Timing-Sensitive Noninterference through Composition

Willard Rafnsson[1(✉)], Limin Jia[2(✉)], and Lujo Bauer[2(✉)]

[1] Max Planck Institute for Software Systems, Saarbrücken, Germany
willardr@mpi-sws.org
[2] Carnegie Mellon University, Pittsburgh, USA
liminjia@cmu.edu, lbauer@cmu.edu

Abstract. Sound compositional reasoning principles are the foundation for analyzing the security properties of complex systems. We present a general theory for compositional reasoning about the information-flow security of interactive discrete-timed systems. We develop a simple core—and with it, a language—of combinators, including ones that orchestrate the execution of a collection of interactive systems. We establish conditions under which timing-sensitive noninterference is preserved through composition, for each combinator in our language. To demonstrate the practicality of our theory, we model secure multi-execution (SME) using our combinators. Through this, we show that our theory makes it straightforward 1) to prove, through compositional reasoning, that complex systems are free of external timing channels, and 2) to identify sub-components that cause information leakage of a composite system.

1 Introduction

End-to-end security is the Holy Grail of information-flow security [38]. It guarantees absence of information leaks between all endpoints of a system. Enforcing end-to-end security is challenging for two main reasons. One is that modern software is large and complex: software platforms execute third-party programs, which have access to user-sensitive data and can interact with each other, the user, and the operating system. The other is that even if a software is secure, a leak may emerge when it is used as part of a larger system. This is because any security guarantee makes assumptions on the system environment, which the larger system can violate [26,28]. For instance, FlowFox [9] (by design) has a timing leak [34] since it violates an assumption that its built-in enforcement mechanism relies on to eliminate timing leaks. To address these challenges, theories for *secure composition* have been studied extensively in event systems (e.g. [24,26,42,44]), process calculi (e.g. [12,16,17,31,36,37]), transition systems (e.g. [32,35]), and thread pools (e.g. [2,25]). These theories facilitate compositional reasoning: sub-components can be analyzed in isolation, and security properties of the entire system can be derived from security properties of its components.

W. Rafnsson—Work done while the author was at Carnegie Mellon University.

M. Maffei and M. Ryan (Eds.): POST 2017, LNCS 10204, pp. 3–25, 2017.
DOI: 10.1007/978-3-662-54455-6_1

This paper investigates compositional reasoning for eliminating timing leaks in interactive systems. Timing channels are a key concern in computer security; they can be used by an adversary to reliably obtain sensitive information [6,11, 22,30], and building systems free of timing channels is a nontrivial matter. Many timing leaks are caused by the environment violating a system assumption, e.g. when the cache affects the timing behavior of an application [11,30]. Despite great interest in eliminating timing leaks [1,5,10,14,46,47], little has been done towards secure composition that eliminates timing leaks [13].

To bridge this gap, we present a *theory for secure composition of timed systems*. We first define a general model of computation, with a notion of interface that simplifies compositional reasoning. For this model of computation, we formalize our security property, *timing-sensitive noninterference*. We develop a core of combinators for composing systems, designed to be expressive yet easy to reason about formally. With it, we implement more practical combinators, i.e. a language for building composite systems, which support reasoning about process scheduling, message routing, and state. We establish compositionality results for the core of combinators, which then translates to compositionality results for the whole language of combinators. Finally, as a case study, we implement secure multi-execution (SME) [10] (an enforcement of timing-sensitive noninterference), and its variant used by FlowFox [9] (which is timing-insensitive). This demonstrates how our formalism makes it straightforward to prove noninterference of a complex system, and to trace the insecurity of a system to faulty component(s).

Our contributions are as follows:

- We define a general system model for timed asynchronous interactive systems (Sect. 3) and formalize timing-sensitive noninterference for these systems (Sect. 4).
- We develop a generic language of process combinators, with primitives for routing messages, maintaining state, scheduling processes, and wiring processes together arbitrarily (Sect. 6).
- Crucially, we identify and prove conditions under which our combinators preserve timing-sensitive noninterference under composition (Sect. 5).
- We demonstrate the practicality of our formalism and language by conducting case studies on secure multi-execution (SME) (Sect. 7).

By implementing SME, we give a complete approach for building large systems free of timing leaks: SME atomic parts, and build the rest using our language. Detailed definitions and proofs can be found in our technical report [33]. The main technical results are the theorems in Sect. 5. The culmination of our work is Fig. 2, which describes the language, and lists the compositionality result for all 28 combinators in it. We begin by motivating our approach in Sect. 2.

2 Motivation

A *system* is a whole of interacting *components*, which can themselves be systems. We refer to the system boundary as its *interface*, and what lies beyond

as its *environment*. We reason about the behavior of a system in terms of how it interacts with its environment through its interface. *Compositional reasoning* is the use of *compositionality results* on parts to derive facts about the whole. *Secure composition* is the study of compositionality results stating conditions under which a secure system can be constructed from secure components. Secure composition is a crucial challenge for securing composite systems: even if all components are secure, insecurities can arise under composition. However, obtaining compositionality results is a nontrivial matter. Each definition of security makes assumptions on how a system is used; if a composition operator—*combinator*—violates such an assumption, then its use may introduce a leak.

To motivate our work, we give examples of timing leaks that arise under composition, and outline challenges for secure composition of interactive systems.

```
sleep H;
L := 1;
```

Timing leaks. A *timing channel* is one through which an adversary learns sensitive information by observing the time at which observed effects occur. A *timing leak* is an information leak through such a channel. For instance, consider the program on the right. Here, "H" and "L" denote "high" (H, secret) and "low" (L, public) confidentiality. Upper-case variables are shared, and we refer to these as *channels*. Lower-case variables are local. We use this convention throughout the paper. The output on the public channel L is delayed as a function of the secret input channel H; by observing the timing of this event, an adversary can infer information about H. Similar to `sleep`, a loop on `h` (key-value lookup), or a branch on `h` where one branch takes longer to execute, also leaks information.

```
H(x){ sleep x }
L(x){ L := 42 }
```

Timing leaks from insecure composition. Timing leaks can arise as a result of composing secure systems. For instance, FlowFox [9] is a prototype of an information-flow secure browser, based on secure multi-execution (SME) [10, 34]. SME is a black-box enforcement that removes insecurities (including timing leaks) in any given process. It does so by running two copies, H and L, of a given process; feeding (a copy of) H and L input to the H-copy, and dropping its L output; and feeding only L input to the L-copy, and dropping its H output. Since the only source of L output (the L-copy) receives no H input, no information can leak. FlowFox implements SME on a per-event basis; inputs are queued, and the queue is serviced by first running the L-copy on the L projection of the next input, then running the H-copy on the input. Each copy finishes handling an input before passing control over to the next copy, implementing *cooperative scheduling*. However, while this approach prevents leaks to output values, the *time* at which the L-copy processes the next input depends on how long it takes for the H-copy to finish processing previous inputs. Thus, despite the process copies being run securely, and the environment just being a queue, the way the two are put together and scheduled creates a timing leak. This is illustrated by the program on the right. This program will, upon receiving a message on H with value n, sleep for n time units, and upon receiving a message on L, output 42 to L. However, running an H and L copy of this program on a queue starting with

(Hn . L0) makes the time at which the L-copy produces L42 depend on the time it takes for the H-copy to react to Hn—a function of n.

Secure composition & interaction: challenges. We have seen that a secure system can easily cause an information leak by being used in unexpected ways by its environment. While it is best that a secure system assumes as little as possible of its environment, such a security guarantee would be very strict, and might not be preserved under composition. The design of a theory for secure composition thus balances 1) environment assumptions, 2) security guarantee, and 3) choice of combinators; each of these factors dramatically impact the others. We outline some challenges that interaction introduces in this context.

```
H := H' ⊕ X   ||   x := 0|1;
                   X := x;
                   h := H;
                   L := (h ⊕ x)
```

One challenge involves the notion of environment that the security definition needs to consider. Clark and Hunt showed that for deterministic programs, an environment can wlg. be considered a fixed stream of inputs [7]. However, this does not apply to nondeterministic programs, as demonstrated by the example on the right [7]. Here, || interleaves components nondeterministically, and `0|1` is a nondeterministic choice. The right component outputs a secret bit `H`, encrypted (using XOR ⊕) with key `x`, to `L`. The output is 0 or 1, independently of `H`. The left component has no `L` outputs. Thus, both components (and the whole) are secure. Say || models hardware interleaving that is, while a priori unknown, deterministic. Then the nondeterminism in || masks a covert channel that emerges when this nondeterminism is *refined* [46] to that of the hardware. For instance, in interleaving right (line) 1, right 2, and left 1, `H = H' ⊕ x` at the time of the `L` output, so `H' ⊕ x ⊕ x = H'` is written to `L`.

```
h := H;
for b in bits(h){
 if b { H1 := 0 }
 else { H0 := 0 }
}
```
≫
```
while 1 {
 x := H1;
 L := 1
}
```
||
```
while 1 {
 x := H0;
 L := 0
}
```

The main problem is that the right component does not keep its encryption key `x` to itself. Its environment can thus, through accident or malice, *adapt* input to the right component, causing the insecurity. To capture this, "animate" environments need to be considered, e.g. *strategies* [42]. While expressive, strategies are always ready to synchronize with a system on input and output operations. Strategies thus do not consider leaks caused by *blocking communication*, which can occur under composition when components are wired together directly. Consider the program on the right. With strategies as environments, all three components are secure; the left component interacts only on `H` channels, and, since a strategy always provides input on request, the other two components output an infinite sequence of `L` 1s and 0s respectively. However, when composed with ≫, which wires its components in a synchronous pipe (i.e. any right-hand side global variable read blocks until the left-hand side writes to said variable, and vice versa), the first `L` output is 0 only if the bitwise representation of `h` contains 0.

Our assumptions. Considering systems that assume that their environment is always ready to synchronize, but that do not guarantee the same, is an incongruous basis for a theory of secure interaction. We therefore adopt an asynchronous model of interaction in our theory. We assume systems can always receive any input (making them *input total* [23,26]), and always take a step (which may produce an output message). Our timing-sensitive noninterference assumes the same of the environment. This strikes a good balance of Pt. 1-3 in the challenges section above; since interaction is nonblocking, composing components will not introduce adverse behavior. This enables rich forms of composition, and at the same time yields a clean, not too strict, notion of noninterference.

3 System model

We begin by presenting our system model, the constraints we impose on it for reasoning about interaction, and our model of time.

