Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency

Farzaneh Derakhshan
Illinois Institute of Technology, Chicago, IL, USA

Stephanie Balzer
Carnegie Mellon University, Pittsburgh, PA, USA

Yue Yao
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Noninterference guarantees that an attacker cannot infer secrets by interacting with a program. Information flow control (IFC) type systems assert noninterference by tracking the level of information learned (`pc`) and disallowing communication to entities of lesser or unrelated level than the `pc`. Control flow constructs such as loops are at odds with this pattern because they necessitate downgrading the `pc` upon recursion to be practical. In a concurrent setting, however, downgrading is not generally safe. This paper utilizes *session types* to track the flow of information and contributes an IFC type system for message-passing concurrent processes that allows downgrading the pc upon recursion. To make downgrading safe, the paper introduces *regrading policies*. Regrading policies are expressed in terms of integrity labels, which are also key to safe composition of entities with different regrading policies. The paper develops the type system and proves *progress-sensitive noninterference* for well-typed processes, ruling out timing attacks that exploit the relative order of messages. The type system has been implemented in a type checker, which supports security-polymorphic processes.

2012 ACM Subject Classification Theory of computation → Linear logic; Security and privacy → Logic and verification; Theory of computation → Process calculi

Keywords and phrases Regrading policies, session types, progress-sensitive noninterference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.11

Related Version *Technical Report*: `https://doi.org/10.48550/arXiv.2407.20410`

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.4`

Funding *Stephanie Balzer*: Supported in part by the Air Force Office of Scientific Research under award number FA9550-21-1-0385 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed here are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Defense.

1 Introduction

With the emergence of new applications, such as Internet of Things and cloud computing, today's software landscape has become increasingly *concurrent*. A dominant computation model adopted by such applications is *message passing*, where several concurrently running processes connected by channels exchange messages. A further common aspect is the need for security, ensuring that confidential information is not leaked to a (malevolent) observer.

Information flow control (IFC) type systems [36, 39, 42] rule out information leakage by type checking. These systems statically track the level of information learned by an entity and disallow propagation to parties of lesser or unrelated levels, given a security lattice. The ultimate property to be asserted by an IFC type system is noninterference, a program equivalence statement up to the confidentiality level of an observer. The gold standard is *progress-sensitive noninterference* (PSNI) [24], which treats divergence as an

© Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao;
licensed under Creative Commons License CC-BY 4.0

observable outcome. PSNI thus only equates a divergent program with another diverging one, whereas *progress-insensitive noninterference* (PINI) regards divergence to be equal to any outcome. Especially in a concurrent setting, PSNI is a sine qua non because the *termination channel* [36] can be scaled to many parallel computations, each leaking "just" one bit [4, 40].

Guaranteeing PSNI, or even PINI for that matter, can become both a blessing and a curse in a concurrent setting. To ensure such a strong property, IFC type systems have to be very restrictive. The troublemakers, in particular, are control flow constructs, such as loops and if statements. Whereas IFC type systems for sequential languages allow the pc label[1] to be lowered to its previous level for the continuation of a control flow construct, even if the construct itself runs at high, this treatment is no longer safe in a concurrent setting [39]. To uphold noninterference, IFC type systems for concurrent languages typically forbid high-security loop guards and may even put restrictions on if statements, depending on thread scheduling and attacker model [35, 37, 39].

The use of linearity provides some relief [7, 20, 46–48], allowing high-security loop guards. Linearity also facilitates race freedom, key to guaranteeing observational determinism and, thus, the absence of certain timing attacks [7, 20, 48]. A family of concurrent languages that employ linearity are *session types* [9, 26, 27, 30, 43]. Session types are used for message-passing concurrency, typically in the context of process calculi, where concurrently running processes communicate along channels. A distinguishing characteristic of session types is their ability to assert *protocol adherence*. A session-typed channel prescribes not only the types of values that can be transported over the channel but also their relative sequencing.

In this paper, we develop a flow-sensitive IFC session type system that not only supports recursive processes with arbitrary recursion guards, including high-security ones, but also identifies synchronization patterns that make it safe for the process body to downgrade to the initial pc level upon recursion. We refer to this adjustment of confidentiality level as *regrading*. To enforce the safety of regrading, we complement confidentiality with *integrity* [8]. Integrity allows prescribing a process a *regrading policy*, ensuring that any confidential information learned during the high-security parts of the loop cannot be rolled forward to the next iteration. Processes are *polymorphic* in the confidentiality and integrity labels, ensuring maximal flexibility of the IFC type system.

We contribute a type checker for our IFC type system, yielding the language SINTEGRITY. The type checker supports security-polymorphic processes using local *security theories*. Well-typed processes in SINTEGRITY enjoy PSNI. To prove this result, we develop a *logical relation* for integrity, showing that well-typed processes are self-related (fundamental theorem, Thm. 1). We then prove that the logical relation is closed under parallel composition and that related processes are bisimilar (adequacy theorem, Thm. 3).

Regrading is related to robust declassification [6, 15, 33, 44, 45, 49], as both allow downgrading the pc using integrity. In contrast to declassification, which deliberately releases information and thus intentionally weakens noninterference, regrading preserves noninterference. The distinction also manifests in how integrity is used. Whereas integrity is used in robust declassification to convey how trustworthy the information is on which a regrading decision is based, integrity in our work is used to impose extra synchronization policies on regrading processes to prevent leakage by downgrading the pc upon recursion. As such, regrading constitutes a more permissive IFC mechanism.

[1] The pc (program counter) label approximates the level of confidential information learned up to the current execution point.

Contributions

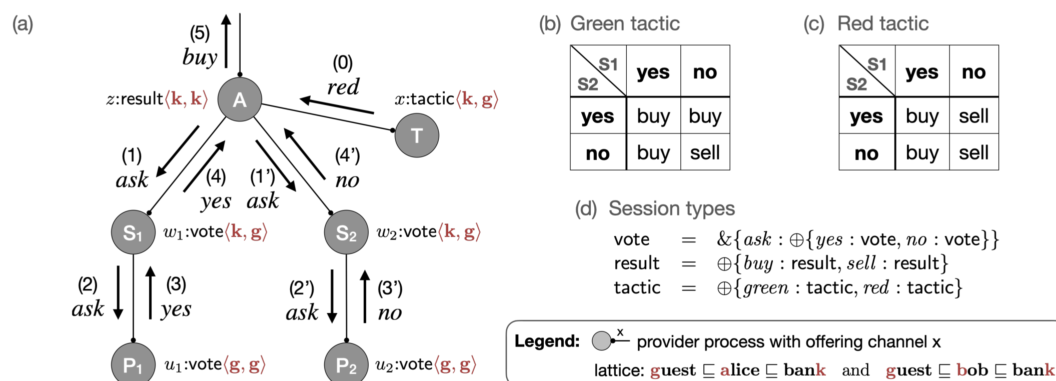
- The notion of a regrading policy to downgrade a process' confidentiality, retaining PSNI.
- The language SINTEGRITY, a flow-sensitive IFC session type system for asynchronous message-passing with confidentiality and integrity to support regrading policies.
- A logical relation for integrity to prove that SINTEGRITY processes satisfy PSNI.
- A type checker for SINTEGRITY, available as an artifact.

The complete formalization with proofs is available as a technical report (TR) [21].

2 Motivating example and background

This section provides an introduction to session-typed programming and IFC control based on a running example. Our language SINTEGRITY is an *intuitionistic linear session type* language [9, 41]; thus, our presentation is specific to that family of session types.

We use a simple bank survey as an example. The survey is carried out by an analyzer at a bank to decide whether to buy or sell a share of stock. The analyzer's decision depends on the opinion of two groups of participants, queried by two surveyors, and a strategy provided by a tactician. For simplicity, we assume that each group of participants only consists of one participant, and the surveyors simply pass the opinion of their participant to the analyzer.



S2 \ S1	yes	no
yes	buy	buy
no	buy	sell

S2 \ S1	yes	no
yes	buy	sell
no	buy	sell

Figure 1 Bank survey: (a) process configuration, (b)/(c) red/green tactic, (d) session types.

A runtime configuration of processes for this example is shown in Fig. 1(a): the analyzer process A, the tactician process T, the surveyor processes S_1 and S_2, along with their participant processes P_1 and P_2, resp. The processes are connected by the channels u_1, u_2, w_1, w_2, x, and z. The figure shows the communications between the analyzer, surveyors, and participants along these channels, with arrows indicating the message being exchanged. The analyzer sends the message *ask* to surveyor S_1 to request a poll (1). Surveyor S_1 then sends the message *ask* to participant P_1 to get their opinion about buying a share (2). Once the surveyor receives P_1's vote (i.e., either *yes* or *no*) (3), it relays the vote back to the analyzer (4). The analogous communication pattern is repeated between the analyzer and surveyor S_2 and participant P_2 (1'–4'). The final decision whether to *buy* or *sell* (5) of the analyzer is based on the tactic provided by the tactician. For simplicity, we assume that the tactician chooses either a *green* or *red* tactic (0). In the green tactic, the analyzer decides to buy the share if at least one of the surveyors votes yes. In the red tactic, the analyzer buys the stock if the first surveyor votes to buy, regardless of the opinion of the second one (see Fig. 1(b-c)).

Table 1 SINTEGRITY constructs. Upper half: types and terms (**b**efore and **a**fter exchange), operational meaning, and **p**olarity. Lower half: spawn and forward terms and operational meaning.

Session type (b/a)		Process term (b/a)		Description
$x{:}\oplus\{\ell{:}A_\ell\}_{\ell\in L}$	$x{:}A_k$	$x.k; P$	P	provider sends label k along x, continues with P
		$\mathbf{case}\, x(\ell \Rightarrow Q_\ell)_{\ell\in L}$	Q_k	client receives label k along x, continues with Q_k
$x{:}\&\{\ell{:}A_\ell\}_{\ell\in L}$	$x{:}A_k$	$\mathbf{case}\, x(\ell \Rightarrow P_\ell)_{\ell\in L}$	P_k	provider receives label k along x, continues with P_k
		$x.k; Q$	Q	client sends label k along x, continues with Q
$x{:}A\otimes B$	$x{:}B$	$\mathbf{send}\, y\, x; P$	P	provider sends channel $y{:}A$ along x, continues with P
		$z{\leftarrow}\mathbf{recv}\, x; Q_z$	Q_y	client receives channel $y{:}A$ along x, continues with Q_y
$x{:}A\multimap B$	$x{:}B$	$z{\leftarrow}\mathbf{recv}\, x; P_z$	P_y	provider receives channel $y{:}A$ along x, continues with P_y
		$\mathbf{send}\, y\, x; Q$	Q	client sends channel $y{:}A$ along x, continues with Q
$x{:}\mathbf{1}$	-	$\mathbf{close}\, x$	-	provider sends "close" along x and terminates
		$\mathbf{wait}\, x; Q$	Q	client receives "close" along x, continues with Q
$x : Y$	$x : A$	-	-	recursive type definition $Y = A$ (Y occurs in A)
Judgmental rules				
$(x^{\langle c,e\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle c_0, e_0\rangle; Q_x$				spawn X along $x^{\langle c,e\rangle}$ with arguments Δ, substitution γ, and running security $\langle c_0, e_0\rangle$, then continue with Q_x
$x \leftarrow y$				forward $x{:}A$ to $y{:}A$

The protocols for these communications can be specified by the session types shown in Fig. 1(d), using the connectives of Table 1. The connectives are drawn from intuitionistic linear logic and obey the following grammar:

$A, B ::= \oplus\{\ell : A_\ell\}_{\ell\in L} \mid \&\{\ell : A_\ell\}_{\ell\in L} \mid A\otimes B \mid A \multimap B \mid 1 \mid Y,$

where L ranges over finite sets of labels denoted by ℓ and k, amounting to primitive values in our system. Type variable Y is a fixed point whose definition $Y = A$ is given in a global signature Σ. The latter is used to define general recursive types. Recursive types must be *contractive* [23], demanding a message exchange before recurring, and *equi-recursive* [19], avoiding explicit (un)fold messages and relating types up to their unfolding. All three types vote, result, and tactic are recursive.

Table 1 provides an overview of SINTEGRITY types and terms. A crucial characteristic of session-typed processes is that a process *changes* its type along with the messages it exchanges. A process' type therefore always reflects the current protocol state. Table 1 lists state transitions caused by a message exchange in columns 1 and 2 with corresponding process terms in columns 3 and 4. Column 5 describes the computational behavior of a type.

