xrightarrow{o^{-1}} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}}}{p_{\mathrm{L}} \otimes p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \otimes p'_{\mathrm{R}}}\otimes^!_{\mathrm{R}}$$

$$\frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}}}{p_{\mathrm{L}} \otimes p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \otimes p_{\mathrm{R}}}\otimes^?$$

2) Relaxed Cascade:

$$\frac{p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}}}{p_{\mathrm{L}} \obslash p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{L}} \obslash p'_{\mathrm{R}}}\obslash^! \qquad \frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}}}{p_{\mathrm{L}} \obslash p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \obslash p_{\mathrm{R}}}\obslash^?$$

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o^{-1}} p'_{\mathrm{R}}}{p_{\mathrm{L}} \obslash p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \obslash p'_{\mathrm{R}}}\obslash$$

3) Cascade:

$$\frac{p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}}}{p_{\mathrm{L}} \ominus p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{L}} \ominus p'_{\mathrm{R}}}\ominus^! \qquad \frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}}}{p_{\mathrm{L}} \ominus p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \ominus p_{\mathrm{R}}}\ominus^?$$

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o^{-1}} p'_{\mathrm{R}}}{p_{\mathrm{L}} \ominus p_{\mathrm{R}} \xrightarrow{!\diamond} p'_{\mathrm{L}} \ominus p'_{\mathrm{R}}}\ominus$$

4) Product:

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}}}{p_{\mathrm{L}} \oplus p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \oplus p_{\mathrm{R}}}\oplus^!_{\mathrm{L}} \qquad \frac{p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}}}{p_{\mathrm{L}} \oplus p_{\mathrm{R}} \xrightarrow{o} p_{\mathrm{L}} \oplus p'_{\mathrm{R}}}\oplus^!_{\mathrm{R}}$$

$$\frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{R}}}{p_{\mathrm{L}} \oplus p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \oplus p'_{\mathrm{R}}}\oplus^?$$

5) Feedback:

$$\frac{p_{\mathrm{L}} \xrightarrow{o} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o^{-1}} p'_{\mathrm{R}}}{p_{\mathrm{L}} \circledcirc p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{L}} \circledcirc p'_{\mathrm{R}}}\circledcirc^!_{\mathrm{L}} \qquad \frac{p_{\mathrm{L}} \xrightarrow{o^{-1}} p'_{\mathrm{L}} \qquad p_{\mathrm{R}} \xrightarrow{o} p'_{\mathrm{R}}}{p_{\mathrm{L}} \circledcirc p_{\mathrm{R}} \xrightarrow{!\diamond} p'_{\mathrm{L}} \circledcirc p'_{\mathrm{R}}}\circledcirc^!_{\mathrm{R}}$$

$$\frac{p_{\mathrm{L}} \xrightarrow{i} p'_{\mathrm{L}}}{p_{\mathrm{L}} \circledcirc p_{\mathrm{R}} \xrightarrow{i} p'_{\mathrm{L}} \circledcirc p_{\mathrm{R}}}\circledcirc^?$$

6) Buffered Loop: Let $\lceil p \rrbracket = \lceil p \rrbracket_\epsilon$.

$$\frac{p \xrightarrow{o} p'}{\lceil p \rrbracket_t \xrightarrow{o} \lceil p' \rrbracket_{t.o}}\lceil\rrbracket^! \qquad \frac{p \xrightarrow{i} p'}{\lceil p \rrbracket_t \xrightarrow{i} \lceil p' \rrbracket_t}\lceil\rrbracket^?$$

$$\frac{p \xrightarrow{o^{-1}} p'}{\lceil p \rrbracket_{o.t} \xrightarrow{!\diamond} \lceil p' \rrbracket_t}\lceil\rrbracket \qquad \frac{p \xrightarrow{?\diamond} p'}{\lceil p \rrbracket_\epsilon \xrightarrow{!\diamond} \lceil p' \rrbracket_\epsilon}\lceil\rrbracket^\epsilon$$