Process domain. We consider a model of computation for processes that interact with their environment (e.g. other processes) by receiving input or producing output. We formalize this as a pair of relations, one specifying which inputs the process can receive, the other which outputs it can produce. Let p range over processes. For $p = \langle \mathcal{R}_{\mathtt{I}}, \mathcal{R}_{\mathtt{O}} \rangle$, p can produce output o and become p' iff $\langle o, p' \rangle \in \mathcal{R}_{\mathtt{O}}$, and p can receive input i and become p' iff $\langle i, p' \rangle \in \mathcal{R}_{\mathtt{I}}$.

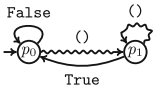
We write $\mathit{Proc}\ I\ O$ to denote the semantic domain of processes that take inputs of type I and produce outputs of type O. We define this set as the greatest fixpoint of the following equation:

$$\mathit{Proc}\ I\ O\ =\ \mathcal{P}(I \times (\mathit{Proc}\ I\ O)) \times \mathcal{P}(O \times (\mathit{Proc}\ I\ O))$$

We take the greatest fixpoint because we wish to reason about the security of processes that possibly run forever. This coalgebraic [18] approach is inspired by the interaction trees of Zanarini et al. [45]. As demonstrated below, this approach is just another way to define a labeled transition system. In contrast to more standard transition system definitions, our approach is less cumbersome since we do not need explicitly named states.

Example 1: Let $p_0 \in (\mathit{Proc}\ ()\ \mathit{Bool})$ be defined as the greatest fixpoint of the following equations.

$$p_0 = \langle \{\langle (), p_1 \rangle\}, \{\langle \mathtt{False}, p_0 \rangle\} \rangle \quad p_1 = \langle \{\langle (), p_1 \rangle\}, \{\langle \mathtt{True}, p_0 \rangle\} \rangle$$



This process outputs a Boolean indicating whether it has received a unit input since its last actuation. The graph describes this behavior; straight arrows are outputs, and wavy arrows are inputs. △

Example 2: Let $\mathbb{M} = \mathbb{C} \times \mathbb{N}$ be the set of messages. $\mathbb{C}$ is the set of channels. Let c range over $\mathbb{C}$, cn abbreviate $\langle c, n \rangle$ (message on c carrying n), and m range over $\mathbb{M}$. $\mathit{Proc}\ \mathbb{M}\ (\mathit{Maybe}\ \mathbb{M})$ is the set of message-passing processes. These can receive a message, or take a step whilst sending a message ($\mathtt{Just}\ m$) or not ($\mathtt{Nothing}$). △

Example 3: Let p, e, x, μ range over programs, expressions, variables $\mathbb{X}$, memories $(\mathbb{C} \cup \mathbb{X}) \to \mathbb{N}$ respectively. We give the semantics of our example programs as message-passing processes denotationally as the greatest fixpoint of $[\![\cdot]\!]_{\mu_0}$, where $\mathrm{img}(\mu_0) = \{0\}$ and $[\![p]\!]_\mu = \langle \mathtt{I}\ \mu\ p, \mathtt{O}\ \mu\ p \rangle$. Here, $\mathtt{I}\ \mu\ p = \{\langle cn, [\![p]\!]_{\mu[c \mapsto n]} \rangle \mid cn \in \mathbb{M}\}$, and $\mathtt{O}$ is given in full in the TR [33]. A sample of its definition:
$\mathtt{O}\ \mu\ (c := e; p) = \{\langle \mathtt{Just}\ c\mu(e), [\![p]\!]_{\mu[c \mapsto \mu(e)]} \rangle\}$ $\quad \mathtt{O}\ \mu\ (\mathtt{skip}; p) = \{\langle \mathtt{Nothing}, [\![p]\!]_\mu \rangle\}$
$\mathtt{O}\ \mu\ (\mathtt{sleep}\, e; p) = (\mu(e) = 0)\ \mathtt{?}\ \{\langle \mathtt{Nothing}, [\![p]\!]_\mu \rangle\}\ \mathtt{:}\ \{\langle \mathtt{Nothing}, [\![\mathtt{sleep}\, (\mu(e) - 1); p]\!]_\mu \rangle\}$

Inputs update memory without stepping the program, and each step produces output `Nothing` except in the global variable assignment case. △

Process behavior. We reason about the behavior of a process strictly in terms of its inputs and outputs. Process inputs and outputs thus constitute its interface to its environment. Let $p = \langle \mathcal{R}_\mathtt{I}, \mathcal{R}_\mathtt{O} \rangle$. We write $p \xrightarrow{o} p'$ iff $\langle o, p' \rangle \in \mathcal{R}_\mathtt{O}$, and $p \overset{i}{\rightsquigarrow} p'$ iff $\langle i, p' \rangle \in \mathcal{R}_\mathtt{I}$. We write $p \xrightarrow{o}$ iff $\exists p' \centerdot p \xrightarrow{o} p'$, and $p \overset{i}{\rightsquigarrow}$ iff $\exists p' \centerdot p \overset{i}{\rightsquigarrow} p'$.

Example: Process p_0 from Example 1 is the least process satisfying $p_0 \overset{()}{\rightsquigarrow} p_1, p_0 \xrightarrow{\mathtt{False}} p_0$, $p_1 \overset{()}{\rightsquigarrow} p_1$, and $p_1 \xrightarrow{\mathtt{True}} p_0$. △

A process thus defines a labeled transition system with input-output effects as labels. We define $\mathit{Eff}\ I\ O$, the set of I-input and O-output effects, as follows.

$$\mathit{Eff}\ I\ O = (\{?\} \times I) \cup (\{!\} \times O).$$

Let e range over effects. We write $?i$, $!o$ as shorthand for $\langle ?, i \rangle$, $\langle !, o \rangle$ respectively. The transition relation $(\cdots\!\!\to)$ is then: $p \overset{?i}{\cdots\!\!\to} p'$ iff $p \overset{i}{\rightsquigarrow} p'$, and $p \overset{!o}{\cdots\!\!\to} p'$ iff $p \xrightarrow{o} p'$.

We consider the sequences of effects performed by a process. Let t range over *traces*, i.e. finite words, and s range over *streams*, i.e. infinite words. Let ϵ denote the empty word, and "." concatenation. Let S^* and S^ω be the set of finite and infinite words over set S. For each p, let $p \overset{\epsilon}{\cdots\!\!\to} p$, let $p \overset{e.t}{\cdots\!\!\to} p''$ iff $\exists p' \centerdot p \overset{e}{\cdots\!\!\to} p' \overset{t}{\cdots\!\!\to} p''$, and let $p \overset{t}{\cdots\!\!\to}$ iff $\exists p' \centerdot p \overset{t}{\cdots\!\!\to} p'$. Likewise, $p \overset{e.s}{\cdots\!\!\to}$ iff $\exists p' \centerdot p \overset{e}{\cdots\!\!\to} p' \overset{s}{\cdots\!\!\to}$.

Example: For p_0 from Example 1, we have $p_0 \overset{\mathtt{!False}}{\cdots\!\!\to} p_0 \overset{\mathtt{?()}}{\cdots\!\!\to} p_1 \overset{\mathtt{?()}}{\cdots\!\!\to} p_1 \overset{\mathtt{!True}}{\cdots\!\!\to} p_0$. Let $t = \mathtt{!False}\,.\,\mathtt{?()}\,.\,\mathtt{?()}\,.\,\mathtt{!True}$. Then $p_0 \overset{t}{\cdots\!\!\to}$, and $p_0 \overset{t^\omega}{\cdots\!\!\to}$. △

Interactive processes. Since we are interested in the interaction of processes, the model of interaction that we consider is of central importance. Ours has two properties. The first property is that processes are *productive*: they can always produce output. This is intuitive, since outputs represent work performed by the process, and the processes that we consider can always perform work (this is similar to e.g. weakly time alive in tSPA [13]). The second property is that processes are *input total* [27] (a.k.a. *input enabled* [23]): processes can always receive any input. This makes communication asynchronous, which simplifies compositional reasoning [26,44] since processes cannot block their environment. This assumption is typically achieved by queuing input or by buffering channels.

Definition 1 (interactive process): p is *interactive* iff

1. $\exists o \centerdot \exists p' \centerdot p \xrightarrow{o} p' \wedge p'$ is interactive. (productive)
2. $\forall i \centerdot \exists p' \centerdot p \overset{i}{\rightsquigarrow} p' \wedge p'$ is interactive. (input total) ◊

An interactive process can always take action, and always accept any input. Interaction between an interactive process and its environment thus never blocks

or stops; to reason about such behavior, it must be modeled, making its effect, e.g. on timing, explicit. We define $\mathit{IProc}\ I\ O$, the set of interactive $\mathit{Proc}\ I\ O$, as

$$\mathit{IProc}\ I\ O = \{p \in \mathit{Proc}\ I\ O \mid p \text{ is interactive}\}.$$

Example: For p_0 from Example 1, $p_0 \in (\mathit{IProc}\ ()\ \mathit{Bool})$, i.e. p_0 is interactive. If we remove a transition from p_0, the resulting process will not be interactive. For instance, removing $p_1 \overset{()}{\rightsquigarrow} p_1$ yields a process that is not input total, as it cannot receive more than one `()` between actuations. △

Timing. We use a *discrete* model of time and conflate transitions with time similar to prior work (e.g. [5,10,13,20]). Our formalism times the *work* performed by a process, which is producing output, since systems receive input asynchronously. As a result, *outputs are timed, and inputs are untimed.* Each output takes one unit of time, and inputs arrive at units of time by arriving between outputs.

Example: For p_0 from Example 1, in $p_0 \xrightarrow{\texttt{False}} p_0 \overset{()}{\rightsquigarrow} p_1 \overset{()}{\rightsquigarrow} p_1 \xrightarrow{\texttt{True}} p_0$, the process performed two time units of work (one per output). Between the outputs, the environment provided two inputs without the process itself performing work. △

To motivate this timing model, consider an operating system process p, waiting to be scheduled. While p is idle, another process can write to p's memory, thus delivering an input to p. p performed no work in receiving it; however, the writing process (and thus the computer) performs work producing said input and thus the passing of time in this exchange is accounted for in the actions of processes. This model of time makes explicit, in the transition history of the whole system, the time that passes while processes wait. This simplifies reasoning.

Our work makes no restriction on how fine the discretization of time is; it can be chosen as needed when a process is being modeled (e.g. to a constant factor of the motherboard clock frequency).

4 Security definition

Based on a notion of attacker observation, we formalize absence of attacks as a semantic security property: timing-sensitive noninterference.

Threat model. We consider an attacker that observes public process inputs and outputs, as well as how much time passes between their occurrence. We assume that the attacker knows how the process is defined. Our goal is to facilitate building processes that preserve *confidentiality*: an attacker that interacts with such a process through its interface learns nothing about inputs to the process that the attacker is not allowed to know.