Linearity ensures that every channel connects exactly two processes, thus imposing a *tree* structure on a configuration of processes, as witnessed by Fig. 1(a). We adopt a form of session types corresponding with intuitionistic linear logic, which moreover introduces a distinction between the two processes connected by a channel, identifying one as the *parent* and the other as the *child*, turning the configuration into a *rooted tree*. The parent and child processes have mutually dual perspectives on the protocol of their connecting channel: The child has the perspective of the *provider* and the parent that of a *client*. Column 5 of Table 1 describes the perspective of the client and provider for each type. We assign a polarity to each session type which determines whether the type has a sending semantics or a receiving semantics. For positive types, the provider sends, and the client receives; for negative types, the provider receives, and the client sends. The types with positive polarity are $\oplus\{\ell{:}A_\ell\}_{\ell\in L}$, $A\otimes B$, and 1, and the types with negative polarity are $\&\{\ell{:}A_\ell\}_{\ell\in L}$ and $A \multimap B$.

$$
\begin{array}{l}
w_1{:}\mathsf{vote}\langle\mathbf{bank},\mathbf{guest}\rangle, w_2{:}\mathsf{vote}\langle\mathbf{bank},\mathbf{guest}\rangle, x{:}\mathsf{tactic}\langle\mathbf{bank},\mathbf{guest}\rangle \vdash \mathsf{A} :: z{:}\mathsf{result}\langle\mathbf{bank},\mathbf{bank}\rangle \\
z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x = \\
\mathbf{case}\, x\, (\mathit{green} \Rightarrow w_1.\mathit{ask};\ \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x)) \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x))) \\
\qquad \mid \mathit{red} \Rightarrow w_1.\mathit{ask};\ \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x)) \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad @\langle\mathbf{guest},\mathbf{guest}\rangle
\end{array}
$$

$$
\begin{array}{l}
u{:}\mathsf{vote}\langle\mathbf{guest},\mathbf{guest}\rangle \vdash \mathsf{S} :: w{:}\mathsf{vote}\langle\mathbf{bank},\mathbf{guest}\rangle \\
w \leftarrow \mathsf{S} \leftarrow u = \ \mathbf{case}\, w\, (\mathit{ask} \Rightarrow (w \leftarrow \mathsf{S}' \leftarrow u))@\langle\mathbf{guest},\mathbf{guest}\rangle \\
u{:}\mathsf{vote}\langle\mathbf{guest},\mathbf{guest}\rangle \vdash \mathsf{S}' :: w{:}\oplus\{\mathit{yes}:\mathsf{vote}, \mathit{no}:\mathsf{vote}\}\langle\mathbf{bank},\mathbf{guest}\rangle \\
w \leftarrow \mathsf{S}' \leftarrow u = u.\mathit{ask}; \mathbf{case}\, u(\mathit{yes} \Rightarrow w.\mathit{yes}; (w \leftarrow \mathsf{S}' \leftarrow u) \mid \mathit{no} \Rightarrow w.\mathit{no}; (w \leftarrow \mathsf{S}' \leftarrow u))@\langle\mathbf{guest},\mathbf{guest}\rangle
\end{array}
$$

Figure 2 Secure process implementations of analyzer and surveyor (see Fig. 1), accepted by SINTEGRITY, but rejected by existing IFC session type systems.

Each process in a configuration is uniquely identified by the channel that connects it to its parent, which we also refer to as its *offering* (or *providing*) channel. We consider the session type of a process to be the protocol of its offering channel. For example, the participant process P_1 in Fig. 1 has type vote, which is also the type of the process' offering channel u_1 that connects P_1 to its client S_1. We say that the client S_1 *uses* the channel u_1.

The connectives $\otimes$ and $\multimap$, not used in the example, allow sending channels along channels. Such higher-order channels change the connectivity structure of a configuration: from the perspective of the provider, $\otimes$ turns a child into a sibling and $\multimap$ a sibling into a child. The former is achieved by sending a subtree to the parent and the latter by receiving a subtree from the parent. § 5.4 showcases an example that uses higher-order channels.

It is now time to explore how to implement the processes of our bank survey example. Fig. 2 gives the process definitions of the analyzer and surveyor. A process definition consists of the process signature (first two lines) and body (after =). The first line indicates the typing of channel variables used by the process (left of $\vdash$) and the type of the providing channel variable (right of $\vdash$). The former are going to be child nodes of the process. The second line binds the channel variables. In SINTEGRITY, $\leftarrow$ generally denotes variable bindings. The channels and the process definitions are annotated with confidentiality and integrity levels (e.g., $\langle\mathbf{bank},\mathbf{guest}\rangle$ and $@\langle\mathbf{guest},\mathbf{guest}\rangle$). We will later describe the meaning of these annotations; the reader can safely ignore them for now.

The analyzer first waits to receive a tactic from the tactician along channel x. In either branch (i.e., *green* or *red*), the analyzer proceeds by requesting a vote from surveyors S_1 and S_2, after which it communicates its decision along its offering channel z before recurring. We remark that the notation $z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x$ used for a tail call does not precisely match up with Table 1 because we are deferring a discussion of security annotations and substitutions for security-polymorphic processes to § 5.1. Moreover, a tail call is syntactic sugar for a spawn combined with a forward; i.e., $z \leftarrow \mathsf{A} \leftarrow w_1, w_2, x$ desugars to $z' \leftarrow \mathsf{A} \leftarrow w_1, w_2, x; z \leftarrow z'$.

We implement a surveyor by two processes S and S' to take advantage of SINTEGRITY's support for regrading, as we will detail in § 3.1. The surveyor starts out as process S and calls process S' right after having received the request from its parent, the analyzer.

Suppose that the tactic is a secret that a participant shall not deduce. The implementations in Fig. 2 respect this security condition: the analyzer interacts with the participants via the surveyors the same regardless of the tactic it received. Existing IFC session type systems [7, 20], however, reject these implementations, because they view the analyzer as

tainted as soon as it learns the secret tactic, and disallow further communication with the participants via the surveyor. This paper relaxes this restriction – while preserving PSNI – and allows the tainted surveyor to interact with the participants while putting safeguards in place (synchronization patterns, § 3.2–§ 3.3 and § 5.2) that prevent the surveyor from leaking the tactic.

3 Key ideas

This section develops the main ideas underlying our flexible IFC session type system; the type system and dynamics is given in § 5. The latter is asynchronous, i.e., non-blocking sends and blocking receives (see § 4.2 and § 5.3). An asynchronous semantics allows for a more permissive noninterference statement since message receipt is not observable.

It may be helpful to foreshadow our attacker model (detailed in § 6.1). We assume that an attacker knows the implementation of all processes and can observe messages sent over channels with lower or equal confidentiality level than the attacker. The attacker cannot measure time but can observe the relative order in which messages are sent along different observable channels. As we aim for PSNI, we need to ensure that an attacker is unable to deduce any information from non-reactiveness either.

3.1 Regrading confidentiality

It is now time to consider the red annotations $\langle \mathbf{c}, \mathbf{e} \rangle$ on channels and the green annotations $@\langle \mathbf{c_0}, \mathbf{e_0} \rangle$ on process terms in Fig. 2, where $\mathbf{c}$, $\mathbf{d}$, $\mathbf{c_0}$, and $\mathbf{e_0}$ range over levels in the security lattice $\mathbf{guest} \sqsubseteq \mathbf{alice} \sqsubseteq \mathbf{bank}$ and $\mathbf{guest} \sqsubseteq \mathbf{bob} \sqsubseteq \mathbf{bank}$. We focus on the first components $\mathbf{c}$ and $\mathbf{c_0}$ for now, which denote confidentiality labels. They are adopted from existing IFC session type systems [7, 20], which are based solely on confidentiality.

The first component $\mathbf{c}$ of the pair $\langle \mathbf{c}, \mathbf{e} \rangle$ indicates the *maximal confidentiality* of a process, i.e., the maximal level of secret information the process may ever obtain. As to be expected, the analyzer (A), the tactician (T), and both surveyors (S_1 and S_2) have maximal confidentiality **bank**, as they are affiliated with the bank and have the clearance of knowing the secret tactic. The processes associated with the participants have the lowest maximal confidentiality **guest**, as they must not gain any information about the bank's secrets.

The first component $\mathbf{c_0}$ of the pair $@\langle \mathbf{c_0}, \mathbf{e_0} \rangle$ denotes a process' *running confidentiality*. It denotes the highest level of secret information a process has obtained so far and thus is analogous to the pc label in imperative languages, making the type system flow-sensitive. The running confidentiality is capped by the maximal confidentiality, i.e., $\mathbf{c_0} \sqsubseteq \mathbf{c}$. When defining a process, a programmer must indicate the process' maximal confidentiality as well as the *initial* running confidentiality at which the process starts out when spawned.

An IFC type system increases the running confidentiality accordingly, whenever information of higher confidentiality is received, and disallow sends from senders with a higher or incomparable running confidentiality than the recipient. For example, the analyzer starts with the running confidentiality **guest**. When it receives the secret from the tactician, its running confidentiality increases to **bank**. After the receive, the analyzer can still send the message *ask* to a surveyor as the maximal confidentiality of the surveyor is **bank**. However, as soon as the surveyor receives this message from the analyzer, its running confidentiality increases to **bank**, which prevents it from sending messages to participants, whose maximal confidentiality is **guest**, because $\mathbf{bank} \not\sqsubseteq \mathbf{guest}$.

To address this limitation of existing IFC session type systems, we develop *regrading policies*. A regrading policy is polymorphic in a level f of the security lattice and certifies that, when regrading the running confidentiality to f, any secrets of confidentiality $d_s \not\sqsubseteq f$ learned so far will not affect future communications of confidentiality at most f after regrading.

To convey the regrading policy that a process must obey, we introduce *integrity* annotations, amounting to the second components in the pairs $@\langle \mathbf{c_0}, \mathbf{e_0} \rangle$ and $\langle \mathbf{c}, \mathbf{e} \rangle$. We refer to $\mathbf{e_0}$ as the *running integrity* of the process and to $\mathbf{e}$ as the *minimal integrity* of the process. The running integrity specifies what level a process is allowed to regrade to and is capped by the minimal integrity, i.e., $\mathbf{e_0} \sqsubseteq \mathbf{e}$. For example, the surveyor process S runs at $@\langle \mathbf{bank}, \mathbf{guest} \rangle$ after having received the request from the analyzer, where the running integrity **guest** licenses it to drop its running confidentiality as low as **guest** upon tail-calling, but forces it to obey that policy too. The minimal integrity $\mathbf{e}$ of a process is naturally capped by its maximal confidentiality $\mathbf{c}$, i.e., $\mathbf{e} \sqsubseteq \mathbf{c}$, because a process cannot learn (and thus drop) more secrets than it is licensed to. As a result, a process with maximal confidentiality and minimal integrity $\langle \mathbf{c}, \mathbf{c} \rangle$ effectively amounts to a non-regrading process.

We draw both integrity and confidentiality levels from the same security lattice, but interpret integrity levels *dually*, as usual: the lower a level in the lattice, the higher its integrity[2]. For regrading this means that the lower the level a process regrades to, the stricter the process' policy becomes. The SINTEGRITY type system thus increases the running integrity of a process upon receiving from a process with a higher minimal integrity and disallows sends from a process of a higher or incomparable running integrity than the minimal integrity of the recipient (see § 5).

The process definitions in Fig. 2 only use concrete levels from the security lattice for confidentiality and integrity annotations. To increase code reusability, SINTEGRITY supports *security-polymorphic* process definitions. Such definitions range over security variables for confidentiality and integrity levels and may state constraints on these variables. The constraints must be satisfied upon spawning, which is checked by the SINTEGRITY type checker using a security theory. § 5 expands on security-polymorphic process definitions.

3.2 The need for regrading policies

While a regrading policy licenses regrading, it also imposes restrictions on a process' communication patterns to guarantee noninterference. To distill these restrictions, we next explore insecure implementations of the analyzer-surveyor example from § 2 that do not satisfy PSNI.

3.2.1 Hasty analyzer – optimization may introduce a timing attack

In the red tactic, the decision of the analyzer does not depend on the result provided by the second surveyor. Hence, one may be tempted to optimize the analyzer implementation by refraining from asking the opinion of the second surveyor in the branch corresponding to the red tactic (see $\mathsf{A_H}$ in Fig. 3). As appealing as this optimization seems, it leads to a leak. An attacker of confidentiality level **guest** can simultaneously observe the sequence of messages transmitted along channels u_1 and u_2 of confidentiality **guest**, which connect the participants to the surveyors, and thus, can deduce which secret tactic was chosen: in case of the green tactic, the sequence of messages along u_1 and u_2 has the recurrence $u_1.ask; u_1.(yes/no); u_2.ask; u_2.(yes/no)$,

[2] We adopt the following convention to avoid any confusion: we use "running integrity", "minimal integrity", and "integrity level" for elements in the security lattice, and otherwise just "integrity". Thus, when the integrity level in the lattice increases, the integrity decreases.

$$
\begin{array}{l}
w_1{:}\mathsf{vote}\langle \mathbf{bank}, \mathbf{guest}\rangle, w_2{:}\mathsf{vote}\langle \mathbf{bank}, \mathbf{guest}\rangle, x{:}\mathsf{tactic}\langle \mathbf{bank}, \mathbf{guest}\rangle \vdash \mathsf{A_H} :: z{:}\mathsf{result}\langle \mathbf{bank}, \mathbf{bank}\rangle \\
z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x = \\
\mathbf{case}\, x\, (\mathit{green} \Rightarrow w_1.\mathit{ask};\ \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x))) \\
\qquad\quad \mid \mathit{red} \Rightarrow w_1.\mathit{ask}; \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2{:}, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A_H} \leftarrow w_1, w_2, x)))\ @\langle \mathbf{guest}, \mathbf{guest}\rangle
\end{array}
$$

$$
\begin{array}{l}
w_1{:}\mathsf{vote}\langle \mathbf{bank}, \mathbf{guest}\rangle, w_2{:}\mathsf{vote}\langle \mathbf{bank}, \mathbf{guest}\rangle, x{:}\mathsf{tactic}\langle \mathbf{bank}, \mathbf{guest}\rangle \vdash \mathsf{A_R} :: z{:}\mathsf{result}\langle \mathbf{bank}, \mathbf{bank}\rangle \\
z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x = \\
\mathbf{case}\, x\, (\mathit{green} \Rightarrow w_1.\mathit{ask};\ \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow w_2.\mathit{ask}; \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x))) \\
\qquad\quad \mid \mathit{red} \Rightarrow w_1.\mathit{ask}; w_2.\mathit{ask};\ \mathbf{case}\, w_1\, (\mathit{yes} \Rightarrow \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{buy};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{no} \Rightarrow \mathbf{case}\, w_2\, (\mathit{yes} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{no} \Rightarrow z.\mathit{sell};\ (z \leftarrow \mathsf{A_R} \leftarrow w_1, w_2, x)))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad @\langle \mathbf{guest}, \mathbf{guest}\rangle
\end{array}
$$

Figure 3 Insecure hasty analyzer $\mathsf{A_H}$ and reckless analyzer $\mathsf{A_R}$, rejected by SINTEGRITY.

whereas it has the recurrence $u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no})$ for the red tactic. Observing, for example, the sequence $u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no}); u_2.\mathit{ask}; u_2.(\mathit{yes}/\mathit{no}); u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no})$, the attacker can deduce that the first tactic used was green and the second one was red. These leaks constitute *timing attacks* because the attacker cannot deduce the secret by only looking at a single channel, but needs to observe the relative timing of messages passed along two or more channels.