Observables. We formalize what each *principal* is allowed to know by means of a security lattice denoted $\langle \mathcal{L}, \sqsubseteq \rangle$, where $\mathcal{L}$ is a set, $(\sqsubseteq) \subseteq \mathcal{L} \times \mathcal{L}$ is a partial order relation over $\mathcal{L}$, and every pair of elements ℓ and ℓ' in $\mathcal{L}$ have a least upper bound $\ell \sqcup \ell'$ and greatest lower bound $\ell \sqcap \ell'$. Any principal, including the attacker, is assumed to be associated with an element of $\mathcal{L}$, and $(\sqsubseteq)$ expresses the relative privileges of principals. Information from a principal may only flow to more privileged principals (i.e. only upwards in the security lattice). We refer to elements

of $\mathcal{L}$ as *security levels*, expressing levels of confidentiality. In examples, we use a two-point lattice $\langle \mathcal{L}_{\mathsf{LH}}, \sqsubseteq \rangle$, where $\mathcal{L}_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$ and $(\sqsubseteq) = \{\langle \mathsf{L}, \mathsf{L}\rangle, \langle \mathsf{L}, \mathsf{H}\rangle, \langle \mathsf{H}, \mathsf{H}\rangle\}$.

We express what each principal observes in inputs and outputs by defining, for each principal, which values are observably equivalent. To identify values that are unobservable to a principal, we introduce a distinguished value $\bullet$ that we assume is not an element of any value space. Any value observably equivalent to $\bullet$ is considered unobservable. Let $V_\bullet = V \cup \{\bullet\}$, and let $v_\bullet$ range over $V_\bullet$. Let $Eq_\bullet\, V$ be the set of equivalence relations over $V_\bullet$.

Definition 2:
$\mathcal{R} : \mathcal{L} \to Eq_\bullet V$ is an *$\mathcal{L}$-equivalence over V* iff $\forall \ell, \ell' . \ell \sqsubseteq \ell' \implies \mathcal{R}_{\ell'} \subseteq \mathcal{R}_\ell$. We say $v_\bullet$ is ℓ-$\mathcal{R}$-equivalent with $v'_\bullet$ iff $\langle v_\bullet, v'_\bullet \rangle \in \mathcal{R}_\ell$. ♢

We define the set $ObsEq\ \mathcal{L}\ V$ of $\mathcal{L}$ -equivalences over V as

$$ObsEq\ \mathcal{L}\ V = \{\mathcal{R} : \mathcal{L} \to Eq_\bullet V \mid \mathcal{R} \text{ is an } \mathcal{L}\text{-equivalence over } V\}.$$

We will consider different $\mathcal{L}$-equivalences over the same set V at the same time; when $\mathcal{L}$ is clear from the context, we let $(\overset{V}{=})$, $(\overset{V}{=})$, and $(\overset{V}{=})$ range over $ObsEq\ \mathcal{L}\ V$.

Example 4: For p_0 from Example 1, say L observes the Boolean outputs, but does not observe the inputs. We capture this as $\mathcal{L}_{\mathsf{HL}}$-equivalences as follows.

$$(\overset{()}{=}_{\mathsf{H}}) = \{\langle (), () \rangle, \langle \bullet, \bullet \rangle\} \qquad (\overset{()}{=}_{\mathsf{L}}) = (\overset{()}{=}_{\mathsf{H}}) \cup \{\langle (), \bullet \rangle, \langle \bullet, () \rangle\}$$
$$(\overset{Bool}{=}_{\mathsf{H}}) = (\overset{Bool}{=}_{\mathsf{L}}) = \{\langle \mathtt{True}, \mathtt{True} \rangle, \langle \mathtt{False}, \mathtt{False} \rangle, \langle \bullet, \bullet \rangle\}$$

Since $() \overset{()}{=}_{\mathsf{L}} \bullet$, L cannot distinguish $()$ from $\bullet$, making *presence* of input to p_0 unobservable to L. The H principal, however, can distinguish all values. △

Example 5: Revisiting Example 2, assume a mapping from channels to security levels $\mathsf{lev} : \mathbb{C} \to \mathcal{L}$. We express that an ℓ-observer observes messages over $(\sqsubseteq \ell)$ channels, using the following projection function $\mathsf{obs} : \mathcal{L} \to \mathbb{M} \to \mathit{Maybe}\ \mathbb{M}$.

$$\mathsf{obs}_\ell\ cn = \mathsf{lev}(c) \sqsubseteq \ell\ \mathtt{?}\ \mathtt{Just}\ cn\ \mathtt{:}\ \mathtt{Nothing}$$

We define two messages to be ℓ-equivalent iff what an ℓ-observer observes in them is the same. That is, for all ℓ, $(\overset{\mathbb{M}}{=}_\ell)$ is the least equivalence relation satisfying

$$m \overset{\mathbb{M}}{=}_\ell m' \quad \text{iff} \quad (\mathsf{obs}_\ell\ m) = (\mathsf{obs}_\ell\ m') \qquad m \overset{\mathbb{M}}{=}_\ell \bullet \quad \text{iff} \quad (\mathsf{obs}_\ell\ m) = \mathtt{Nothing}$$

Since $(\mathsf{obs}_{\mathsf{L}}\ m) = \mathtt{Nothing}$ for messages on H channels, $m \overset{\mathbb{M}}{=}_{\mathsf{L}} \bullet$, meaning L will not observe the presence of such inputs. We let $\dot{V} = V^\bullet = \mathit{Maybe}\ V$, and let $\dot{v}$ range over $\dot{V}$. Let $\mathsf{eqmaybe}(L, \overset{V}{=})$ be the least equivalence relation $(\overset{\dot{V}}{=})$ satisfying

$$(\mathtt{Just}\, v) \overset{\dot{V}}{=}_\ell (\mathtt{Just}\, v') \text{ iff } v \overset{V}{=}_\ell v' \quad (\mathtt{Just}\, v) \overset{\dot{V}}{=}_\ell \bullet \text{ iff } v \overset{V}{=}_\ell \bullet \quad \mathtt{Nothing} \overset{\dot{V}}{=}_\ell \bullet \text{ iff } \ell \notin L.$$

L is the set of principals that can distinguish $\mathtt{Nothing}$ from unobservable $\mathtt{Just}\ v$. We compare outputs with $(\overset{\dot{\mathbb{M}}}{=}) = \mathsf{eqmaybe}(\emptyset, \overset{\mathbb{M}}{=})$. △

Noninterference. An interactive process is noninterfering iff unobservable input does *not interfere* with observable output. An attacker observing public effects of such a process thus cannot infer any knowledge of its secret inputs.

To motivate our formalization of noninterference, consider the set of streams a process can perform. Each time the process performs an effect, this set shrinks to the set of streams prefixed by the effects that the process has performed so far. To violate noninterference, a process must receive secret input that renders some public behavior impossible. Our formalization stipulates that a process can, through its own actions, avoid states where it can be influenced by its environment in this manner. We achieve this by requiring that, at any point of the execution, secret input can be inserted, changed or removed, without affecting the ability of the process to perform a given stream of observable effects.

Definition 3: $\mathcal{R} \subseteq (\mathit{Eff}\ I\ O)^\omega \times (\mathit{IProc}\ I\ O)$ is an ℓ-($\overset{I}{=}$)-($\overset{O}{=}$)-*simulation* iff

1. $\forall \langle ?i\,.\,s, p\rangle \in \mathcal{R}\,.\,i \overset{I}{=}_\ell \bullet \implies \langle s, p\rangle \in \mathcal{R}$.
2. $\forall \langle\ \ \ s, p\rangle \in \mathcal{R}\,.\,\forall i \overset{I}{=}_\ell \bullet\,.\,\exists p'\,.\,p \overset{i}{\rightsquigarrow} p' \wedge \langle s, p'\rangle \in \mathcal{R}$.
3. $\forall \langle ?i\,.\,s, p\rangle \in \mathcal{R}\,.\,\forall i' \overset{I}{=}_\ell i\,.\,\exists p'\,.\,p \overset{i'}{\rightsquigarrow} p' \wedge \langle s, p'\rangle \in \mathcal{R}$.
4. $\forall \langle !o\,.\,s, p\rangle \in \mathcal{R}\,.\,\exists o' \overset{O}{=}_\ell o\,.\,\exists p'\,.\,p \xrightarrow{o'} p' \wedge \langle s, p'\rangle \in \mathcal{R}$.

p ℓ-($\overset{I}{=}$)-($\overset{O}{=}$)-*simulates* s, written $s \left\langle \overset{I}{=}\overset{O}{=} \right]_\ell p$, iff $\langle s, p\rangle$ is in some ℓ-($\overset{I}{=}$)-($\overset{O}{=}$)-simulation. ◊

Definition 4 (noninterfering p):
p is ($\overset{I}{=}$)-($\overset{O}{=}$)-*noninterfering*, written $p \in \mathrm{NI}(\overset{I}{=}, \overset{O}{=})$, iff

$$\forall \ell\,.\,\forall s\,.\,p \overset{s}{\cdots\!\!\rightarrow} \implies s \left\langle \overset{I}{=}\overset{O}{=} \right]_\ell p. \qquad \lozenge$$

This coinductive definition requires that, for each ℓ, and for each stream s that p can perform, p must ℓ-simulate s (Definition 3). For p to ℓ-simulate s, p needs to satisfy four conditions. Pt. 1 and 2 deal with unobservable input (and are therefore vacuously true when I has no values unobservable to ℓ). Pt. 1 states that if $s = ?i\,.\,s'$, the presence of i in s is not required for p to be able to simulate s. Similarly, Pt. 2 states that the absence of i is not required either. Pt. 3 and 4 deal with observable as well as unobservable effects. Pt. 3 states that if $s = ?i\,.\,s'$, p must simulate s' after any $i' \overset{I}{=}_\ell i$ has been inserted into p, i.e. unobservably changing the next input will not prevent the process from simulating the rest. Finally, Pt. 4 states that if $s = !o\,.\,s'$, p must be capable of producing some $o' \overset{O}{=}_\ell o$ and subsequently simulate s'.

This definition is *timing-sensitive*; p must be able to simulate s without inserting, observably changing, or deleting output, or any observable input. Thus, p must be able to preserve the timing of public effects in s.