3.2.2 Reckless analyzer – be careful with synchronization

The previous example shows that a send along a channel, present in one branch, but omitted from another, may lead to a leak. One may naively suspect that these leaks only involve sends. The analyzer version $\mathsf{A_R}$ in Fig. 3 showcases the opposite: mismatched receives are at least as dangerous as mismatched sends. In the original implementation (Fig. 2), the analyzer synchronizes the communications of surveyors and participants across branches, ensuring, in particular, that the second participant always casts their vote after the first. The reckless analyzer $\mathsf{A_R}$ breaks this synchronization in the red branch by swapping the order of $\mathbf{case}\, w_1$ and $w_2.\mathit{ask}$. This minimal change allows the two surveyors to run concurrently when the tactic is red and produce the sequence of messages $u_2.\mathit{ask}; u_2.(\mathit{yes}/\mathit{no}); u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no})$ along channels u_1 and u_2, a sequence that is impossible to produce in the green tactic (recall that receives are blocking, but sends are not). Both $\mathsf{A_H}$ and $\mathsf{A_R}$ leak the secret with a timing attack, i.e., the simultaneous observation of the relative order of sends along several channels.

There is a subtle connection between timing attacks and leaks due to the non-reactivity of a process. For instance, let us assume that the second participant loops internally and never casts its vote. The attacker can then deduce the secret tactic in the hasty implementation of the analyzer by only observing the communications of the first participant along u_1: the sequence $u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no}); u_1.\mathit{ask}; u_1.(\mathit{yes}/\mathit{no})$ indicates that the prior tactic was red. A similar scenario holds for the reckless analyzer when the first participant is non-reactive.

3.3 Regrading policies in a nutshell

Our model allows the running confidentiality of a process to be dropped as low as its running integrity. Performing such a venturous act, needs a corresponding safety net in place: a regrading policy that is polymorphic in the running integrity to preserve noninterference. The examples in § 3.2 suggest that a regrading policy must enforce the following properties:

1. The continuation of a process after regrading must not depend on any secret higher than or incomparable to its running integrity. That is, when branching on a secret $\mathbf{d}_s$, the same process must be spawned for the recursive call in every branch, if that process regrades to a level $\mathbf{e}_0$ such that $\mathbf{d}_s \not\sqsubseteq \mathbf{e}_0$.
2. Whether a process reaches its regrading point or not must not depend on any secret higher than or incomparable to its running integrity.

The latter property is violated in both analyzer implementations of Fig. 3, amounting to a leak. In the hasty implementation $\mathsf{A_H}$, the second surveyor only gets to the regrading point if the secret tactic is green. In the reckless implementation $\mathsf{A_R}$, if the secret tactic is green, the second surveyor gets to the regrading point only if the first participant casts their vote, whereas if the secret is red, there is no such chaining.

The above properties capture semantically what conditions secure processes that employ regrading must meet to observe PSNI. In § 5.2 we develop static checks that, when satisfied by a process, ensure that the process also meets these semantic conditions. We refer to those checks as *synchronization pattern* checks, and they are enforced by the SINTEGRITY type checker. The pattern checks are of the form $\Psi \vDash P \sim_{\langle d,f \rangle} Q$ and synchronize P and Q in terms of their communication actions: if P outputs along channel x, so must Q, and if P inputs along channel x, so must Q, and vice versa. The pattern checks are invoked pairwise for every two branches, P_i and P_j, in a **case** statement, requiring that $\Psi \vDash P_i \sim_{\langle d,f \rangle} P_j$. The check is conditioned on the running confidentiality d and running integrity f at the branching point.

An important feature of our regrading policies is that they are *compositional*. We take advantage of the fact that intuitionism imposes a rooted tree structure on process configurations and require that a configuration aligns with the security lattice: for every child process and parent process with maximal confidentiality and minimal integrity $\langle \mathbf{c}, \mathbf{e} \rangle$ and $\langle \mathbf{c}', \mathbf{e}' \rangle$, resp., it must hold that $\langle \mathbf{c}, \mathbf{e} \rangle \sqsubseteq \langle \mathbf{c}', \mathbf{e}' \rangle$, ensuring that a child process can learn at most as much as its parent and has at least an as stringent regrading policy as its parent. We design our type system to preserve this property as an invariant.

4 Blueprint for Formal Development

Before delving into the formal development, we review the statics and dynamics of a vanilla intuitionistic session type system and give a roadmap for the upcoming technical sections. We use the intuitionistic session type system introduced by Toninho et al. [9, 41] as our vanilla intuitionistic session type system. SINTEGRITY enhances such a vanilla session type system with confidentiality and integrity annotations to establish PSNI. SINTEGRITY adopts the former from existing intuitionistic IFC session type systems [7, 20]. The integrity annotations as well as the synchronization patterns are contributions unique to SINTEGRITY. The addition requires us to define the relationships between all these levels, expressed as invariants, and the development of synchronization patterns. Similar to the system in [7] our language supports general recursion and allows processes to be polymorphic in confidentiality levels. SINTEGRITY extends label polymorphism to also accommodate integrity levels.

4.1 Vanilla intuitionistic session types – statics

Process terms and session types are built by the grammar in § 2 and Table 1. The process typing judgment is of the form $\Delta \vdash_\Sigma P :: x{:}A$, to be read as: *"Process P provides a session of type A along channel x, given the typing of sessions offered along channels in Δ.* Δ is a *linear* typing context consisting of the channels connecting P to its children, and x connects P to its parent. The global signature Σ includes recursive type definitions and process definitions.

4.1.1 Process term typing

Fig. 4 lists the process term typing rules. The parts in red are specific to SINTEGRITY and can be ignored for now; we discuss them in § 5. As is usual in intuitionistic linear session type languages, the rules are given in a sequent calculus. When read from bottom to top, the rules closely follow the behavior described in Table 1: right rules describe a type from the point of view of a provider, and left rules from the point of view of a client. For example, rule $\oplus_{R_1}$ describes the behavior of the process that provides a channel with the protocol $\oplus\{\ell{:}A_\ell\}_{\ell\in L}$: it chooses a label $k{\in}L$ and sends it to the client along channel x, and then continues by checking process P providing A_k in the premise. Note that the typing rules $\oplus_{R_1}$ and $\oplus_{R_2}$ are identical, ignoring the security annotations. Rule Fwd ensures that the type of the two channels involved in forwarding is the same. Rule Spawn spawns a new child process X along the fresh channel x; it first checks that X is defined in the signature (first premise) and thus is well-typed and then continues with type-checking the continuation Q (last premise).

4.1.2 Signature checking

To support general recursive types, we employ a global signature Σ comprised of all process definitions. Each process definition is typed individually, assuming that the other processes in the signature are well-typed. The signature also comprises recursive type definitions. When typing a process with a recursive protocol, the signature is consulted to unfold the definition.

For example, the signature for the bank survey example in § 2 consists of the definitions for processes A, S, and S′ as shown in Fig. 1 and the definition of recursive types as shown in Fig. 1(d). In our formal development, we use a more concise syntax for process definitions than what is shown in Fig. 1. In particular, we write them in the form of $\Delta \vdash X = P{::}(z{:}A)$. For instance, the concise version of the process definition for process S in Fig. 1, ignoring its security annotations, is $u{:}\mathsf{vote} \vdash \mathsf{S} = \mathit{case}\, w\ (\mathit{ask} \Rightarrow (w \leftarrow \mathsf{S}' \leftarrow u)){::}w{:}\mathsf{vote}$.

Type checking starts with typing the signature by the rules listed in Fig. 5; again, ignore the parts in red for now, as they will be discussed later in § 5. The rules are in a sequent calculus and should be read from bottom to top. Rule Σ_3 ensures that each process definition in the signature is well-typed. It invokes the process term typing judgment for a process definition relative to the entire global signature Σ (fifth premise) and continues with checking the rest of the signature (sixth premise). Rule Σ_2 ensures that all recursive types in the signature are well-formed via its first premise, the judgment $\Vdash_\Sigma A$ **wfmd**. This judgment denotes a well-formed session type definition, which, if recursive, must be *equi-recursive* [19] and *contractive*. Equi-recursiveness ensures that types are related up to their unfolding without requiring explicit (un)fold messages (see rules TVar_R and TVar_L). Contractiveness demands an exchange before recurring.

4.2 Vanilla intuitionistic session types – dynamics

At runtime, process definitions result in a configuration of processes structured as a forest of rooted trees. The nodes in the forest represent runtime processes and messages, denoted as $\mathbf{proc}(y_\alpha; P)$ and $\mathbf{msg}(M)$, resp. We use metavariables $\mathcal{C}$ and $\mathcal{D}$ to refer to a configuration and formally define it as a set of runtime processes and messages (the nodes in the tree). The connection between the nodes will be inferred through configuration typing. In $\mathbf{proc}(y_\alpha, P)$, the metavariable y_α represents the process' offering channel, and P represents the process' source code (where free variables have been substituted by channels). Runtime messages $\mathbf{msg}(M)$ are a special form of processes created to model asynchronous communication: we

implement asynchronous sends by spawning off the message $\mathbf{msg}(M)$ that carries the sent message M. A sent message M can be of the form $x.k$, $\mathbf{send}\, y\, x$, or $\mathbf{close}\, x$, corresponding to label output, channel output, and a termination message, resp.

Runtime channels y_α are annotated with a generation subscript α, which distinguishes them from channel variables y used in the statics. Using generation subscripts, we can ensure that both the sender and receiver agree on a new name for the continuation channel without explicitly passing the name in a message. We will see an example of using generation subscripts in the next paragraph.

4.2.1 Asynchronous dynamics

We chose an asynchronous semantics for SINTEGRITY because it weakens the attacker model, allowing a more permissive IFC enforcement, and is also a sensible model for practical purposes. The dynamics is given in Fig. 8 in terms of multiset rewriting rules [14] (again, for now the parts in red can be ignored). Multiset rewriting rules express the dynamics as state transitions between configurations and are *local* in that they only mention the parts of a configuration they rewrite.

For example, in case of $\otimes_{\mathsf{snd}}$, the provider $\mathbf{proc}(y_\alpha, \mathbf{send}\, x_\beta\, y_\alpha; P)$ spawns off the message process $\mathbf{msg}(\mathbf{send}\, x_\beta\, y_\alpha)$, indicating that the channel x_β is sent over channel y_α. Since sends are non-blocking, the provider steps to its continuation $\mathbf{proc}(y_{\alpha+1}, ([y_{\alpha+1}/y_\alpha]P))$, allocating a new generation $\alpha{+}1$ of the carrier channel y_α. In $\otimes_{\mathsf{recv}}$, upon receipt of the message, the receiving client process $\mathbf{proc}(y_\alpha, w \leftarrow \mathbf{recv}\, y_\alpha; P)$ will increment the generation of the carrier channel in its continuation. The scenario is similar for $\oplus_{\mathsf{snd}}$ and $\oplus_{\mathsf{rcv}}$, but the sent message is a label in this case, and similar for $\multimap_{\mathsf{snd}}, \multimap_{\mathsf{rcv}}$ and $\&_{\mathsf{snd}}, \&_{\mathsf{rcv}}$, except that in these cases the sender is the client and the receiver the provider. In the rules for the termination protocol, i.e., 1_{snd} and 1_{rcv}, there is no continuation channel. Rule SPAWN creates a process offering along a fresh runtime channel x_0 by looking up the definition of the spawnee in the signature.

The dynamics for the forwarding process $\mathbf{proc}(y_\alpha, y_\alpha \leftarrow x_\beta)$ is often described as fusing the two channels, y_α and x_β. We, however, represent forward as syntactic sugar by including forwarder processes defined by structural induction on the type of the channels involved in the forward, amounting to an identity expansion. The reader may see the TR for the details.

4.2.2 Configuration typing

The configuration typing judgment is of the form $\Delta \Vdash_\Sigma \mathcal{C} :: \Delta'$ indicating that the configuration $\mathcal{C}$ provides sessions along the channels in Δ', using sessions provided along channels in Δ. Δ and Δ' are both linear contexts, consisting of actual runtime channels of the form $y_\alpha{:}B$. We often use the term *open configurations* to emphasize that our configurations may have external free channels in both Δ and Δ' to communicate with the environment. This is in contrast to restricting Δ to be an empty context, which means the configuration only has external free channels to communicate with a client.

Fig. 7 shows the typing rules, enforcing that the configuration is structured as a forest and the source code of each node is well-typed. For brevity, Fig. 7 omits a channel's generation as well as Σ, which is fixed. The **emp** rule types an empty forest. The **comp** rule types each tree in the forest. The **proc** rule and the **msg** rule check the well-typedness of the root node of a tree when it is a process or message, resp., using the last premises. Well-typedness of the remaining forest is checked by the eighth and fourth premise of the latter two rules, resp. The last premise of the **msg** rule calls message typing rules, which we provide in TR-Sect. 3.3.

The typing rules ensure progress and preservation, i.e., the dynamics can always step an open configuration $\Delta \Vdash \mathcal{C} :: \Delta'$ to $\Delta \Vdash \mathcal{C}' :: \Delta'$.