`L := H` $= p_{\mathrm{E}}$

`if H {L := 1} {L := 0}` $= p_{\mathrm{I}}$

`if H {L := 1}` $= p_{\mathrm{P}}$

`sleep H; L := 1` $= p_{\mathrm{T}}$

`skip` $= p_1$

`sleep 100; L := 1` $= p_2$

Example: The top four programs on the right violate $\overset{\mathrm{M}}{=}$-NI. $[\![p_{\mathrm{E}}]\!]_{\mu_0} = p$ has an *explicit flow*. Assume $p \in \mathrm{NI}(\overset{\mathrm{M}}{=}, \overset{\dot{\mathrm{M}}}{=})$, with $(\overset{\mathrm{M}}{=})$ and $(\overset{\dot{\mathrm{M}}}{=})$ as defined in Example 5. Let $s = !\mathtt{Just}\ \mathtt{L0}\,.\,(!\mathtt{Nothing})^\omega$. Since $p \overset{s}{\cdots\!\!\rightarrow}$, $s \left\langle \overset{\mathrm{M}}{=}\overset{\dot{\mathrm{M}}}{=} \right]_{\mathtt{L}} p$ must hold. So there must exist a $\mathtt{L}$-($\overset{\mathrm{M}}{=}$)-($\overset{\dot{\mathrm{M}}}{=}$)-simulation $\mathcal{R}$ for which $\langle s, p\rangle \in \mathcal{R}$. By Definition 3 Pt.

2, since $\texttt{?H1} \overset{M}{=}_{\text{L}} \bullet$, $\langle s, p' \rangle \in \mathcal{R}$ where $p \overset{\texttt{?H1}}{\rightsquigarrow} p'$. However, $\texttt{Just L1} \overset{\hat{M}}{\neq}_{\text{L}} \texttt{Just L0}$, is the only output p' can perform, so $\mathcal{R}$ violates Pt. 4, contradicting $s \langle \overset{M}{=} \overset{\hat{M}}{=}]_{\text{L}} p$. Thus $p \notin \text{NI}(\overset{M}{=}, \overset{\hat{M}}{=})$. $[\![\rho_{\text{I}}]\!]_{\mu_0}$ has an *implicit flow*; the proof that it violates NI is nearly identical. $[\![\rho_{\text{P}}]\!]_{\mu_0}$ has a *progress leak*. $[\![\rho_{\text{P}}]\!]_{\mu_0}$ can perform $s = (\texttt{!Nothing})^\omega$; if ?H1 is inserted, $[\![\rho_{\text{P}}]\!]_{\mu_0}$ eventually outputs $\texttt{Just L1} \overset{\hat{M}}{\neq}_{\text{L}} \texttt{Nothing}$. $[\![\rho_{\text{T}}]\!]_{\mu_0}$ has a *timing leak*. Let $s = \texttt{!Nothing.!Just L1.}(\texttt{!Nothing})^\omega$. $[\![\rho_{\text{T}}]\!]_{\mu_0}$ can perform s. However, inserting ?H42 delays !Just L1.

The last two programs satisfy NI. Let $p = [\![\rho_1]\!]_{\mu_0}$, let s such that $p \overset{s}{\Longrightarrow}$, and let $(\overset{M}{=})$ and $(\overset{\hat{M}}{=})$ be as given in Example 5. We show that $s \langle \overset{M}{=} \overset{\hat{M}}{=}]_\ell p$, for all ℓ. Let $\mathcal{R} = \{\langle \hat{s}, [\![\texttt{skip}]\!]_\mu \rangle \mid \hat{s} \in (\mathit{Eff}\ \mathbb{M}\ \{\texttt{Nothing}\})^\omega\}$. Since $p = [\![\rho_1]\!]_{\mu_0}$, $p = [\![\rho_1]\!]_{\mu_0}$, $s \in (\mathit{Eff}\ \mathbb{M}\ \{\texttt{Nothing}\})^\omega$, so $\langle s, p \rangle \in \mathcal{R}$. The proof that $\mathcal{R}$ is a ℓ-$(\overset{M}{=})$-$(\overset{\hat{M}}{=})$ -simulation involves picking any $\langle s, [\![\texttt{skip}]\!]_\mu \rangle \in \mathcal{R}$, and showing that Pt. 1-4 of Definition 3 hold (using that for all μ, c and n, $[\![\texttt{skip}]\!]_\mu \overset{cn}{\rightsquigarrow} [\![\texttt{skip}]\!]_{\mu[c \mapsto n]}$ and $[\![\texttt{skip}]\!]_\mu \xrightarrow{\texttt{Nothing}} [\![\texttt{skip}]\!]_\mu$). Similarly, $[\![\rho_2]\!]_{\mu_0}$ satisfies NI, since it ignores inputs. △

5 Combinator core

We develop a core of combinators for composing processes, presented in Figure 1. The core is expressive yet easy to reason about; instead of striving for a minimal core, we designed this core such that each combinator in it embodies a clearly-defined responsibility. We prove that the core combinators are all security preserving; composing secure components yields a secure whole. We use this core to implement a language of security-preserving combinators, in Sect. 6.

Core. Each core combinator in Figure 1 is a function that takes a set of processes as parameter and returns a new process. The combinators are designed for building secure composites using secure parts. By introducing a primitive process, e.g. $[\![\texttt{skip}]\!]_{\mu_0}$, the core becomes a core language for implementing processes.

Fig. 1: Core combinators

The `map` combinator transforms incoming and outgoing messages. With `map`, we can tag messages, providing means of routing messages. The `sta` combinator maintains state, updating and forwarding it upon receiving input and output. With `sta`, we can implement queues and counters. The compositionality results for `sta` enable reasoning about the security of state maintained by a system. The `swi` combinator maintains a Boolean state that determines whether the given process is "on" or "off". In (`swi` b p), b determines whether or not p is running. If $b = \texttt{False}$, then p is "off". Thus, when (`swi` b p) is tasked for output, it merely produces `Nothing` without touching p (by rule (Swi↑)). With `swi`, we can implement scheduling strategies and process termination, facilitating secure implementation of runtime systems. Notice that in (`swi False` p), p receives input. This lets the environment write values into p's memory while p is waiting. The

`maybe` combinator ignores non-value inputs. That is, (`maybe` p) ignores `Nothing` input, and inputs i to p on receiving `Just` i (rule (MAP?)). With `maybe`, we can, together with `map`, filter incoming messages, removing those not intended to the process. The `par` combinator executes two processes in parallel. With `par`, composite processes can be built. The `loop` combinator feeds process output back in as input, which can orchestrate interactions between subcomponents.

Compositionality of core. Our main results are compositionality results for each core combinator, stating how each preserves security. The proofs are by coinduction. We sketch the proof for `map`; the other proofs are similar.

map. The `map` combinator preserves the security of its given process as long as its given functions do not introduce insecurities. We identify two ways a function can introduce insecurities. The former is when a function maps observably equivalent values to observably different values. Functions, that do not, are noninterfering. The latter is when the input function maps an unobservable input to an observable one. Functions, that do not, are unobservable-preserving.

Definition 5 (noninterference): forall $f : I \to O$, $(\overset{I}{=})$, and $(\overset{O}{=})$, f is $(\overset{I}{=})$-$(\overset{O}{=})$-noninterfering, written $f \in \mathrm{NI}(\overset{I}{=}, \overset{O}{=})$, iff $\forall \ell \,.\, \forall i, i' \,.\, i \overset{I}{=}_\ell i' \implies (f\ i) \overset{O}{=}_\ell (f\ i')$ ♢

Definition 6 (unobservable-preserving): forall $f : I \to O$, $(\overset{I}{=})$, and $(\overset{O}{=})$, f is $(\overset{I}{=})$-$(\overset{O}{=})$-unobservable-preserving, written $f \in \mathrm{PU}(\overset{I}{=}, \overset{O}{=})$, iff $\forall \ell \,.\, \forall i \,.\, i \overset{I}{=}_\ell \bullet \implies (f\ i) \overset{O}{=}_\ell \bullet$. ♢

Theorem 1 (`map`): forall $p \in \mathit{IProc}\ I'\ O$, $f : I \to I'$, $g : O \to O'$, $(\overset{I}{=})$, $(\overset{I'}{=})$, $(\overset{O}{=})$, and $(\overset{O'}{=})$, if $p \in \mathrm{NI}(\overset{I'}{=}, \overset{O}{=})$, $f \in \mathrm{NI}(\overset{I}{=}, \overset{I'}{=}) \cap \mathrm{PU}(\overset{I}{=}, \overset{I'}{=})$ and $g \in \mathrm{NI}(\overset{O}{=}, \overset{O'}{=})$, then $(\texttt{map}\ f\ g\ p) \in \mathrm{NI}(\overset{I}{=}, \overset{O'}{=})$.

Proof sketch. Pick everything universally quantified in Theorem 1, satisfying the stated assumptions. By Definition 4, the proof of $(\texttt{map}\ f\ g\ p) \in \mathrm{NI}(\overset{I}{=}, \overset{O'}{=})$ is carried out in two steps: given ℓ and s such that $(\texttt{map}\ f\ g\ p) \overset{s}{\dashrightarrow}$, the first step is to find a relation $\mathcal{R} \subseteq (\mathit{Eff}\ I\ O)^\omega \times (\mathit{IProc}\ I\ O)$ that relates s and $(\texttt{map}\ f\ g\ p)$; the second step is to prove that $\mathcal{R}$ is a ℓ-stream-simulation (Definition 3). Let

$$\mathcal{R} = \{\langle \hat{s}, \texttt{map}\ f\ g\ \hat{p}\rangle \mid \exists s' \,.\, s' \langle \overset{I' O}{=\!=}] \hat{p} \wedge (\texttt{map}\ f\ g\ s') \overset{s}{\dashrightarrow}\}.$$

Here, $(\texttt{map}\ f\ g\ s') \overset{\hat{s}}{\dashrightarrow}$ relates an activity of the composite process to the activity of the inner process; $(\texttt{map}\ f\ g\ s') \overset{\hat{s}}{\dashrightarrow}$ iff for some process $\hat{p}$, $(\texttt{map}\ f\ g\ \hat{p}) \overset{\hat{s}}{\dashrightarrow}$ and $\hat{p} \overset{s'}{\dashrightarrow}$ (thus s' is what $\hat{p}$ did as $(\texttt{map}\ f\ g\ \hat{p})$ computed $\hat{s}$). To see that $\langle s, \texttt{map}\ f\ g\ p\rangle \in \mathcal{R}$, construct s' from the proof of $(\texttt{map}\ f\ g\ p) \overset{s}{\dashrightarrow}$ such that $p \overset{s'}{\dashrightarrow}$ and $(\texttt{map}\ f\ g\ s') \overset{s}{\dashrightarrow}$. Then invoke $p \in \mathrm{NI}(\overset{I'}{=}, \overset{O}{=})$ to establish $s' \langle \overset{I' O}{=\!=}] p$. The proof that $\mathcal{R}$ is a ℓ-stream-simulation involves picking any pair $\langle \hat{s}, \texttt{map}\ f\ g\ \hat{p}\rangle \in \mathcal{R}$, and showing that points 1) through 4) of Definition 3 hold through case analysis. ■

sta. The compositionality result for `sta` states how to introduce state into a large system without violating security: `sta` preserves the security of a given process as long as the state update functions do not introduce insecurities. These functions can do so in two ways: using unobservable parts of input and state to observably update state, and observably updating state upon receiving an unobservable input. Functions that do not do this are noninterfering and equivalence-preserving.

Definition 7: forall
$f : I \to V \to O$, $(\overset{I}{=})$, $(\overset{V}{=})$, and $(\overset{O}{=})$, f is $(\overset{I}{=})$-$(\overset{V}{=})$-$(\overset{O}{=})$-noninterfering, $f \in \mathrm{NI}(\overset{I}{=}, \overset{V}{=}, \overset{O}{=})$, iff $\forall \ell \,.\, \forall i, i' \,.\, i \overset{I}{=}_\ell i' \implies \forall v, v' \,.\, v \overset{V}{=}_\ell v' \implies (f\, i\, v) \overset{O}{=}_\ell (f\, i'\, v')$. ♢

Definition 8 (equivalence-preserving): forall $f : I \to V \to V$, $(\overset{I}{=})$, and $(\overset{V}{=})$, f is $(\overset{I}{=})$-$(\overset{V}{=})$-equivalence-preserving, $f \in \mathrm{PE}(\overset{I}{=}, \overset{V}{=})$, iff $\forall \ell \,.\, \forall i \,.\, i \overset{I}{=}_\ell \bullet \implies \forall v \,.\, (f\ i\ v) \overset{V}{=}_\ell v$ ♢

Theorem 2 (sta): forall p, f, g, v, $(\overset{I}{=})$, $(\overset{V}{=})$, and $(\overset{O}{=})$, if $p \in \mathrm{NI}(\overset{V\times I}{=}, \overset{O}{=})$, $g \in \mathrm{NI}(\overset{O}{=}, \overset{V}{=}, \overset{V}{=})$, and $f \in \mathrm{NI}(\overset{I}{=}, \overset{V}{=}, \overset{V}{=}) \cap \mathrm{PE}(\overset{I}{=}, \overset{V}{=})$, then $(\mathtt{sta}\ f\ g\ v\ p) \in \mathrm{NI}(\overset{I}{=}, \overset{V\times O}{=})$, where $(\overset{V\times O}{=}) = \mathsf{eqpair}(\overset{V}{=}, \overset{O}{=})$ and $(\overset{V\times I}{=}) = \mathsf{eqpair}\bullet\mathtt{R}(\overset{V}{=}, \overset{I}{=})$. □

Let $\mathsf{eqpair}(\overset{A}{=}, \overset{B}{=})$, $\mathsf{eqpair}\bullet\mathtt{LR}(\overset{A}{=}, \overset{B}{=})$ and $\mathsf{eqpair}\bullet\mathtt{R}(\overset{A}{=}, \overset{B}{=})$ denote the least equivalence relation $(\overset{A\times B}{=})$

$$\langle a, b\rangle \overset{A\times B}{=}_\ell \langle a', b'\rangle \quad \text{iff } a \overset{B}{=}_\ell a' \text{ and } b \overset{B}{=}_\ell b' \quad (1)$$
$$\langle a, b\rangle \overset{A\times B}{=}_\ell \bullet \quad \text{iff } a \overset{A}{=}_\ell \bullet \text{ and } b \overset{B}{=}_\ell \bullet \quad (2)$$
$$\langle a, b\rangle \overset{A\times B}{=}_\ell \bullet \quad \text{iff } b \overset{B}{=}_\ell \bullet \quad (3)$$

satisfying (1), (1) and (2), and (1) and (3) respectively. Here, $\mathsf{eqpair}(\overset{A}{=}, \overset{B}{=})$ is componentwise observable equivalence, with observable presence, and $\mathsf{eqpair}\bullet\mathtt{LR}(\overset{A}{=}, \overset{B}{=})$,$\mathsf{eqpair}\bullet\mathtt{LR}(\overset{A}{=}, \overset{B}{=})$ weaken $\mathsf{eqpair}(\overset{A}{=}, \overset{B}{=})$ by making the presence of pairs unobservable when both, or the right, components are, respectively.

swi The compositionality result for `swi` states how to switch processes (to e.g. implement schedulers) securely: `swi` preserves security as long as unobservables cannot affect the switch state, and, as a result, stagger observable process output. We consider two ways to meet this restriction. One way this restriction is met for a principal ℓ is for ℓ to fully observe the switch state; that way, no information can ever leak to ℓ through it. Such observers are *aware* of the value of the switch.

Definition 9 (awareness): forall ℓ and $(\overset{V}{=})$, ℓ is aware of v under $(\overset{V}{=})$, $\ell \in \mathsf{A}(v, \overset{V}{=})$, iff $\forall \dot{v} \,.\, v \overset{V}{=}_\ell \dot{v} \implies v = \dot{v}$. ♢

For instance, $\mathsf{A}(\mathtt{True}, \overset{Bool}{=})$ is the set of principals who can distinguish `True` from every other value in *Bool* (i.e. `False`). In the case of `swi`, those observers observe the switch signals, and thus the switch state. Since the switch state can be inferred by knowing whether the switched process took a step, only $\mathsf{A}(\mathtt{True}, \overset{Bool}{=})$ are allowed to distinguish `Nothing` output from `Just` o for unobservable o. Relation $(\overset{\dot{o}}{=}) = \mathsf{eqmaybe}(\mathsf{A}(\mathtt{True}, \overset{Bool}{=}), \overset{O}{=})$ achieves this. Another way this restriction is met for a principal ℓ is if all process output is ℓ-unobservable. Then, ℓ is *oblivious* to p.

Definition 10
(oblivious): forall ℓ, $p \in \mathit{IProc}\ I\ O$, and $(\overset{O}{=})$, ℓ is $(\overset{O}{=})$-*oblivious* to p, $\ell \in \mathsf{O}(p, \overset{O}{=})$, iff $(\forall i, p' \,.\, p \overset{i}{\rightsquigarrow} p' \implies \ell \in \mathsf{O}(p', \overset{O}{=})) \wedge (\forall o, p' \,.\, p \xrightarrow{o} p' \implies \ell \in \mathsf{O}(p', \overset{O}{=}) \wedge o \overset{O}{=}_\ell \bullet)$. ♢

An ℓ observer that is not aware of the value of the switch will then, by $(\overset{\dot{o}}{=})$, not be able to infer any information about the switch state, since all output from the switched process look the same.

Theorem 3 (swi): forall p, $(\overset{I}{=})$, $(\overset{O}{=})$, and $(\overset{Bool}{=})$, if $p \in \mathrm{NI}(\overset{I}{=}, \overset{Bool\times O}{=})$ and $\forall \ell \,.\, \ell \in L$, then $\mathtt{swi}\ b\ p \in \mathrm{NI}(\overset{I}{=}, \overset{\dot{o}}{=})$, where $L = \mathsf{A}(\mathtt{True}, \overset{Bool}{=}) \cup \mathsf{O}(p, \overset{Bool\times O}{=})$, $(\overset{\dot{o}}{=}) = \mathsf{eqmaybe}(\mathsf{A}(\mathtt{True}, \overset{Bool}{=}), \overset{O}{=})$, $(\overset{Bool\times I}{=}) = \mathsf{eqpair}\bullet\mathtt{LR}(\overset{Bool}{=}, \overset{I}{=})$, and $(\overset{Bool\times O}{=}) = \mathsf{eqpair}\bullet\mathtt{R}(\overset{Bool}{=}, \overset{O}{=})$. □

maybe, loop, par. The compositionality results for `maybe`, `loop` and `par` are simple in comparison to the above. For instance, `maybe` preserves the security of a process, even for principals who do not observe `Nothing`, since nothing is ever delivered to the process when such input is received. Using `loop` to create feedback around a secure process does not introduce insecurities, since the process must always meet its public deadlines regardless of what the source of its input is. Looping thus cannot cause an interactive process to block itself. Our theory therefore eliminates known challenges for security under feedback [26,35,43]. Finally, composing secure processes with `par` yields a secure process, since all it does is run the processes in parallel.

Theorem 4 (maybe): forall p, $(\overset{I}{=})$, $(\overset{O}{=})$, if $p \in \mathrm{NI}(\overset{I}{=}, \overset{O}{=})$, then $\texttt{maybe}\ p \in \mathrm{NI}(\overset{\mathbb{1}}{=}, \overset{O}{=})$, where $(\overset{\mathbb{1}}{=}) = \mathsf{eqmaybe}(\emptyset, \overset{I}{=})$. □

Theorem 5 (loop): forall p and $(\overset{I}{=})$, if $p \in \mathrm{NI}(\overset{I}{=}, \overset{I}{=})$, then $\texttt{loop}\ p \in \mathrm{NI}(\overset{I}{=}, \overset{I}{=})$. □

Theorem 6 (par): forall p_1, p_2, $(\overset{I}{=})$, $(\overset{O_1}{=})$, and $(\overset{O_2}{=})$, if $p_1 \in \mathrm{NI}(\overset{I}{=}, \overset{O_1}{=})$ and $p_2 \in \mathrm{NI}(\overset{I}{=}, \overset{O_2}{=})$, then $\texttt{par}\ p_1\ p_2 \in \mathrm{NI}(\overset{I}{=}, \overset{O_1 \times O_2}{=})$, where $(\overset{O_1 \times O_2}{=}) = \mathsf{eqpair}(\overset{O_1}{=}, \overset{O_2}{=})$. □

6 Combinator language

With this core, we build a rich language (Figure 2a) of combinators that mediate the interaction of processes. The language, in addition to facilitating the wiring of process outputs and inputs, includes combinators for transforming and filtering messages, maintaining state, and for switching processes on or off. Complex systems, including schedulers, runtime monitors, and even runtime systems can be implemented in this language. By virtue of compositionality results for our core, the combinators in our language are security preserving. The crucial point is that the compositionality results can be invoked to prove noninterference of processes implemented in our language, obtaining noninterference *by construction*. To demonstrate, we use this language to implement an enforcement of timing-sensitive noninterference in Sect. 7.

Language. The language is summarized in Figure 2a. The figure displays the type of each combinator in the language, along with a brief description of its semantics. For brevity, we leave out descriptions of combinators that are trivial specializations of a more general combinator (e.g. ones with suffix `I` or `O`: specializations that operate only on input and output). The implementation of each combinator in terms of core combinators is given in the TR [33].

Message transformation & process state. `mapI`, `mapO`, `staI`, and `staO` are trivial specializations of the core `map` and `sta` combinators. For instance, `mapI` is defined as $\texttt{mapI}\ f\ p = \texttt{map}\ f\ \texttt{id}\ p$. Thus, `mapI` only transforms inputs. We make heavy use of `mapI` and `mapO` for routing and restructuring messages in Sect. 7.