4.3 Roadmap for SINTEGRITY

To develop the ideas discussed in § 3 and establish PSNI, we supplement the vanilla type system with a security layer. Here, we provide a roadmap to the key parts of our development.

4.3.1 Regrading policy type system

The first step in our formal development is to enrich the process term typing judgment with security levels as $\Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}A\langle c, e\rangle$. Here, Ψ denotes a security theory which includes the security lattice and polymorphic confidentiality and integrity variables. The pair $\langle c_0, e_0\rangle$ denotes the running confidentiality and integrity of the process, aka its taint level. The pair $\langle c, e\rangle$ denotes the max confidentiality and min integrity of the process. Similarly, each channel in Δ is annotated with a pair of confidentiality and integrity levels denoting its provider's max confidentiality and min integrity.

Similarly, we use security labels to annotate configurations and configuration typing judgments. In particular, runtime processes in configuration $\mathcal{C}$ now have the form $\mathbf{proc}(y_\alpha\langle c, e\rangle, P@\langle c', e'\rangle)$, where $\langle c, e\rangle$ is the pair of max confidentiality and min integrity of the process, and $\langle c', e'\rangle$ is the pair of its running confidentiality and integrity.

The typing rules include security constraints highlighted in red – the ones we have been ignoring in § 4.1. The purpose of these security annotations is to (i) ensure that the taint levels are propagated correctly, (ii) prevent a tainted process from sending information to a process with a lower max confidentiality/higher min integrity, (iii) ensure that a process regrades its running confidentiality only as low as its running integrity, and (iv) verify that the process indeed adheres to the policy enforced by its running integrity. The first three conditions are enforced by imposing the security constraints on the process term typing rules in Fig. 4. The last check is enforced by the synchronization pattern checks in Fig. 6.

4.3.2 PSNI via a logical relation

Our ultimate goal is to prove that well-typed SINTEGRITY processes enjoy PSNI. We prove PSNI as an equivalence up to an attacker's confidentiality level ξ using a logical relation, which then delivers a process bisimulation.

To define PSNI for an open configuration in the shape of a tree $\Psi_0; \Delta \Vdash \mathcal{D} :: u_\alpha{:}T\langle c, e\rangle$, given a global security lattice Ψ_0 fixed for an application, we consider the external free channels $y_\beta{:}B\langle c', e'\rangle \in \Delta, u_\alpha{:}T\langle c, e\rangle$ with max confidentiality $c' \sqsubseteq \xi$. We call the set of these channels that connect a configuration to its environment and that are observable to an attacker, the *confidentiality interface.*

Such an open configuration satisfies noninterference if, when composed with different high-confidentiality processes, behaves the same along the confidentiality interface. We prove that all well-typed open configurations enjoy PSNI by designing a logical relation and showing that (i) all well-typed configurations are self-related (fundamental theorem, Thm. 1) and (ii) any two related configurations are bisimilar (adequacy theorem, Thm. 3).

To prove these results, our logical relation needs to consider some free channels in $\Delta, u_\alpha{:}T\langle c, e\rangle$ that are not directly observable in terms of their confidentiality but can have an observable effect due to their integrity. We thus define a superset of the confidentiality interface that additionally contains channels $y_\beta{:}B\langle c', e'\rangle \in \Delta, u_\alpha{:}T\langle c, e\rangle$ with min integrity $e' \sqsubseteq \xi$. We call this interface the *integrity interface.*

5 Regrading policy type system

This section formalizes SINTEGRITY's type system with synchronization patterns and asynchronous dynamics. SINTEGRITY supports security-polymorphic process definitions, an example of which is discussed in § 5.4.

5.1 Process term typing

Let us recall the process term typing judgment from § 4.3:

$$\Psi; \Delta \vdash_{\Sigma} P@\langle c_0, e_0 \rangle :: x{:}A\langle c, e \rangle.$$

We read it as: *"Process P, with maximal confidentiality and minimal integrity $\langle c, e \rangle$ and running confidentiality and integrity $\langle c_0, e_0 \rangle$, provides a session of type A along channel x, given the typing of sessions offered along channels in Δ and given a security theory Ψ".* Δ is a *linear* typing context with the grammar $\Delta ::= \cdot \mid x{:}A\langle c, e \rangle, \Delta$. A security theory Ψ is used for type checking security-polymorphic process definitions. It consists of the global security lattice Ψ_0 which is fixed for an application, security variables ψ, and constraints on the variables (see § 5.4 and Sect. 2 in the TR).

We impose the following properties on the typing judgment, as discussed in detail in § 3. These properties are maintained by typing as invariants. When reading them, note that "high integrity" and "low confidentiality" both mean a "lower level" in the security lattice.

(a) $\forall y{:}B\langle d, f \rangle \in \Delta. \Psi \Vdash d \sqsubseteq c, \Psi \Vdash f \sqsubseteq e$: ensuring that a child process can learn at most as much as its parent and has at least an as stringent regrading policy as its parent.

(b) $\Psi \Vdash c_0 \sqsubseteq c$ and $\Psi \Vdash e_0 \sqsubseteq e$: ensuring that a process knows at most as much as it is licensed to and adheres to at least an as stringent regrading policy as it promises.

(c) $\Psi \Vdash e_0 \sqsubseteq c_0$ and $\Psi \Vdash e \sqsubseteq c$: ensuring that a process cannot drop more secrets than it knows and is licensed to learn, resp.

Moreover, the typing rules for input and output have to conform to the following schema to make sure that the running confidentiality and running integrity correctly reflect the taint level and that a tainted process does not leak information via a send:

(1) **after** receiving a message, the running confidentiality and running integrity of the receiving process must be increased to **at least** the maximal confidentiality and minimal integrity of the sending process, and

(2) **Before** sending a message, the running confidentiality and running integrity of the sending process must be **at most** the maximal confidentiality and minimal integrity of the receiving process.

Conforming to this schema leads to the premises of the form $\Psi \Vdash \langle d_1, f_1 \rangle = \langle c, e \rangle \sqcup \langle d_0, f_0 \rangle$ and $\Psi \Vdash \langle d_0, f_0 \rangle \sqsubseteq \langle c, e \rangle$ to meet condition (1) and (2), resp., above. The judgments are defined formally in Sect. 2 in the TR.

It is time to consider the red security annotations of the typing rules in Fig. 4. We explain how the rules satisfy conditions (1) and (2) above:

- $\oplus$: There are two versions of the right rule for $\oplus$. Both versions establish condition (2) on sends without extra premises by the invariant (b). The difference between the two versions lies in whether $\Psi \Vdash c = e$ is derivable or not derivable ($\Psi \nVdash c = e$). If $\Psi \Vdash c = e$ is derivable, then rule $\oplus_{R_1}$ applies; if it is not, rule $\oplus_{R_2}$ applies. In the former case, the client of x, on the receiving side, adjusts its running integrity to at least $e{=}c$ upon receiving the sent message, and thus, it cannot regrade to a lower (or unrelated) level than c. In the latter case, the min integrity e of the process is strictly lower than its max confidentiality c. This means that the client of x might, in fact, continue to have

$$\frac{\Psi \Vdash c = e \qquad k \in L \qquad \Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}A_k\langle c, e\rangle}{\Psi; \Delta \vdash_\Sigma (x^{\langle c,e\rangle}.k; P)@\langle c_0, e_0\rangle :: x{:}\oplus\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle}\ \oplus_{R_1}$$

$$\frac{\Psi \not\Vdash c = e \qquad \forall i,j \in L.\, A_i = A_j \qquad k \in L \qquad \Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}A_k\langle c, e\rangle}{\Psi; \Delta \vdash_\Sigma (x^{\langle c,e\rangle}.k; P)@\langle c_0, e_0\rangle :: x{:}\oplus\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle}\ \oplus_{R_2}$$

$$\frac{\begin{array}{c}\Psi \Vdash \langle d_1, f_1\rangle = \langle c, e\rangle \sqcup \langle d_0, f_0\rangle \\ \forall k \in L \qquad \Psi; \Delta, x{:}A_k\langle c, e\rangle \vdash_\Sigma Q_k@\langle d_1, f_1\rangle :: z{:}C\langle d, f\rangle \qquad \forall i,j \in L.\, \Psi \vDash Q_i \sim_{\langle d_1, f_1\rangle} Q_j\end{array}}{\Psi; \Delta, x{:}\oplus\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle \vdash_\Sigma (\mathbf{case}\, x^{\langle c,e\rangle}(\ell \Rightarrow Q_\ell)_{\ell\in L})@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \oplus_L$$

$$\frac{\forall k \in L \qquad \Psi; \Delta \vdash_\Sigma P_k@\langle c, e\rangle :: x{:}A_k\langle c, e\rangle \qquad \forall i,j \in L.\, \Psi \vDash P_i \sim_{\langle c,e\rangle} P_j}{\Psi; \Delta \vdash_\Sigma (\mathbf{case}\, x^{\langle c,e\rangle}(\ell \Rightarrow P_\ell)_{\ell\in L})@\langle c_0, e_0\rangle :: x{:}\&\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle}\ \&_R$$

$$\frac{\Psi \Vdash c = e \qquad \Psi \Vdash \langle d_0, f_0\rangle \sqsubseteq \langle c, e\rangle \qquad k \in L \qquad \Psi; \Delta, x{:}A_k\langle c, e\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}{\Psi; \Delta, x{:}\&\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle \vdash_\Sigma (x^{\langle c,e\rangle}.k; Q)@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \&_{L_1}$$

$$\frac{\begin{array}{c}\Psi \not\Vdash c = e \qquad \forall i,j \in L.\, A_i = A_j \\ \Psi \Vdash \langle d_0, f_0\rangle \sqsubseteq \langle c, e\rangle \qquad k \in L \qquad \Psi; \Delta, x{:}A_k\langle c, e\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle\end{array}}{\Psi; \Delta, x{:}\&\{\ell{:}A_\ell\}_{\ell\in L}\langle c, e\rangle \vdash_\Sigma (x^{\langle c,e\rangle}.k; Q)@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \&_{L_2}$$

$$\frac{\Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}B\langle c, e\rangle}{\Psi; \Delta, y{:}A\langle d, f\rangle \vdash_\Sigma (\mathbf{send}\, y\, x^{\langle c,e\rangle}; P)@\langle c_0, e_0\rangle :: x{:}A \otimes B\langle c, e\rangle}\ \otimes_R$$

$$\frac{\Psi \Vdash \langle d_1, f_1\rangle = \langle c, e\rangle \sqcup \langle d_0, f_0\rangle \qquad \Psi; \Delta, x{:}B\langle c, e\rangle, y{:}A\langle c, e\rangle \vdash_\Sigma Q@\langle d_1, f_1\rangle :: z{:}C\langle d, f\rangle}{\Psi; \Delta, x{:}A \otimes B\langle c, e\rangle \vdash_\Sigma (y^{\langle c,e\rangle} \leftarrow \mathbf{recv}\, x^{\langle c,e\rangle}; Q_{y^{\langle c,e\rangle}})@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \otimes_L$$

$$\frac{\Psi; \Delta, y{:}A\langle c, e\rangle \vdash_\Sigma P@\langle c, e\rangle :: x{:}B\langle c, e\rangle}{\Psi; \Delta \vdash_\Sigma (y^{\langle c,e\rangle} \leftarrow \mathbf{recv}\, x^{\langle c,e\rangle}; P_{y^{\langle c,e\rangle}})@\langle c_0, e_0\rangle :: x{:}A \multimap B\langle c, e\rangle}\ \multimap_R$$

$$\frac{\Psi \Vdash \langle d_0, f_0\rangle \sqsubseteq \langle c, e\rangle \qquad \Psi; \Delta, x{:}B\langle c, e\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}{\Psi; \Delta, x{:}A \multimap B\langle c, e\rangle, y{:}A\langle c, e\rangle \vdash_\Sigma (\mathbf{send}\, y\, x^{\langle c,e\rangle}; Q)@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \multimap_L$$

$$\frac{\Psi \Vdash \langle c_1, e_1\rangle = \langle c_2, e_2\rangle}{\Psi; y{:}A\langle c_1, e_1\rangle \vdash_\Sigma (x^{\langle c_2,e_2\rangle} \leftarrow y^{\langle c_1,e_1\rangle})@\langle c_0, e_0\rangle :: x{:}A\langle c_2, e_2\rangle}\ \textsc{Fwd}$$

$$\frac{\begin{array}{c}\Psi'; \Delta_1' \vdash_\Sigma X = P@\langle \psi_0, \omega_0\rangle :: x{:}A\langle \psi, \omega\rangle \in \Sigma \\ \Psi \Vdash \gamma : \Psi' \qquad \hat{\gamma}(\Delta_1') = \Delta_1 \qquad \Psi \Vdash \langle \hat{\gamma}(\psi), \hat{\gamma}(\omega)\rangle \sqsubseteq \langle d, f\rangle \\ \Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\psi_0) \qquad \Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0) \qquad \Psi; \Delta_2, x{:}A\langle \hat{\gamma}(\psi), \hat{\gamma}(\omega)\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle\end{array}}{\Psi; \Delta_1, \Delta_2 \vdash_\Sigma ((x^{\langle \hat{\gamma}(\psi), \hat{\gamma}(\omega)\rangle} \leftarrow X[\gamma] \leftarrow \Delta_1)@\langle \hat{\gamma}(\psi_0), \hat{\gamma}(\omega_0)\rangle; Q_x)@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \textsc{Spawn}$$

$$\frac{}{\Psi; \cdot \vdash_\Sigma (\mathbf{close}\, x^{\langle c,e\rangle})@\langle c_0, e_0\rangle :: x{:}\mathbf{1}\langle c, e\rangle}\ \mathbf{1}_R$$

$$\frac{\Psi \Vdash \langle d_1, f_1\rangle = \langle c, e\rangle \sqcup \langle d_0, f_0\rangle \qquad \Psi; \Delta \vdash_\Sigma Q@\langle d_1, f_1\rangle :: z{:}C\langle d, f\rangle}{\Psi; \Delta, x{:}\mathbf{1}\langle c, e\rangle \vdash_\Sigma (\mathbf{wait}\, x^{\langle c,e\rangle}; Q)@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \mathbf{1}_L$$

Figure 4 Process term typing rules of SINTEGRITY.

its running integrity as low as $e \sqsubset c$ and, at some point in the future, drop its running confidentiality to e and start sending to channels with lower (or unrelated) confidentiality than c. The additional premise $\forall i,j \in L.\, A_i = A_j$ in $\oplus_{R_2}$ prevents potential leaks through different continuation protocols at that future point, i.e., it ensures that the client's future communications with channels of lower confidentiality level than c do not depend on the continuation protocol chosen now.

$$\frac{Y = A \in \Sigma \qquad \Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}A\langle c, e\rangle}{\Psi; \Delta \vdash_\Sigma P@\langle c_0, e_0\rangle :: x{:}Y\langle c, e\rangle}\ \text{TVAR}_R$$

$$\frac{Y = A \in \Sigma \qquad \Psi; \Delta, x{:}A\langle c, e\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}{\Psi; \Delta, x{:}Y\langle c, e\rangle \vdash_\Sigma Q@\langle d_0, f_0\rangle :: z{:}C\langle d, f\rangle}\ \text{TVAR}_L \qquad \frac{}{\Vdash_{\Sigma;\Psi_0} (\cdot)\ \textbf{sig}}\ \Sigma_1$$

$$\frac{\Vdash_\Sigma A\ \textbf{wfmd} \qquad \Vdash_{\Sigma;\Psi_0} \Sigma'\ \textbf{sig}}{\Vdash_{\Sigma;\Psi_0} Y = A, \Sigma'\ \textbf{sig}}\ \Sigma_2$$

$$\frac{\begin{array}{c}\forall i \in \{1 \dots n\}. \Psi \Vdash \langle \psi_i, \omega_i\rangle \sqsubseteq \langle \psi, \omega\rangle,\ \Psi \Vdash \omega_i \sqsubseteq \psi_i \\ \Psi \Vdash \langle \psi_0, \omega_0\rangle \sqsubseteq \langle \psi, \omega\rangle \qquad \Psi \Vdash \omega_0 \sqsubseteq \psi_0 \qquad \Psi \Vdash \omega \sqsubseteq \psi \\ \Psi; y_1{:}B_1\langle \psi_1, \omega_1\rangle, \dots, y_n{:}B_n\langle \psi_n, \omega_n\rangle \vdash_\Sigma P@\langle \psi_0, \omega_0\rangle :: x{:}A\langle \psi, \omega\rangle \qquad \Vdash_{\Sigma;\Psi_0} \Sigma'\ \textbf{sig}\end{array}}{\Vdash_{\Sigma;\Psi_0} \Psi; y_1{:}B_1\langle \psi_1, \omega_1\rangle, \dots, y_n{:}B_n\langle \psi_n, \omega_n\rangle \vdash X = P@\langle \psi_0, \omega_0\rangle :: x{:}A\langle \psi, \omega\rangle, \Sigma'\ \textbf{sig}}\ \Sigma_3$$

Figure 5 Signature checking rules of SINTEGRITY.

The first premise of rule $\oplus L$ updates the running integrity and confidentiality based on x's security levels to enforce condition (1) for receives. Moreover, as explained in § 3.3, the third premise invokes the pattern check pairwise for every two branches conditioned on the running confidentiality d_1 and running integrity f_1 after the receive. We detail the synchronization pattern check rules later in § 5.2.

- &: The left and right rules for & are dual to $\oplus$, except that the sends in $\&_{L_1}$ and $\&_{L_2}$ have to be guarded by their second and third premises, resp., to ensure condition (2) on sends. In $\&_R$, the updated running confidentiality and running integrity is equal to the max confidentiality and max integrity by invariant (b).
- $\otimes$, $\multimap$, 1: The rules for the rest of the connectives use the same set of premises to ensure conditions (1) and (2). Rules $\otimes_R$ and $\multimap_L$, moreover, ensure that a channel can be sent over another channel only if they have the same security levels.
- FWD: The forward rule requires that the security levels of the involved channels match.
- SPAWN: The rule relies on an order-preserving substitution $\Psi \Vdash \gamma : \Psi'$, guaranteeing that the security terms provided by the spawner comply with the order expected among those terms by the spawnee. The substitution maps the security terms in the context in the signature to the one provided by the spawner, i.e., $\hat{\gamma}(\Delta_1') = \Delta_1$. The rule also establishes invariants (a)-(c) for the newly spawned process via the premise $\Psi \Vdash \langle \hat{\gamma}(\psi), \hat{\gamma}(\omega)\rangle \sqsubseteq \langle d, f\rangle$. The running confidentiality and the running integrity of the spawned process will result from applying the substitution to the corresponding levels in the signature, i.e., $\hat{\gamma}(\psi_0)$ and $\hat{\gamma}(\omega_0)$, resp. The premises $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\psi_0)$ and $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0)$ allow the newly spawned process to start its running confidentiality and integrity at least at the spawner's running integrity f_0. This facilitates regrading to f_0 in case of a tail call. Note that $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0)$ prevents the spawnee from employing more pattern checks than the spawner because the spawnee would otherwise be affected by the spawners negligence.

Signature checking. The syntax of process definitions in the signature is also enhanced with the security levels and is of the form $\Psi; \Delta \vdash X = P@\langle \psi_0, \omega_0\rangle{::}(z{:}A\langle \psi, \omega\rangle)$. Fig. 5 lists the signature checking rules. Signature checking happens relative to a globally fixed security lattice Ψ_0 of concrete security levels. Rule Σ_3 initiates type-checking of a process definition via its fifth premise and enforces invariants (a)-(c) on the process via the first four premises.

$$\frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash \uparrow_{x^{\langle c,e\rangle}} .P \sim_{\langle d,f\rangle} Q}\ \textsc{Unsync}_1 \qquad \frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash P \sim_{\langle d,f\rangle} \uparrow_{x^{\langle c,e\rangle}} .Q}\ \textsc{Unsync}_2$$

$$\frac{\Psi \Vdash d \sqsubseteq f}{\Psi \vDash P \sim_{\langle d,f\rangle} Q}\ \textsc{Unsync}_3$$

$$\frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e_0 \quad \forall y{:}B\langle c',e'\rangle \in \Delta.\Psi \Vdash d \sqsubseteq e' \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash (x^{\langle c,e\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle c_0,e_0\rangle; P_x \sim_{\langle d,f\rangle} Q}\ \textsc{Unsync-Spawn}_1$$

$$\frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e_0 \quad \forall y{:}B\langle c',e'\rangle \in \Delta.\Psi \Vdash d \sqsubseteq e' \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash P \sim_{\langle d,f\rangle} (x^{\langle c,e\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle c_0,e_0\rangle; Q_x}\ \textsc{Unsync-Spawn}_2$$

$$\textsc{SndLab}\ \frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \not\Vdash d \sqsubseteq e \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash x^{\langle c,e\rangle}.k; P \sim_{\langle d,f\rangle} x^{\langle c,e\rangle}.\ell; Q}$$

$$\textsc{RcvLab}\ \frac{\Psi \not\Vdash d \sqsubseteq f \quad \forall j \in I, k \in L.\Psi \vDash P_j \sim_{\langle d,f\sqcup e\rangle} Q_k}{\Psi \vDash \textbf{case}\, x^{\langle c,e\rangle}(\ell \Rightarrow P_\ell)_{\ell \in I} \sim_{\langle d,f\rangle} \textbf{case}\, x^{\langle c,e\rangle}(\ell \Rightarrow Q_\ell)_{\ell \in L}}$$

$$\textsc{SndChn}\ \frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \not\Vdash d \sqsubseteq e \quad \Psi \vDash P \sim_{\langle d,f\rangle} Q}{\Psi \vDash \textbf{send}\, y\, x^{\langle c,e\rangle}; P \sim_{\langle d,f\rangle} \textbf{send}\, y\, x^{\langle c,e\rangle}; Q}$$

$$\textsc{RcvChn}\ \frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \vDash [y/y_1]P \sim_{\langle d,f\sqcup e\rangle} [y/y_2]Q}{\Psi \vDash y_1 \leftarrow \textbf{recv}\, x^{\langle c,e\rangle}; P_{y_1} \sim_{\langle d,f\rangle} y_2 \leftarrow \textbf{recv}\, x^{\langle c,e\rangle}; Q_{y_2}}$$

$$\frac{\Psi \not\Vdash d \sqsubseteq f \qquad (\Psi \not\Vdash d \sqsubseteq e_0 \text{ or } \exists y{:}B\langle c',e'\rangle \in \Delta.\Psi \not\Vdash d \sqsubseteq e') \quad \Psi \vDash [x/x_1]P \sim_{\langle d,f\rangle} [x/x_2]Q}{\Psi \vDash (x_1^{\langle c,e\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle c_0,e_0\rangle; P_{x_1} \sim_{\langle d,f\rangle} (x_2^{\langle c,e\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle c_0,e_0\rangle; Q_{x_2}}\ \textsc{Sync-Spawn}$$

$$\frac{\Psi \not\Vdash d \sqsubseteq f}{\Psi \vDash x^{\langle c_1,e_1\rangle} \leftarrow y^{\langle c_2,e_2\rangle} \sim_{\langle d,f\rangle} x^{\langle c_1,e_1\rangle} \leftarrow y^{\langle c_2,e_2\rangle}}\ \textsc{Fwd}$$

$$\frac{\Psi \not\Vdash d \sqsubseteq f}{\Psi \vDash \textbf{close}\, x^{\langle c,e\rangle} \sim_{\langle d,f\rangle} \textbf{close}\, x^{\langle c,e\rangle}}\ \textsc{Close} \qquad \frac{\Psi \not\Vdash d \sqsubseteq f \quad \Psi \vDash P \sim_{\langle d,f\sqcup e\rangle} Q}{\Psi \vDash \textbf{wait}\, x^{\langle c,e\rangle}; P \sim_{\langle d,f\rangle} \textbf{wait}\, x^{\langle c,e\rangle}; Q}\ \textsc{Wait}$$

Figure 6 Synchronization pattern checking rules of SINTEGRITY.

5.2 Synchronization patterns

To check synchronization patterns, we use the judgment $\Psi \vDash P \sim_{\langle d,f\rangle} Q$, defined inductively in Fig. 6. The judgment states that process terms P and Q are *synchronized* in terms of their communication pattern, meaning that if P outputs along channel x, so must Q, and that if P inputs along channel x, so must Q, and vice versa. The check is *conditioned* on the running confidentiality d and running integrity f of the recipient after branching, and is pairwise called for all branches of a **case** statement. Let us assume that right after branching, the known secret of a process (its running confidentiality) is of level d. The goal of the synchronization pattern checks is to rule out any leakage of this secret of level d via regrading. Such leakage is only possible if the process (or any process that receives this secret from it) regrades to a lower or unrelated level than the secret d. However, if $d \sqsubseteq f$, we know that this can never happen. Therefore, if $\Psi \Vdash d \sqsubseteq f$, the judgment $\Psi \vDash P \sim_{\langle d,f\rangle} Q$ trivially holds. This case is handled by Rule $\textsc{Unsync}_3$ and is a base case of the inductive definition.

The interesting case is when $\Psi \not\Vdash d \sqsubseteq f$, meaning that the process can potentially regrade to a lower (or unrelated) level than d. In this case, the rules have to ensure that the secret d does not affect the ability of the process itself or the processes communicating with it to reach a regrading point. Furthermore, the secret d cannot affect the continuation of the process after regrading. In this case, the rules consider whether the next action in P and Q is a receive, send (except close), spawn, close, or forward:

- The receives are checked to be synchronized in P and Q by the rules RcvLab and RcvChan. The pattern check is invoked inductively on the continuation, with updated running integrity ($f \sqcup e$) to take into account the receive. The confidentiality of the learned secret d, however, remains constant under inductive invocations as it has to continue preventing the leak of the original secret. The receives have to be synchronized as long as $\Psi \not\Vdash d \sqsubseteq f$ holds, since different receives in P and Q might result in one branch reaching the regrading point and the other one not (related to non-reactiveness).
- Different sends in two branches of a process does not impact whether or not the process itself reaches a regrading point (sends are non-blocking). But, it may impact whether or not the other process, on the receiving side, reaches the regrading point based on the secret. If the carrier channel's min integrity e is high enough, the receiving process cannot regrade to a level lower (or unrelated) than d, and we do not need to synchronize the sends. The sends must only be synchronized if the carrier channel's min integrity e is not greater than or equal to the level d of the secret ($d \not\sqsubseteq e$). Rules Unsync$_1$ and Unsync$_2$ correspond to the former case where $d \sqsubseteq e$; for brevity, in these rules, we use process terms with any output prefix defined as $\uparrow_{x^{\langle c,e\rangle}} .P \triangleq x^{\langle c,e\rangle}.k; P \mid \textbf{send}\, y\, x^{\langle c,e\rangle}; P$. And rules SndLab and SndChan correspond to the latter where $d \not\sqsubseteq e$. In either case, the pattern check is invoked inductively on the continuation, with unchanged running integrity.
- Similar to the reasoning in the case of sends, if the running integrity of the spawned process and the min integrity of all its channels are high enough, there is no need to synchronize the spawns (Unsync-Spawn rules). Otherwise, the two branches must spawn the same processes with the same arguments (Sync-Spawn).
- Rules Close and Fwd are the other base cases of the inductive definition. They insist that the two branches P and Q can synchronize their termination behavior.