Message filtering. `filter` drops messages that do not satisfy a predicate. We implement `filter` using `map`, by transforming predicates into functions that map

messages that do not satisfy the predicate to `Nothing`. We then use `maybe` to discard resulting `Nothing` input. We cannot do the same for output; the process still performed work. The `source` combinator drops all input.

Message tagging. The tagging combinators tag and untag messages. These are simple specializations of `map`; for instance, `tagI` v $p \overset{i}{\rightsquigarrow}$ iff $p \overset{(v,i)}{\rightsquigarrow}$. The only non-trivial tagging combinator, `tokenI` $v\,p$, treats a tag as a token, only passing an input to p if the input is tagged with v (consuming the token). A sample use of the tag combinators is implementing point-to-point communication; this can be done by having senders tag a message with the ID of a recipient process, and having said process use `tokenI` to only process messages addressed to it.

Process switching. Two specializations of `swi` are noteworthy. `swiI` combinator, by only switching its subprocess on or off upon receiving input, implements a preemptive switching strategy. Likewise, `swiC`, by using input to switch its subprocess on, and output to switch it off, implements a cooperative switching strategy. We use `swiI` and `swiC` in to implement scheduling strategies in Sect. 7.

Process composition. With `par` and `loop`, we can compose any number of processes that all receive copies of each other's output. This "universal" composition can be specialized to more restricted forms of communication, including "sequential" composition, using our other combinators to selectively route messages.

Compositionality of language. The compositionality results for our language are listed in Figure 2b. Each black-bordered box contains a compositionality result; the first line is its guarantee, while subsequent lines in the box are assumptions under which that guarantee holds. Occurrences of unbound variables in a box are implicitly universally quantified. For instance, the first six lines under "process state" is one compositionality result, namely Theorem 2 restated.

The meaning of each assumption has already been explained in Sect. 5, save for three. First, $\mathsf{O}(\overset{V}{=}) = \{\ell \mid \forall v \,.\, v \overset{V}{=}_\ell \bullet\}$. That part of the assumption of `tokenI` states that ℓ must either be aware of the token (thus observing presence of all input), or oblivious to all input (thus public output is independent of all input). Second, $(\overset{Bool}{=})_\ell = \{\langle \texttt{True}, \texttt{True}\rangle, \langle \texttt{False}, \texttt{False}\rangle, \langle \bullet, \bullet\rangle\}, \forall \ell$. Third, $f|I$ is the restriction of f to I. We use these two definitions to state that for *observable* observably equivalent values, the filter functions make the same filtering decision.

Compositionality follow from Theorems 1, 2, 3, 4, 5, and 6.

Corollary 1 (composition): Each statement in Figure 2b is true. □

Fig. 2: Language

7 Case study: SME

To demonstrate the practicality of our results, we implement secure multi-execution (SME) [10], the enforcement that we discussed in Sect. 2.

We develop two variations of SME, which differ in how the execution of process copies is managed. The former variant uses a preemptive scheduling strategy to schedule the process copies. For this variant, we show how a proof of soundness can be straightforwardly obtained by invoking our compositionality results. The latter variant uses a cooperative scheduling strategy. Here we demonstrate a timing leak, and, using our compositionality results, trace the insecurity in the implementation to a single component. Together, this demonstrates that our theory can be used to straightforwardly establish timing-sensitive noninterference of a complex system, and to identify subcomponents that cause insecurities.

We stress that our construction easily generalizes to lattices of any shape and size, like SME does [34], even though for clarity of presentation, we assume the two-point lattice $\mathcal{L}_{\mathsf{HL}}$. We will use the definition of message-passing processes and their observables, i.e. $\mathbb{M}$, obs, $(\overset{\mathbb{M}}{=})$ and $(\overset{\dot{\mathbb{M}}}{=})$, from Examples 2 and 5.

Secure execution. At first, it appears our compositionality results will not aid us in establishing soundness for an implementation of SME in our language; our results assume that processes being composed are secure, while SME makes no such assumption. We observe that only a tiny part of SME is responsible for enforcing security. We deconstruct SME, separating plumbing and scheduling from this part, prove that the part enforces NI, and then leverage our compositionality results to show that plumbing and scheduling does not introduce insecurities.

This tiny part is SE: a combinator for executing any given $p \in \mathit{IProc}\ \mathbb{M}\ [\dot{\mathbb{M}}]$ securely. SE secures the ℓ-copies. With $(\mathtt{SE}\ \ell\ p)$ denoting the securely executed ℓ-copy of p, SE achieves this effect by *1)* feeding only the ℓ-observable part of input to the ℓ-copy, and *2)* dropping all non-ℓ parts of output from p. Intuitively, $(\mathtt{SE}\ \ell\ p)$ is a secure process since $(\mathtt{SE}\ \ell\ p)$ outputs messages only on channels labeled ℓ, and computes these using only input on channels labeled $(\sqsubseteq \ell)$. Both *1)* and *2)* are needed; without *1)*, input from $(\not\sqsubseteq \ell)$ can flow to output channels labeled ℓ, and without *2)*, $(\mathtt{SE}\ \ell\ p)$ can leak between incomparable channels in $(\sqsubseteq \ell)$.

```
SE : ℒ -> IProc 𝕄 𝕄̇ -> IProc 𝕄 𝕄̇
SE ℓ p = map obsℓ prjℓ (maybe p)
```

Listing 1.1: Secure Execution

To achieve *1)*, we use $(\mathtt{mapI}\ \mathsf{obs}_\ell\ p).\ \mathsf{obs}_\ell$ preprocesses input to p in the manner required by *1)*. To achieve *2)*, we use $(\mathtt{mapO}\ \mathsf{prj}_\ell\ p)$, where $\mathsf{prj} : \mathcal{L} \to \dot{\mathbb{M}} \to \dot{\mathbb{M}}$ is a function that projects output on non-ℓ channels to Nothing.

$$\mathsf{prj}_\ell\ \mathtt{Nothing} = \mathtt{Nothing} \qquad \mathsf{prj}_\ell\ cn = (\mathsf{lev}(c) = \ell)\ \mathtt{?}\ \mathtt{Just}\ cn\ \mathtt{:}\ \mathtt{Nothing}$$

With this, we define SE as in Listing 1.1.

Theorem 7: $\forall \ell, p \bullet (\mathtt{SE}\ \ell\ p) \in \mathrm{NI}(\overset{\mathbb{M}}{=}, \overset{\dot{\mathbb{M}}}{=})$.

Proof sketch. Pick ℓ and p. We need to prove $(\mathtt{SE}\ \ell\ p) \in \mathrm{NI}(\overset{\mathbb{M}}{=}, \overset{\dot{\mathbb{M}}}{=})$. Pick ℓ', and s such that $(\mathtt{SE}\ \ell\ p) \overset{s}{\dashrightarrow}$. Case on $\ell \sqsubseteq \ell'$. We use the following simulations in the cases.

$$\mathcal{R}_1 = \{\langle \hat{s}, \mathtt{SE}\ \ell\ \hat{p}\rangle \mid \mathtt{SE}\ \ell\ \hat{p} \overset{\hat{s}}{\dashrightarrow}\} \quad \mathcal{R}_0 = \{\langle \hat{s}, \mathtt{SE}\ \ell\ \hat{p}\rangle \mid \hat{s} \in (\mathit{Eff}\ \mathbb{M}\ \{cn \mid \mathsf{lev}(c) = \ell\}^\bullet)^\omega\}$$

In the "true" case, SE replaces ℓ'-unobservable input with Nothing (by definition of obs_ℓ), which in turn gets dropped by maybe. Since ℓ'-observable inputs are only ℓ'-$(\overset{\mathbb{M}}{=})$-observably equivalent with themselves, this together gives that changes in ℓ'-unobservable input to $(\mathtt{SE}\ \ell\ p)$ never propagate into p. Thus, we can show that $\mathcal{R}_1$, which relates streams to processes very tightly, is a ℓ'-$(\overset{\mathbb{M}}{=})$-$(\overset{\dot{\mathbb{M}}}{=})$-stream-simulation. It is also easy to see that $\langle s, \mathtt{SE}\ \ell\ p\rangle \in \mathcal{R}_1$. In the "false" case, we use a different observation: SE maps all output from p to Nothing if it is not a message on a ℓ-labeled channel (by definition of prj_ℓ). Since messages on ℓ-labeled channels are ℓ'-$(\overset{\dot{\mathbb{M}}}{=})$-equivalent to Nothing, *none* of the outputs from $\mathtt{SE}\ \ell\ p$ are ℓ'-observable. This lets us use $\mathcal{R}_0$. To establish $\langle s, \mathtt{SE}\ \ell\ p\rangle \in \mathcal{R}$, we use the following lemma. ■

Lemma 1: $\forall p, \ell \centerdot (\mathtt{SE}\ \ell\ p) \in \mathit{IProc}\ \mathbb{M}\ \{cn \mid \mathsf{lev}(c) = \ell\}$. □

Scheduler processes. Our two variations of SME execute ℓ-copies concurrently, with executions coordinated by a scheduler process. A scheduler chooses which process copy goes next by outputting its security level. Like previous work on SME [10,20,34], our schedulers receive no input. This simplifies reasoning (this way, schedulers cannot leak information [20]). Our schedulers are rich enough to express practical scheduling strategies, including Round-Robin scheduling.

The set of schedulers is $\mathit{IProc}\ \emptyset\ \mathcal{L}$. Since schedulers receive no input, we make scheduler choices public. We define $\ell' \overset{\mathcal{L}}{=}_\ell \ell''$ iff $\ell' = \ell''$. Let $(\overset{\mathcal{L}}{=}) = \mathsf{eqmaybe}(\mathcal{L}, \overset{\mathcal{L}}{=})$, and let $(\overset{\emptyset}{=}_\ell) = \{\langle \bullet, \bullet\rangle\}$, $\forall \ell$. Since schedulers receive no input, the following is clear.