5.3 Configuration typing and asynchronous dynamics

The configuration typing judgment is of the form $\Psi_0; \Delta \Vdash \mathcal{C} :: \Delta'$, where Ψ_0 is the security lattice and $\mathcal{C}$ is a set of runtime processes $\textbf{proc}(y_\alpha\langle c, e\rangle, P@\langle c', e'\rangle)$ and messages $\textbf{msg}(M)$. Fig. 7 shows the configuration typing. The security premises in the **proc** and **msg** rules enforce the invariants (a)-(c) on the process term judgment before invoking process typing.

The dynamic rules in Fig. 8 take care of updating the running confidentiality and integrity of each process after a receive. For brevity, we write $\langle p\rangle$ to refer to a pair of confidentiality and integrity labels $\langle c, e\rangle$. Rule Spawn relies on the substitution mapping γ given by the programmer and its lifting $\hat{\gamma}$ to the process term level. It looks up the definition of process X in the signature and instantiates the security variables occurring in the process body using γ. The condition $\hat{\gamma}(\Psi') = \Psi_0$ ensures that all security variables are instantiated with a concrete security level. For brevity, we omit a channel generations as well as Σ, which is fixed.

5.4 Banking example

The following example implements a bank that authorizes transactions made by its customers and sends a copy to their bank accounts. In line with our security lattice, we assume that the bank has two customers, Alice and Bob. To authenticate themselves, a customer sends their

$$\frac{}{\Psi_0; x{:}A[\langle d,e\rangle] \Vdash \cdot :: (x{:}A[\langle d,e\rangle])}\ \textbf{emp} \qquad \frac{\Psi_0; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi_0; \Delta_0' \Vdash \mathcal{C}_1 :: x{:}A[\langle d,e\rangle]}{\Psi_0; \Delta_0, \Delta_0' \Vdash \mathcal{C}\,\mathcal{C}_1 :: \Delta, x{:}A[\langle d,e\rangle]}\ \textbf{comp}$$

$$\frac{\begin{array}{c}\Psi_0 \Vdash d_1 \sqsubseteq d \qquad \Psi_0 \Vdash e_1 \sqsubseteq e \\ \forall y{:}B[\langle d',e'\rangle] \in \Delta_0', \Delta\,(\Psi_0 \Vdash d' \sqsubseteq d) \qquad \forall y{:}B[\langle d',e'\rangle] \in \Delta_0', \Delta\,(\Psi_0 \Vdash e' \sqsubseteq e) \\ \Psi_0 \Vdash e_1 \sqsubseteq d_1 \qquad \Psi_0 \Vdash e \sqsubseteq d \qquad \forall y{:}B[\langle d',e'\rangle] \in \Delta_0', \Delta\,(\Psi_0 \Vdash e' \sqsubseteq d') \\ \Psi_0; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi_0; \Delta_0', \Delta \vdash P@\langle d_1, e_1\rangle :: (x{:}A[\langle d,e\rangle])\end{array}}{\Psi_0; \Delta_0, \Delta_0' \Vdash \mathcal{C}\,\textbf{proc}(x[\langle d,e\rangle], P@\langle d_1,e_1\rangle) :: (x{:}A[\langle d,e\rangle])}\ \textbf{proc}$$

$$\frac{\begin{array}{c}\forall y{:}B[\langle d',e'\rangle] \in \Delta_0', \Delta\,(\Psi_0 \Vdash d' \sqsubseteq d) \qquad \Psi_0 \Vdash e \sqsubseteq d \qquad \forall y{:}B[\langle d',e'\rangle] \in \Delta_0', \Delta\,(\Psi_0 \Vdash e' \sqsubseteq d') \\ \Psi_0; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi_0; \Delta_0', \Delta \vdash M@\langle d,e\rangle :: (x{:}A[\langle d,e\rangle])\end{array}}{\Psi_0; \Delta_0, \Delta_0' \Vdash \mathcal{C}, \textbf{msg}(M) :: (x{:}A[\langle d,e\rangle])}\ \textbf{msg}$$

Figure 7 Configuration typing rules of SINTEGRITY.

$$\begin{array}{ll}
\textsc{Spawn} & \textbf{proc}(y_\alpha\langle p\rangle, (x^{\langle p'\rangle} \leftarrow X[\gamma] \leftarrow \Delta)@\langle p_2\rangle; Q@\langle p_1\rangle) \\
& \hfill (\Psi'; \Delta' \vdash X = P@\langle \psi_0, \omega_0\rangle :: x : B'\langle \psi,\omega\rangle \in \Sigma) \\
& \mapsto \textbf{proc}(x_0\langle p'\rangle, ([x_0, \Delta/x, \Delta']\hat{\gamma}(P))@\langle p_2\rangle)\ \textbf{proc}(y_\alpha\langle p\rangle, ([x_0/x]Q)@\langle p_1\rangle) \\
& \hfill (\Psi_0 \Vdash \gamma : \Psi',\ x_0\ \mathit{fresh}) \\
1_{\mathsf{snd}} & \textbf{proc}(y_\alpha\langle p\rangle, (\textbf{close}\,y_\alpha)@\langle p_1\rangle) \mapsto \textbf{msg}(\textbf{close}\,y_\alpha) \\
1_{\mathsf{rcv}} & \textbf{msg}(\textbf{close}\,y_\alpha)\ \textbf{proc}(x_\beta\langle p'\rangle, (\textbf{wait}\,y_\alpha; Q)@\langle p_1\rangle) \mapsto \textbf{proc}(x_\beta\langle p'\rangle, Q@\langle p_1\rangle \sqcup \langle p\rangle) \\
\oplus_{\mathsf{snd}} & \textbf{proc}(y_\alpha\langle p\rangle, y_\alpha.k; P@\langle p_1\rangle) \mapsto \textbf{proc}(y_{\alpha+1}\langle p\rangle, ([y_{\alpha+1}/y_\alpha]P)@\langle p_1\rangle)\ \textbf{msg}(y_\alpha.k) \\
\oplus_{\mathsf{rcv}} & \textbf{msg}(y_\alpha.k)\ \textbf{proc}(u_\gamma\langle p'\rangle, \textbf{case}\,y_\alpha^{\langle p\rangle}((\ell \Rightarrow P_\ell)_{\ell\in L})@\langle p_1\rangle) \\
& \mapsto \textbf{proc}(u_\gamma\langle p'\rangle, ([y_{\alpha+1}/y_\alpha]P_k)@\langle p_1\rangle \sqcup \langle p\rangle) \\
\&_{\mathsf{snd}} & \textbf{proc}(y_\alpha\langle p\rangle, (x_\beta.k; P)@\langle p_1\rangle) \mapsto \textbf{msg}(x_\beta.k)\ \textbf{proc}(y_\alpha\langle p\rangle, ([x_{\beta+1}/x_\beta]P)@\langle p_1\rangle) \\
\&_{\mathsf{rcv}} & \textbf{proc}(y_\alpha\langle p\rangle, (\textbf{case}\,y_\alpha(\ell \Rightarrow P_\ell)_{\ell\in L})@\langle p_1\rangle)\ \textbf{msg}(y_\alpha.k) \mapsto \textbf{proc}(v_\delta\langle p\rangle, ([y_{\alpha+1}/y_\alpha]P_k)@\langle p\rangle) \\
\otimes_{\mathsf{snd}} & \textbf{proc}(y_\alpha\langle p\rangle, (\textbf{send}\,x_\beta\,y_\alpha; P)@\langle p_1\rangle) \mapsto \textbf{proc}(y_{\alpha+1}\langle p\rangle, ([y_{\alpha+1}/y_\alpha]P)@\langle p_1\rangle)\ \textbf{msg}(\textbf{send}\,x_\beta\,y_\alpha) \\
\otimes_{\mathsf{rcv}} & \textbf{msg}(\textbf{send}\,x_\beta\,y_\alpha)\ \textbf{proc}(u_\gamma\langle p'\rangle, (w \leftarrow \textbf{recv}\,y_\alpha^{\langle p\rangle}; P)@\langle p_1\rangle) \\
& \mapsto \textbf{proc}(u_\gamma\langle p'\rangle, ([x_\beta/w][y_{\alpha+1}/y_\alpha]P)@(\langle p_1\rangle \sqcup \langle p\rangle)) \\
\multimap_{\mathsf{snd}} & \textbf{proc}(y_\alpha\langle p\rangle, (\textbf{send}\,x_\beta\,u_\gamma; P)@\langle p_1\rangle) \mapsto \textbf{msg}(\textbf{send}\,x_\beta\,u_\gamma)\ \textbf{proc}(y_\alpha\langle p\rangle, ([u_{\gamma+1}/u_\gamma]P)@\langle p_1\rangle) \\
\multimap_{\mathsf{rcv}} & \textbf{proc}(y_\alpha\langle p\rangle, (w \leftarrow \textbf{recv}\,y_\alpha; P)@\langle p_1\rangle)\ \textbf{msg}(\textbf{send}\,x_\beta\,y_\alpha) \mapsto \textbf{proc}(v_\delta\langle p\rangle, ([x_\beta/w][y_{\alpha+1}/y_\alpha]P)@\langle p\rangle)
\end{array}$$

Figure 8 Asynchronous dynamics of SINTEGRITY.

token to the bank. The bank then verifies the token and, if the verification is successful, sends the message *succ* to the customer, otherwise the message *fail*. Moreover, if the verification is successful, the bank creates a transaction statement and sends it to another process that represents the account of the customer in the bank. Once done, the bank continues to serve the next customer by making a recursive call. We assume that the bank alternates between its two customers, Alice and Bob, by making a mutually recursive call from $\mathsf{Bank_A}$, which serves Alice, to $\mathsf{Bank_B}$, which serves Bob, and vice versa. At each recursive call, a bank process regrades its running confidentiality to interact with the next customer. The example showcases a characteristic feature of our type system: it accepts an implementation for a bank that interactively communicates with Alice and Bob without jeopardizing noninterference.

The following session types dictate the above protocol:

$$\begin{array}{ll}
\mathsf{customer} = \oplus\{\mathit{tok_{black}} : \&\{\mathit{succ} : \mathsf{customer}, \mathit{fail} : \mathsf{customer}\}, & \mathsf{account} = \mathsf{transfer} \multimap \mathsf{account} \\
\qquad\qquad\quad \mathit{tok_{white}} : \&\{\mathit{succ} : \mathsf{customer}, \mathit{fail} : \mathsf{customer}\}\} & \mathsf{transfer} = \oplus\{\mathit{transaction} : 1\}
\end{array}$$

Fig. 9 shows the process implementations $\mathsf{Bank_A}$, $\mathsf{Bank_B}$, $\mathsf{Customer_A}$, and $\mathsf{Statement_A}$. The latter two are the implementation of Alice's customer and statement process, resp. The implementation of $\mathsf{Customer_A}$ is as expected. $\mathsf{Statement_A}$ signals a single transfer by sending the label transaction and terminates. The implementation of corresponding processes for Bob,

$\Psi; y_1{:}\mathsf{customer}\langle\psi, \mathbf{guest}\rangle, y_2{:}\mathsf{customer}\langle\psi', \mathbf{guest}\rangle, w_1{:}\mathsf{account}\langle\psi, \psi\rangle, w_2{:}\mathsf{account}\langle\psi', \psi'\rangle$
$\vdash \mathsf{Bank_A} :: x{:}1\langle\mathbf{bank}, \mathbf{bank}\rangle$
$x \leftarrow \mathsf{Bank_A} \leftarrow y_1, y_2, w_1, w_2 =$
$\mathbf{case}\, y_1\, (tok_{white} \Rightarrow y_1.succ;\ (u \leftarrow \mathsf{Statement_A}[\gamma] \leftarrow \cdot); \mathbf{send}\, u\, w_1; (x' \leftarrow \mathsf{Bank_B}[\gamma] \leftarrow y_1, y_2, w_1, w_2);$
$\quad x \leftarrow x'$
$\quad \mid tok_{black} \Rightarrow y_1.fail; (x' \leftarrow \mathsf{Bank_B}[\gamma] \leftarrow y_1, y_2, w_1, w_2);\ x \leftarrow x'\,)@\langle\mathbf{guest}, \mathbf{guest}\rangle$

$\Psi; y_1{:}\mathsf{customer}\langle\psi, \mathbf{guest}\rangle, y_2{:}\mathsf{customer}\langle\psi', \mathbf{guest}\rangle, w_1{:}\mathsf{account}\langle\psi, \psi\rangle, w_2{:}\mathsf{account}\langle\psi', \psi'\rangle$
$\vdash \mathsf{Bank_B} :: x{:}1\langle\mathbf{bank}, \mathbf{bank}\rangle$
$x \leftarrow \mathsf{Bank_B} \leftarrow y_1, y_2, w_1, w_2 =$
$\mathbf{case}\, y_2\, (tok_{black} \Rightarrow y_2.succ;\ (u \leftarrow {}_{\mathsf{B}}[\gamma] \leftarrow \cdot); \mathbf{send}\, u\, w_2; (x' \leftarrow \mathsf{Bank_A}[\gamma] \leftarrow y_1, y_2, w_1, w_2);\ x \leftarrow x'$
$\quad \mid tok_{white} \Rightarrow y_2.fail; (x' \leftarrow \mathsf{Bank_A}[\gamma] \leftarrow y_1, y_2, w_1, w_2);\ x \leftarrow x'\,)@\langle\mathbf{guest}, \mathbf{guest}\rangle$

$\Psi; \cdot \vdash \mathsf{Customer_A} :: y{:}\mathsf{customer}\langle\psi, \mathbf{guest}\rangle$
$y \leftarrow \mathsf{Customer_A} \leftarrow \cdot =$
$y.tok_{white}; \mathbf{case}\, y\, (succ \Rightarrow (y' \leftarrow \mathsf{Customer_A}[\gamma] \leftarrow \cdot);\ y \leftarrow y'$
$\quad \mid fail \Rightarrow (y' \leftarrow \mathsf{Customer_A}[\gamma] \leftarrow \cdot);\ y \leftarrow y')\ \ @\langle\psi, \mathbf{guest}\rangle$

$\Psi; \cdot \vdash \mathsf{Statement_A} :: u{:}\mathsf{transfer}\langle\psi, \psi\rangle$
$u \leftarrow \mathsf{Statement_A} \leftarrow \cdot =\ \ u.transaction; \mathbf{close}\, u\ \ @\langle\psi, \psi\rangle$

Figure 9 Security-polymorphic process definitions.

i.e., $\mathsf{Customer_B}$, and $\mathsf{Statement_B}$, would be similar. The example is typed using the security theory Ψ, consisting of the concrete security lattice Ψ_0, the security variables ψ and ψ', and the set of constraints $\{\mathbf{guest} \sqsubseteq \psi \sqsubseteq \mathbf{bank}, \mathbf{guest} \sqsubseteq \psi' \sqsubseteq \mathbf{bank}\}$. (See TR for the formal definition of a security theory.) To execute this program using the dynamics in Fig. 8, we provide the order-preserving substitution $\Psi_0 \Vdash \gamma' :: \Psi$, defined as $\gamma' := \{\psi \mapsto \mathbf{alice}, \psi' \mapsto \mathbf{bob}\}$.