Corollary 2: $\forall p \in \mathit{IProc}\ \emptyset\ (\mathit{Maybe}\ \mathcal{L}) \centerdot p \in \mathrm{NI}(\overset{\emptyset}{=}, \overset{\mathcal{L}}{=})$. □

Secure multi-execution, preemptive. Our first variation of SME schedules ℓ-copies *preemptively*. Example SME schedulers of this sort are Multiplex-2 [20], and the deterministic fair schedulers [34]. In this variation, the ℓ-copies run in parallel with a scheduler. In each time unit, the scheduler can switch one of the process copies on or off (preempting it).

```
1 SMEP : IProc ∅ L̇ -> IProc M Ṁ -> IProc M Ṁ
2 SMEP pS p =
3   mapI Right (in (map0 merge runs) (source pS))
4  where
5   runs = par (SwCP H (SE H p)) (SwCP L (SE L p))
```

Listing 1.2: SME, Preemptive

The SMEP combinator in Listing 1.2 achieves this effect. Here, $\mathtt{SME_P}\ p_\mathtt{S}\ p$ securely executes a H- and a L-copy of a process p in parallel (line 5). These ℓ-copies are made switchable by $\mathtt{SwC_P}$ (defined later). The scheduler $p_\mathtt{S}$ interacts with these switches by means of the in construct. Whereas in ensures $p_\mathtt{S}$ interacts only with the ℓ-copies, source makes this interaction unidirectional.

```
SwCP : L -> IProc M Ṁ -> IProc (Either L̇ M) Ṁ
SwCP ℓ p =
  mapI tobmℓ (swiI False (maybe p))
 where
  tobm : L -> Either L̇ M -> Bool * Ṁ
  tobm ℓ (Right m)       = ⟨False,Just m⟩
  tobm ℓ (Left Nothing)  = ⟨False,Nothing⟩
  tobm ℓ (Left (Just ℓ')) = ⟨ℓ==ℓ',Nothing⟩
```

Listing 1.3: Switch Copy, Preemptive

Before explaining the `maps` on line 3, let's delve into $\texttt{SwC}_\texttt{P}$, in Listing 1.3. Besides making p preemptively switchable (by `swiI`), $(\texttt{SwC}_\texttt{P}\ \ell\ p)$ defines the interface between an ℓ-copy and the scheduler. As the type of $\texttt{SwC}_\texttt{P}$ indicates, $(\texttt{SwC}_\texttt{P}\ \ell\ p)$ receives switch commands from the scheduler, and messages from the environment. Function `tobm` specifies how $(\texttt{SwC}_\texttt{P}\ \ell\ p)$ reacts to input. The function outputs a pair $\langle b, \dot{m}\rangle$; b determines whether the switch should be flipped, and $\dot{m}$ is the input message (if any) to p. Here, $b = \texttt{True}$ iff the input is ℓ from the scheduler, and $\dot{m} = \texttt{Just}\ m$ iff the input is m from the environment.

Note that `tobm` only changes how data is packaged w/o changing the data itself (except ℓ==ℓ', which is public). Thus, `tobm` is noninterfering. This, together with our compositionality results, gives us that $\texttt{SwC}_\texttt{P}$ is security-preserving.

Corollary 3: $\forall p, \ell \centerdot p \in \text{NI}(\overset{\mathbb{M}}{=}, \overset{\mathbb{M}^\bullet}{=}) \implies (\texttt{SwC}_\texttt{P}\ \ell\ p) \in \text{NI}(\overset{I}{=}, \overset{\mathbb{M}^\bullet}{=})$,
where $I = \mathit{Either}\ \mathcal{L}^\bullet\ \mathbb{M}$ and $(\overset{I}{=}) = \mathsf{eqeither}(\overset{\mathcal{L}^\bullet}{=}, \overset{\mathbb{M}}{=})$. □

In Listing 1.2 line 3, `Right` maps environment input into $\mathit{Either}\ \mathcal{L}^\bullet\ \mathbb{M}$ (the space of values switched ℓ-copies receive). Finally, `merge` projects each pair of output messages (if any) from the ℓ-copies to a single message. It does so by preferring the right component, choosing the left component only if the right component is `Nothing`. We define $\mathsf{merge} : O^\bullet \times O^\bullet \to O^\bullet$ as follows.

$$\mathsf{merge}\ \langle \dot{o}, \texttt{Nothing}\rangle = \dot{o} \qquad \mathsf{merge}\ \langle \_, \texttt{Just}\ o\rangle = \texttt{Just}\ o$$

Lemma 2: $\forall \ell_\texttt{H}, \ell_\texttt{L} \centerdot \ell_\texttt{H} \not\sqsubseteq \ell_\texttt{L} \implies (\mathsf{merge}|I) \in \text{NI}(\overset{\mathbb{M}^{\bullet 2}}{=}, \overset{\mathbb{M}^\bullet}{=})$, where $(\overset{\mathbb{M}^{\bullet 2}}{=}) = \mathsf{eqpair}(\overset{\mathbb{M}^\bullet}{=}, \overset{\mathbb{M}^\bullet}{=})$ and $I = \{cn \mid \mathsf{lev}(c) = \ell_\texttt{H}\}^\bullet \times \{cn \mid \mathsf{lev}(c) = \ell_\texttt{L}\}$. □

By Lemma 1, the output space of $(\texttt{SwC}_\texttt{P}\ \ell\ (\texttt{SE}\ \ell\ p))$ is $\{cn \mid \mathsf{lev}(c) = \ell\}^\bullet$

Corollary 4: $\forall p, \ell \centerdot (\texttt{SwC}_\texttt{P}\ \ell\ (\texttt{SE}\ \ell\ p)) \in \mathit{IProc}\ I\ O$,
where $O = \{cn \mid \mathsf{lev}(c) = \ell\}$ and $I = \mathit{Either}\ \mathcal{L}^\bullet\ \mathbb{M}$. □

Now, $\{cn \mid \mathsf{lev}(c) = \texttt{H}\}^\bullet \times \{cn \mid \mathsf{lev}(c) = \texttt{L}\}^\bullet$ is the output space of `runs`. This lets us invoke Lemma 2 on the `mapO merge` part of $\texttt{SME}_\texttt{P}$. By invoking the compositionality results for `source`, `in` and `mapI`, we get a proof of soundness of $\texttt{SME}_\texttt{P}$.

Corollary 5: $\forall p_\texttt{S}, p \centerdot (\texttt{SME}_\texttt{P}\ \ell\ p) \in \text{NI}(\overset{\mathbb{M}}{=}, \overset{\mathbb{M}^\bullet}{=})$. □

This venture highlights the power of our approach: it enables SME to simply be implemented, reducing soundness to proving properties of simple components.

Secure multi-execution, cooperative. Our second variation of SME schedules ℓ-copies *cooperatively*. An example scheduler of this sort is $\texttt{select}_{\text{lowprio}}$ [10], implemented in FlowFox [9] on a per-event basis. Here, processes are arranged like in $\texttt{SME}_\texttt{P}$. The key difference is that at only one process (including the scheduler) can be active at a time. An active process remains active until it releases control. When an ℓ-copy does, the scheduler receives control, remaining active until it determines which process copy to activate, and activates it.

However, as we will confirm, this approach has a timing leak: allowing the H-copy to control when it releases control to the scheduler means that the time at which the L-copy is subsequently activated can depend on H information [20,34].

```
SMEC : IProc ∅ 𝓛̇ -> IProc 𝕄 (Bool*𝕄̇) -> IProc 𝕄 𝕄̇
SMEC pS p =
  map Right snd (in (map0 merge runs) (SwSC pS))
 where
  runs = par (SwCC H (SEC H p)) (SwCC L (SEC L p))
```

Listing 1.4: SME, Cooperative

```
SwCC : 𝓛 -> IProc 𝕄 (Bool*𝕄̇)
         -> IProc (Either 𝓛̇ 𝕄) (Bool*𝕄̇)
SwCC ℓ p =
  map tobmℓ tobm' (swiC False (map0 tobbm (maybe p)))
 where  // tobbm and tobm' omitted; see the TR.
```

Listing 1.5: Switch Copy, Cooperative

The $\texttt{SME}_\texttt{C}$ combinator, in Listing 1.4, implements this approach. The structure is exactly like $\texttt{SME}_\texttt{P}$. However, a few combinators have been modified. First, the type of $\texttt{SME}_\texttt{C}$ is different; processes to be multi-executed are now $\mathit{IProc}\ \mathbb{M}\ (\mathit{Bool} \times \mathbb{M}^\bullet)$. The Boolean output signifies control release. Second, SE needs to be modified slightly as a result. The new combinator, $\texttt{SE}_\texttt{C}$, enforces NI; see the TR [33] for details. Third, the process switch needs to be updated to match this new scheduling semantics. The new switch, $\texttt{SwC}_\texttt{C}$, is given in Listing 1.5. Compared to $\texttt{SwC}_\texttt{P}$, $\texttt{SwC}_\texttt{C}$ replaces swiI with swiC, and propagates a release signal from the process to both swiC and the scheduler. The following should thus be of no surprise.

Corollary 6: $\forall p, \ell \bullet p \in \mathrm{NI}(\overset{\mathbb{M}}{=}, \overset{O}{=}) \implies (\texttt{SwC}_\texttt{C}\ \ell\ p) \in \mathrm{NI}(\overset{I}{=}, \overset{O}{=})$, where $I = \mathit{Either}\ \mathcal{L}^\bullet\ \mathbb{M}$, $O = \mathit{Bool} \times \mathbb{M}^\bullet$, $(\overset{I}{=}) = \mathsf{eqeither}(\overset{\mathcal{L}^\bullet}{=}, \overset{\mathbb{M}}{=})$, and $(\overset{O}{=}) = \mathsf{eqpair}(\mathsf{eqat}(\mathit{Bool}, \ell), \overset{\mathbb{M}^\bullet}{=})$. □

Here, eqat defines that values are observable only to principals at or above a given level; for all ℓ, $\mathsf{eqat}(A, \ell)$ is the least equivalence relation $(\overset{A}{=}_\ell)$ satisfying

$$a \overset{A}{=}_{\ell'} a' \text{ iff } \ell \not\sqsubseteq \ell' \vee a = a' \qquad\qquad a \overset{A}{=}_{\ell'} \bullet \text{ iff } \ell \not\sqsubseteq \ell'.$$

```
SwSC : IProc ∅ 𝓛̇ -> IProc (Bool*𝕄̇) (Either 𝓛̇ 𝕄)
SwSC b p =
  map tobu toelm (swiC True (map0 tobelm (source p)))
 where // tobu, toelm and tobelm in the TR.
```

Listing 1.6: Switch Scheduler, Cooperative

Things start to go wrong in the scheduler switch, $\texttt{SwS}_\texttt{C}$, sketched in Listing 1.6. This switch follows the structure of $\texttt{SwS}_\texttt{P}$. When switched on (by a ℓ-copy), $\texttt{SwS}_\texttt{C}$ remains active until it produces a security level ℓ (which, in turn, by $\texttt{SME}_\texttt{C}$, switches the ℓ-copy on).

Now a problem emerges in swiC. Since the Boolean used to switch the scheduler comes from the H- and the L-copy, L needs to be oblivious to the scheduler process. However, the scheduler process outputs security levels to the L-copy, which are L. If we instead make security levels H, Corollary 6 becomes false; the switch signal sent to the L-copy becomes H, forcing the switch on the L-copy to be H, and since L is not oblivious to the L-copy, a leak can occur. There thus appears to be an irreparable conflict in this variation of SME; L output must be independent of H input, but the time at which the L-copy regains control depends on output from the H-copy, which depends on H input.