Let us examine the pattern checks $\sim_{\langle\psi,\mathbf{guest}\rangle}$ invoked by $\mathbf{case}\, y_1(\dots)$ in $\mathbf{Bank}_A$, relating the branches corresponding to black and white tokens. The sends along y_1 match in both branches, as demanded by SndLab (since $\Psi \nVdash \psi \sqsubseteq \mathbf{guest}$), even though the sent labels are not the same. The unsynchronized spawn and send along w_1 is verified by Unsync-Spawn$_1$ and Unsync$_1$, resp., since $\Psi \Vdash \psi \sqsubseteq \psi$. The matching tail calls are verified with Sync-Spawn.

6 Progress-sensitive noninterference

This section presents our main result, PSNI, which we prove using a logical relation.

6.1 Attacker model

The attacker model assumes a configuration $\mathcal{D}$ with prior annotation of its free channels with security levels, the attacker's confidentiality level ξ, and a nondeterministic scheduler. The attacker knows the source code of $\mathcal{D}$, can only observe the messages sent along the free channels of $\mathcal{D}$ with confidentiality level $c \sqsubseteq \xi$, and cannot measure the passing of time.

6.2 Noninterference via an integrity logical relation

Noninterference amounts to a process equivalence up to the confidentiality level ξ of an observer. In a message-passing system, it boils down to an equivalence of a configuration with interacting processes. This section focuses on noninterference for tree-shaped configurations. The definition can be extended to forests by enforcing pairwise relation between their trees.

An open configuration $\Psi_0; \Delta \Vdash \mathcal{D} :: x_\alpha{:}A\langle c, e\rangle$ has the free channels Δ and x_α to communicate with its external environment; it sends outgoing messages to and receives incoming messages from the environment along these free channels. Two observationally equivalent

$$(\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^{m+1} \quad \text{iff} \quad (\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash K) \text{ and} \forall\, \Upsilon_1,\, \Theta_1,\, \mathcal{D}_1'.\text{ if } \mathcal{D}_1 \mapsto^{*\Upsilon_1;\Theta_1} \mathcal{D}_1' \text{ then } \exists\, \Upsilon_2\, \mathcal{D}_2'.\text{ such that } \mathcal{D}_2 \mapsto^{*\Upsilon_2} \mathcal{D}_2' \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and } \forall\, y_\alpha \in \mathbf{Out}(\Delta \Vdash K).\,\mathbf{if}\; y_\alpha \in \Upsilon_1.\,\mathbf{then}\; (\mathcal{D}_1'; \mathcal{D}_2') \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^{m+1}_{\cdot;y_\alpha} \text{ and } \forall\, y_\alpha \in \mathbf{In}(\Delta \Vdash K).\mathbf{if}\; y_\alpha \in \Theta_1.\,\mathbf{then}\; (\mathcal{D}_1'; \mathcal{D}_2') \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^{m+1}_{y_\alpha;\cdot}$$

$$(\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^{0} \quad \text{iff} \quad (\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash K)$$

Figure 10 Term interpretation of logical relation.

(l_1) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta, y_\alpha{:}1\langle c, e\rangle \Vdash K]\!]^{m+1}_{y^\alpha;\cdot}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta, y_\alpha{:}1\langle c, e\rangle \Vdash K)$ then
$(\mathbf{msg}(\mathbf{close}\, y_\alpha^{\langle c,e\rangle})\mathcal{D}_1; \mathbf{msg}(\mathbf{close}\, y_\alpha^{\langle c,e\rangle})\mathcal{D}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^{m}$

(l_2) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta, y_\alpha : \oplus\{\ell{:}A_\ell\}_{\ell\in I}\langle c, e\rangle \Vdash K]\!]^{m+1}_{y^\alpha;\cdot}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta, y_\alpha{:} \oplus \{\ell{:}A_\ell\}_{\ell\in I}\langle c, e\rangle \Vdash K)$ and $\forall k_1, k_2 \in I.$if$(c \sqsubseteq \xi \to k_1 = k_2)$ then
$(\mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_1)\mathcal{D}_1; \mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_2)\mathcal{D}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta, y_{\alpha+1}{:}A_{k_1}\langle c, e\rangle \Vdash K]\!]^{m}$

(l_3) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta, y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}\langle c, e\rangle \Vdash K]\!]^{m+1}_{\cdot;y^\alpha}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta, y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}\langle c, e\rangle \Vdash K)$ and $\exists k_1, k_2 \in I.(c \sqsubseteq \xi \to k_1 = k_2)$ and
$\mathcal{D}_1 = \mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_1)\mathcal{D}_1'$ and $\mathcal{D}_2 = \mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_2)\mathcal{D}_2'$ and
$(\mathcal{D}_1'; \mathcal{D}_2') \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta, y_{\alpha+1}{:}A_{k_1}\langle c, e\rangle \Vdash K]\!]^{m}$

(l_4) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta, y_\alpha{:}A \otimes B\langle c, e\rangle \Vdash K]\!]^{m+1}_{y^\alpha;\cdot}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta, y_\alpha{:}A \otimes B\langle c, e\rangle \Vdash K)$ and$\forall x_\beta \notin dom(\Delta, y_\alpha{:}A \otimes B\langle c, e\rangle, K)$.
$(\mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}, y_\alpha^{\langle c,e\rangle})\mathcal{D}_1; \mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}, y_\alpha^{\langle c,e\rangle})\mathcal{D}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta, x_\beta{:}A\langle c, e\rangle, y_{\alpha+1}{:}B\langle c, e\rangle \Vdash K]\!]^{m}$

(l_5) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta', \Delta'', y_\alpha{:}A \multimap B\langle c, e\rangle \Vdash K]\!]^{m+1}_{\cdot;y^\alpha}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta', \Delta'', y_\alpha{:}A \multimap B\langle c, e\rangle \Vdash K)$ and
$\mathcal{D}_1 = \mathcal{T}_1\mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}\; y_\alpha^{\langle c,e\rangle})\, \mathcal{D}_1''$ and for $\mathcal{T}_1 \in \mathsf{Tree}_{\Psi_0}(\Delta' \Vdash x_\beta{:}A\langle c, e\rangle)$
$\mathcal{D}_2 = \mathcal{T}_2\mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}\; y_\alpha^{\langle c,e\rangle})\, \mathcal{D}_2''$ and for $\mathcal{T}_2 \in \mathsf{Tree}_{\Psi_0}(\Delta' \Vdash x_\beta{:}A\langle c, e\rangle)$ and
$(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta' \Vdash x_\beta{:}A\langle c, e\rangle]\!]^{m}$ and
$(\mathcal{D}_1''; \mathcal{D}_2'') \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta'', y_{\alpha+1}{:}B\langle c, e\rangle \Vdash K]\!]^{m}$

Figure 11 Value interpretation of logical relation for *left* communications.

configurations may only differ in outgoing messages of confidentiality level $c_o \not\sqsubseteq \xi$, assuming that the incoming messages of confidentiality level $c_i \sqsubseteq \xi$ are the same. We introduce a *logical relation* that captures this idea and accounts for integrity and regrading policies.

The logical relation relates two open configurations $\mathcal{D}_1$ and $\mathcal{D}_2$ – the two runs of the program under consideration – and asserts that $\mathcal{D}_1$ and $\mathcal{D}_2$ send related messages to the environment, if they receive related messages from the environment. The term interpretation of the logical relation, defined in Fig. 10, allows the first configuration $\mathcal{D}_1$ to step internally until the configuration is ready to send or receive a message across at least one external channel. Then, it requires the second configuration $\mathcal{D}_2$ to step internally so that the resulting configurations are in the value interpretation of the logical relation, defined in Fig. 11 and Fig. 12. We call the external channels, e.g., $\Delta \Vdash K$ in Fig. 10, the interface of $\mathcal{D}_1$ and $\mathcal{D}_2$. The metavariable K stands for either $x_\alpha{:}A\langle c, e\rangle$ or simply $_{:}1\langle\top,\top\rangle$ which refers to an arbitrary unobservable channel.

The idea is to build an interface consisting of those external channels of the configurations that may impact the attacker's observations. As such, not only do we need to include the observable channels, i.e., with confidentiality level $c \sqsubseteq \xi$, in the interface, but also those with higher integrity than the observer, i.e., with integrity level $e \sqsubseteq \xi$. After all, if a channel's integrity is high enough (and thus its level is low), the messages along it may affect an observable outcome via synchronization patterns. We call such an interface *integrity interface* since low-confidentiality channels are all high-integrity by typing.

(r_1) $(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\cdot \Vdash y_\alpha{:}1\langle c,e\rangle]\!]^{m+1}_{\cdot;y^\alpha}$
iff $(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\cdot \Vdash y_\alpha)$ and $\mathcal{D}_1 = \mathbf{msg}(\mathbf{close}\, y_\alpha^{\langle c,e\rangle})$ and $\mathcal{D}_2 = \mathbf{msg}(\mathbf{close}\, y_\alpha^{\langle c,e\rangle})$

(r_2) $(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![(\Delta \Vdash y_\alpha{:}\oplus\{\ell{:}A_\ell\}_{\ell\in I}\langle c,e\rangle)]\!]^{m+1}_{\cdot;y^\alpha}$
iff $(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash y_\alpha : \oplus\{\ell{:}A_\ell\}_{\ell\in I}\langle c,e\rangle)$ and$\exists k_1,k_2 \in I.\ (c \sqsubseteq \xi \rightarrow k_1 = k_2)$
$\mathcal{D}_1 = \mathcal{D}_1'\mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_1)$ and $\mathcal{D}_2 = \mathcal{D}_2'\mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_2)$
and $(\mathcal{D}_1';\mathcal{D}_2') \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash y_{\alpha+1}{:}A_{k_1}\langle c,e\rangle]\!]^m$

(r_3) $(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta \Vdash y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}\langle c,e\rangle]\!]^{m+1}_{y^\alpha;\cdot}$
iff $(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}\langle c,e\rangle)$ then$\forall k_1,k_2 \in I.$if $(c \sqsubseteq \xi \rightarrow k_1 = k_2)$ then
$(\mathcal{D}_1\mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_1), \mathcal{D}_2\mathbf{msg}(y_\alpha^{\langle c,e\rangle}.k_2)) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash y_{\alpha+1}{:}A_{k_1}\langle c,e\rangle]\!]^m$

(r_4) $(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta',\Delta'' \Vdash y_\alpha{:}A\otimes B\langle c,e\rangle]\!]^{m+1}_{\cdot;y^\alpha}$
iff$(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta',\Delta'' \Vdash y_\alpha{:}A\otimes B\langle c,e\rangle)$ and $\exists x_\beta.$
$\mathcal{D}_1 = \mathcal{D}_1'\mathcal{T}_1\mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})$ for $\mathcal{T}_1 \in \mathsf{Tree}_{\Psi_0}(\Delta'' \Vdash x_\beta{:}A\langle c,e\rangle)$ and
$\mathcal{D}_2 = \mathcal{D}_2'\mathcal{T}_2\mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})$ for $\mathcal{T}_2 \in \mathsf{Tree}_{\Psi_0}(\Delta'' \Vdash x_\beta{:}A\langle c,e\rangle)$ and
$(\mathcal{T}_1;\mathcal{T}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta'' \Vdash x_\beta{:}A\langle c,e\rangle]\!]^m$ and
$(\mathcal{D}_1';\mathcal{D}_2') \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta' \Vdash y_{\alpha+1}{:}B\langle c,e\rangle]\!]^m$

(r_5) $(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{V}^{\xi}_{\Psi_0}[\![\Delta \Vdash y_\alpha{:}A \multimap B\langle c,e\rangle]\!]^{m+1}_{y^\alpha;\cdot}$
iff $(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash y_\alpha{:}A \multimap B\langle c,e\rangle)$ and $\forall x_\beta \notin dom(\Delta, y_\alpha{:}A \multimap B\langle c,e\rangle)$.
$(\mathcal{D}_1\mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle}); \mathcal{D}_2\mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta, x_\beta{:}A\langle c,e\rangle \Vdash y_{\alpha+1}{:}B\langle c,e\rangle]\!]^m$

Figure 12 Value interpretation of logical relation for *right* communications.