8 Related Work

We discuss work in areas most related to ours: information-flow control of timing channels, timed interaction, and theories of information-flow secure composition.

Timing channels. Timing channels can be categorized as internal and external [39]. Several program analyses and transformations have been proposed to stop leaks through external channels. Proposed white-box approaches include the following. Hedin and Sands developed a type system that rejects programs for which the time it takes to reach the point of the `L` effect can depend on `H` [14]. Zhang et al. annotate statements in an imperative language with a read and write label expressing how information can flow through the runtime [47]. Agat gave a program transformation that, in a program that passes Denning-style enforcement [41] (which rules out explicit and implicit flows), pads `H ifs` and bans `H whiles` [1]. Askarov et al. present a black-box timing leak mitigator [5]. Here, outputs are queued, and released FIFO according to a pre-programmed schedule. If no output is in the FIFO when a release is scheduled, the schedule is updated (i.e. slowed). This places a logarithmic bound on timing leaks. Devriese and Piessens formalize secure multi-execution (SME) that executes a program multiple times, once for each security level, while carefully dispatching inputs and ensuring that an execution at a given level is responsible for producing outputs for sinks at that level [10].

Whereas the above approaches performs little or no exploration of compositionality, we demonstrate that our timing-sensitive noninterference is preserved under composition. Our combinators can be used to prove timing-sensitive noninterference in large systems, by construction. By implementing SME, we have shown that it is compatible with our theory. The mitigations are not compatible with our theory as-is, since these allow leaks through timing, whereas our theory allows no leaks. Modifying our theory to accommodate these is a promising line of future work. The compatibility of the other approaches to our theory is unclear as they make environment assumptions that may be incompatible with ours. Compared to [14], our discrete-timed model is simplistic. We note, however, that no part of our theory places restrictions on how fine the discretization of time can be. Our work focuses on eliminating external timing channels, because they have been demonstrated to be exploitable [6,11,22,30], and because internal timing channels are caused by external timing channels of subcomponents.

Timed interaction. Timed models of interaction have been studied extensively in a process algebraic setting [8,13,15,29,40]. The prevalent approach has been to introduce a special timed tick action to the model, leaving synchronization constructs untimed [8,13,15,40]. This tick action requires special attention in the theory; for instance, it is useful to require that processes are weakly time alive, i.e. never prevent time from passing by engaging in infinite interaction. Instead, our model times output, alleviating the need to introduce a special action and machinery around it. This yields a cleaner theory; for instance, progress is already built into our definition of interactive process. While this limits how much work a process can do in a time unit, the discretization of time can be arbitrarily fine. Whereas these calculi mostly use bisimulation to compare

processes, our simulation relation is more forgiving when it comes to reasoning about nondeterministic choice. Since our theory operates on transition systems as opposed to on a language of processes, our theory is more general.

Focardi and Gorrieri's work on information-flow secure interaction is particularly related to ours [13]. Their security properties are bisimulation-based, with the `H` part of environment modeled explicitly as a process that binds all `H` channels and only interacts on `H` channels. In contrast, our environments are implicit, and can e.g. be any interactive process.

Timed I/O automata are real-time systems that synchronize through discrete, timeless actions [21]. Like our interactive processes, these systems are input total, and it is assumed that time can pass. However, systems are finite-state, and, like the process algebras, passage of time is separate from synchronization.

In summary, while our model of time is weaker than those in some other timed computation models (notably, dense time), time can be discretized as needed, and conflating output with passage of time greatly simplifies our theory.

Theories for secure composition (information-flow). With his seminal paper [26], McCullough sparked a study into information-flow secure composition of nondeterministic systems in the 80s that continues to this day [19,24,28,35,42,44]. This work studies the relative merits of several trace-based formalizations of possibilistic *progress-sensitive noninterference* [3,4], in terms of whether they are preserved under e.g. universal composition, sequential composition (a.k.a. cascade), and feedback. Whereas some properties are preserved under all of these [19], others fail for some combinators, most notably feedback [43]. These models are all untimed. It would be worthwhile to apply our timing model in these settings and explore how these security properties classify programs. Requiring that the presence of all output is `L` is a good starting point, since this makes these properties timing-sensitive in our timing model. However, more work may be needed, since the system models differ subtly (e.g. they are not all input total). Our simulation relation is inspired by Rafnsson and Sabelfeld [35]. While their relation was designed to facilitate an inductive proof principle, ours is designed around a coinductive proof principle. Our simulation is simpler as a result.

Secure composition has also been studied in great detail in a process algebraic setting [12,16,17,31,36,37], Parallel composition is one of the defining features of process algebra, making compositional reasoning a key concern. In contrast to this work (which studies compositionality of parallel composition), our work studies compositionality of a language of combinators. Further, our model is timed, while these are not. Finally, the behavioral equivalence of choice is bisimulation, which we find to be too strict for possibilistic noninterference.

More recently, Mantel et al. explore secure composition in a shared-memory concurrent setting [2,25]. They develop a security condition that is sensitive to the assumptions that each thread makes on whether other threads can read or write to shared variables. For instance, the right-component of the Clark-Hunt example in Sect. 2 assumes that no other thread reads `x`, and, thus, the two components cannot be securely composed since the left-component violates this assumption. Their approach is more fine-grained than ours, since compositional-

ity is parameterized by individual environment assumptions of subcomponents. However, their system model is untimed, threads are arranged in a fixed, flat structure, communication is only via shared memory, and only parallel composition is considered. In contrast, our system model is timed, and our combinators enable modeling fairly arbitrary structures of interacting processes (including shared memory), as demonstrated in Sect. 7. Exploring whether this finer granularity can be introduced into our theory is a promising direction of future work.

All of these approaches consider only combinators that passively glue together two processes, facilitating interaction. In contrast, our combinators actually *do* something, e.g. maintain state, switch processes on or off, and transform messages. As a result, our theory presents a rich toolset for reasoning about secure composition, made even richer by its generic nature (arbitrary message types, combinators parameterized by functions, etc.).

Acknowledgment. This research was supported in part by US Navy grant N000141310156 and NSF grant 1320470.

References

1. Agat, J.: Transforming out timing leaks. In: POPL (2000)
2. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: CSF (2015)
3. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88313-5_22
4. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: CSF (2009)
5. Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: CCS (2010)
6. Brumley, D., Boneh, D.: Remote timing attacks are practical. Comput. Netw. **48**(5), 701–716 (2005)
7. Clark, D., Hunt, S.: Noninterference for deterministic interactive programs. In: FAST (2008)
8. Corradini, F., D'Ortenzio, D., Inverardi, P.: On the relationships among four timed process algebras. Fundamenta Informaticae **38**(4), 377–395 (1999)
9. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a web browser with flexible and precise information flow control. In: CCS (2012)
10. Devriese, D., Piessens, F.: Non-interference through secure multi-execution. In: S&P (2010)
11. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: CCS (2000)
12. Focardi, R., Gorrieri, R.: A classification of security properties for process algebras. JCS **3**(1), 5–33 (1995)
13. Focardi, R., Gorrieri, R., Martinelli, F.: Real-time information flow analysis. JSAC **21**(1), 20–35 (2003)
14. Hedin, D., Sands, D.: Timing aware information flow security for a JavaCard-like bytecode. ENTCS **141**(1), 163–182 (2005)
15. Hennessy, M., Regan, T.: A temporal process algebra. In: FORTE (1990)

16. Honda, K., Vasconcelos, V., Yoshida, N.: Secure information flow as typed process behaviour. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 180–199. Springer, Heidelberg (2000). doi:10.1007/3-540-46425-5_12
17. Honda, K., Yoshida, N.: A uniform type structure for secure information flow. In: POPL (2002)
18. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. EATCS Bull. **62**, 62–222 (1997)
19. Johnson, D.M., Thayer, F.J.: Security and the composition of machines. In: CSFW (1988)
20. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure flow, exploring a new approach. In: S&P (2011)
21. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: Timed, I/O automata: a mathematical framework for modeling and analyzing real-time systems. In: RTSS (2003)
22. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). doi:10.1007/3-540-68697-5_9
23. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**, 219–246 (1989)
24. Mantel, H.: On the composition of secure systems. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 81–94, May 2002
25. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: CSF (2011)
26. McCullough, D.: Specifications for multi-level security and a hook-up property. In: S&P, pp. 161–166 (1987)
27. McCullough, D.: Noninterference and the composability of security properties. In: S&P, pp. 177–186 (1988)
28. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: S&P (1994)
29. Nicollin, X., Sifakis, J.: An overview and synthesis on timed process algebras. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 376–398. Springer, Heidelberg (1992). doi:10.1007/3-540-55179-4_36
30. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). doi:10.1007/11605805_1
31. Pottier, F.: A simple view of type-secure information flow in the pi-Calculus. In: CSFW, June 2002
32. Rafnsson, W., Hedin, D., Sabelfeld, A.: Securing interactive programs. In: CSF (2012)
33. Rafnsson, W., Jia, L., Bauer, L.: Timing-sensitive noninterference through composition (Technical report). Technical report CMU-CyLab-16-005, CMU CyLab (2016)
34. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: fine-grained, declassification-aware, and transparent. In: CSF (2013)
35. Rafnsson, W., Sabelfeld, A.: Compositional information-flow security for interactive systems. In: CSF, pp. 277–292 (2014)
36. Ryan, P.Y.A.: Mathematical models of computer security. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 1–62. Springer, Heidelberg (2001). doi:10.1007/3-540-45608-2_1
37. Ryan, P., Schneider, S.: Process algebra and non-interference. In: CSFW (1999)

38. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. JSAC **21**(1), 5–19 (2003)
39. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: CSFW (2000)
40. Ulidowski, I., Yuen, S.: Extending process languages with time. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 524–538. Springer, Heidelberg (1997). doi:10.1007/BFb0000494
41. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. JCS **4**(3), 167–187 (1996)
42. Wittbold, J.T., Johnson, D.M.: Information flow in nondeterministic systems. In: S&P (1990)
43. Zakinthinos, A., Lee, E.S.: How and why feedback composition fails. In: CSFW (1996)
44. Zakinthinos, A., Lee, E.S.: A general theory of security properties. In: S&P (1997)
45. Zanarini, D., Jaskelioff, M., Russo, A.: Precise enforcement of confidentiality for reactive systems. In: CSF (2013)
46. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: CSFW (2003)
47. Zhang, D., Askarov, A., Myers, A.C.: Language-based control and mitigation of timing channels. In: PLDI (2012)