$(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1\langle c_1,e_1\rangle) \equiv^{\Psi_0}_{\xi} (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2\langle c_2,e_2\rangle)$ iff
$\mathcal{D}_1 \in \mathsf{Tree}(\Delta_1 \Vdash x_\alpha{:}A_1\langle c_1,e_1\rangle)$ and $\mathcal{D}_2 \in \mathsf{Tree}(\Delta_2 \Vdash y_\beta{:}A_2\langle c_2,e_2\rangle)$ and $\Delta_1 \Downarrow^{\mathsf{ig}} \xi = \Delta_2 \Downarrow^{\mathsf{ig}} \xi = \Delta$ and
$x_\alpha{:}A_1\langle c_1,e_1\rangle \Downarrow^{\mathsf{ig}} \xi = y_\beta{:}A_2\langle c_2,e_2\rangle \Downarrow^{\mathsf{ig}} \xi = K$ and $\forall \mathcal{B}_1 \in \mathbf{L\text{-}IProvider}^{\xi}(\Delta_1).\, \forall \mathcal{B}_2 \in \mathbf{L\text{-}IProvider}^{\xi}(\Delta_2).$
$\forall \mathcal{T}_1 \in \mathbf{L\text{-}IClient}^{\xi}(x_\alpha{:}A_1\langle c_1,e_1\rangle).\forall \mathcal{T}_2 \in \mathbf{L\text{-}IClient}^{\xi}(y_\beta{:}A_2\langle c_2,e_2\rangle).$
$\forall m.\, (\mathcal{B}_1\mathcal{D}_1\mathcal{T}_1, \mathcal{B}_2\mathcal{D}_2\mathcal{T}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^m$, and $\forall m.\, (\mathcal{B}_2\mathcal{D}_2\mathcal{T}_2, \mathcal{B}_1\mathcal{D}_1\mathcal{T}_1) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta \Vdash K]\!]^m$.

$\cdot \in \mathbf{L\text{-}IProvider}^{\xi}(\cdot)$
$\mathcal{B} \in \mathbf{L\text{-}IProvider}^{\xi}(\Delta, x_\alpha{:}A\langle c,e\rangle)$ iff
$e \not\sqsubseteq \xi$ and $\mathcal{B} = \mathcal{B}'\mathcal{T}$ and $\mathcal{B}' \in \mathbf{L\text{-}IProvider}^{\xi}(\Delta)$ and $\mathcal{T} \in \mathsf{Tree}(\cdot \Vdash x_\alpha{:}A\langle c,e\rangle)$, **or**
$e \sqsubseteq \xi$ and $\mathcal{B} \in \mathbf{L\text{-}IProvider}^{\xi}(\Delta)$

$\mathcal{T} \in \mathbf{L\text{-}IClient}^{\xi}(x_\alpha{:}A\langle c,e\rangle)$ iff
$e \not\sqsubseteq \xi$ and $\mathcal{T} \in \mathsf{Tree}(x_\alpha{:}A\langle c,e\rangle \Vdash _ : 1\langle\top,\top\rangle)$, **or** $e \sqsubseteq \xi$ and $\mathcal{T} = \cdot$

Figure 13 Logical equivalence.

To build an *integrity interface* for $\mathcal{D}_1$ and $\mathcal{D}_2$, we close off their external low-integrity ($e \not\sqsubseteq \xi$) channels on the left by composing the channels with any well-typed provider and on the right with any well-typed client. We may use different low-integrity clients and providers to compose with each program run. These clients/providers can send different and unsynchronized messages along their high-confidentiality and low-integrity connections to $\mathcal{D}_1$ and $\mathcal{D}_2$. The term interpretation is designed to ensure that well-typed configurations do not leak these different messages to the attacker. Fig. 13 defines an equivalence relation between two configurations based on this idea: it composes them with low-integrity providers/clients and calls the term interpretation symmetrically on the compositions. In the definition, we use the projection function to build the integrity interface, e.g., $\Delta \Downarrow^{\mathsf{ig}} \xi$ projects out the channels $y_\beta{:}A\langle c,e\rangle \in \Delta$ with $\xi \not\sqsubseteq e$. The predicate $\mathcal{D}_1 \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash K)$ indicates that the configuration $\mathcal{D}_1$ is well-typed. In the term and value interpretations, we generalize this predicate to the binary case, $(\mathcal{D}_1;\mathcal{D}_2) \in \mathsf{Tree}_{\Psi_0}(\Delta \Vdash K)$ indicating that both $\mathcal{D}_1$ and $\mathcal{D}_2$ are of the same type.

The term interpretation allows stepping configuration $\mathcal{D}_1 \mapsto^{*\Upsilon_1;\Theta_1} \mathcal{D}_1'$ by iterated application of the rewriting rules defined in Fig. 8. The star expresses that zero to multiple internal steps can be taken. The superscripts $\Upsilon_1;\Theta_1$ denote two sets of channels occurring in the

interface $\Delta \Vdash K$. The set Θ_1 collects the *incoming* channels, i.e., channels that a process in $\mathcal{D}_1$ is ready to receive from, and the set Υ_1 collects the *outgoing* channels, i.e., channels with a message in $\mathcal{D}_1$ ready to be sent. Assuming that $\mathcal{D}_1$ steps to $\mathcal{D}_1'$, generating the outgoing channels Υ_1, $\mathcal{D}_2$ must be stepped $\mathcal{D}_2 \mapsto^{*\Upsilon_2} \mathcal{D}_2'$ to produce at least the same set of outgoing channels, i.e., the set Υ_2 such that $\Upsilon_1 \subseteq \Upsilon_2$. The term interpretation then calls the value interpretation on the resulting configurations $\mathcal{D}_1', \mathcal{D}_2'$ *for every* channel that has a message ready for transmission in $\mathcal{D}_1'$, and thus $\mathcal{D}_2'$, and *for every* channel that has a process waiting for a message in $\mathcal{D}_1'$. Insisting on Υ_2 being a superset of Υ_1 ensures progress-sensitive noninterference without timing attacks: if a configuration produces observable messages along a set of channels, the other configuration has to be able to produce the equivalent set of messages with zero or some internal steps. The term interpretation uses focus channels as a subscript to the value interpretation to support simultaneous communications – when there are multiple messages ready to be sent or received along channels in the interface. The subscript $\cdot; y_\alpha$ indicates that $y_\alpha \in \Upsilon_1$ and $y_\alpha; \cdot$ that $y_\alpha \in \Theta_1$.

The value interpretation accounts for every message sent from or received by $\mathcal{D}_1$ and $\mathcal{D}_2$, amounting to two cases per connective: one for a message exchanged along a channel in K and one for a message exchanged along a channel in Δ. We refer to the former as communications to the *right* (Fig. 12) and the latter as communications to the *left* (Fig. 11). The value interpretation generally establishes the following pattern: it asserts relatedness of outgoing messages, but assumes relatedness of incoming messages. For example, & on the left (l_3 in Fig. 11) *asserts* the sending of related messages and pushes the messages into the environment, yielding $\mathcal{D}_1'$, $\mathcal{D}_2'$. Now, $\mathcal{D}_1'$, $\mathcal{D}_2'$, can each step internally, e.g., to consume the incoming messages, requiring them to be in the term interpretation. On the other hand, & on the right (r_3 in Fig. 12) *assumes* receipt of related messages and pushes the messages into the configurations $\mathcal{D}_1$ and $\mathcal{D}_2$. *Relatedness* for messages is determined by how they can impact the attacker's observations. If their carrier channel is observable to the attacker, i.e., has confidentiality level $c \sqsubseteq \xi$, then related messages must have the same labels. But if the channel only affects the attacker's observations via synchronization patterns, related messages may have different labels. The clause $c \sqsubseteq \xi \to k_1 = k_2$ in the value interpretation conveys this, enforcing equality of the communicated labels only if the channel is observable.

Relatedness for higher-order types ($\otimes$ and $\multimap$) is a bit more subtle. In particular, it requires future observations along the exchanged channels to be related. For example, l_5 in Fig. 11 for $\multimap$-left asserts existence of a message $\mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})$ and of subtrees $\mathcal{T}_1$ and $\mathcal{T}_2$ in $\mathcal{D}_1$ and $\mathcal{D}_2$. The clause comprises two invocations of the term relation, $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta' \Vdash x_\beta{:}A\langle c,e\rangle]\!]^m$, asserting that future observations to be made along the sent channel x_β are related, and $(\mathcal{D}_1''; \mathcal{D}_2'') \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta'', y_{\alpha+1}{:}B\langle c,e\rangle \Vdash K]\!]^m$, asserting that the continuations $\mathcal{D}_1''$ and $\mathcal{D}_2''$ are related. Conversely, r_5 in Fig. 12 for $\multimap$-right assumes receipt of a message $\mathbf{msg}(\mathbf{send} x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})$ and invokes the term relation $(\mathcal{D}_1 \mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle}); \mathcal{D}_2 \mathbf{msg}(\mathbf{send}\, x_\beta^{\langle c,e\rangle}\, y_\alpha^{\langle c,e\rangle})) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta, x_\beta{:}A\langle c,e\rangle \Vdash y_{\alpha+1}{:}B\langle c,e\rangle]\!]^m$.

As we support general recursive types, we need an index to stratify our logical relation [3,5]. We tie our index to the number of *observations* that can be made along the interface $\Delta \Vdash K$, as suggested in [7]. We thus bound the value and term interpretation of our logical relation by the number of observations m, for $m \geq 0$, and attach them as superscripts to the relation's interface $\Delta \Vdash K$. The base case of the term interpretation, i.e., $m = 0$, is a trivial relation.

The fundamental theorem states that any well-typed SINTEGRITY configuration is equivalent to itself up to the level of an arbitrary observer.

▶ **Theorem 1** (Fundamental theorem). *For all security levels ξ, and a well-typed configuration $\Psi_0; \Delta \Vdash \mathcal{D} :: u_\alpha{:}T\langle c,e\rangle$ we have $(\Delta \Vdash \mathcal{D} :: u_\alpha{:}T\langle c,e\rangle) \equiv^{\Psi_0}_{\xi} (\Delta \Vdash \mathcal{D} :: u_\alpha{:}T\langle c,e\rangle)$.*

Proof. We present a proof sketch; see the TR for details. By the definition of logical equivalence (Fig. 13), we first need to close off the external low integrity ($e \not\sqsubseteq \xi$) channels in Δ and $u_\alpha{:}T\langle c, e\rangle$ by composing $\mathcal{D}$ with arbitrary well-typed providers and clients, resp. We use low integrity clients, $\mathcal{T}_1$ and $\mathcal{T}_2$, and low-integrity providers, $\mathcal{B}_1$ and $\mathcal{B}_2$, to compose with each run, resulting in two configurations, $\mathcal{D}_1 = \mathcal{B}_1\mathcal{D}\mathcal{T}_1$ and $\mathcal{D}_2 = \mathcal{B}_2\mathcal{D}\mathcal{T}_2$. Configurations $\mathcal{D}_1$ and $\mathcal{D}_2$ are both well-typed for the integrity interface: $\Psi_0; \Delta' \Vdash \mathcal{D}_i :: K$ where $\Delta' = \Delta\Downarrow^{\mathsf{ig}}\xi$ and $K = u_\alpha{:}T\langle c, e\rangle\Downarrow^{\mathsf{ig}}\xi$. By the definition in Fig. 13, it is enough to show that $\mathcal{D}_1$ and $\mathcal{D}_2$ are in the term interpretation with the integrity interface, i.e., $\forall m. (\mathcal{D}_1, \mathcal{D}_2) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta' \Vdash K]\!]^m$ and $\forall m. (\mathcal{D}_2, \mathcal{D}_1) \in \mathcal{E}^{\xi}_{\Psi_0}[\![\Delta' \Vdash K]\!]^m$. We prove the former by induction on m; the proof of the latter is symmetric. Specifically, we prove a more general theorem (Thm. 6.1 in TR) for any $\mathcal{D}_1$ and $\mathcal{D}_2$ with the same observable outcome, using the notion of relevant nodes (Def. 4.2 in TR). By the definition of the term interpretation in Fig. 10, the base case ($m = 0$) is straightforward. For the inductive case, following the first row of Fig. 10, we assume arbitrary Υ_1, Θ_1, and $\mathcal{D}_1'$ such that $\mathcal{D}_1 \mapsto^{*\Upsilon_1;\Theta_1} \mathcal{D}_1'$. We apply a lemma (Lem. 4.5 in TR) stating that $\mathcal{D}_2$ can simulate the internal steps taken by $\mathcal{D}_1$, producing at least the same set of outgoing channels Υ_2, i.e., $\mathcal{D}_2 \mapsto^{*\Upsilon_2} \mathcal{D}_2'$, such that $\mathcal{D}_1'$ and $\mathcal{D}_2'$ continue to have the same observable outcomes. Finally, for every channel x_α in Υ_1 and Θ_1, we case analyze on the type of x_α, showing that $\mathcal{D}_1'$ and $\mathcal{D}_2'$ are in the value interpretation, with x_α being the focus channel. To do so, we use the induction hypothesis to establish that after the corresponding communication with the environment, the continuations of $\mathcal{D}_1'$ and $\mathcal{D}_2'$ are related by the term interpretation for a smaller index. ◀

6.3 Adequacy

Next, we prove an adequacy theorem showing that two logically equivalent configurations are *bisimilar* up to observations of confidentiality ξ.

For adequacy, we are interested in *a confidentiality interface*, i.e., channels with observable max confidentiality $c \sqsubseteq \xi$; after all, our goal is to prove that the configurations are equivalent up to the confidentiality of an observer. Because the integrity interface of our logical relation is a *superset* of the confidentiality interface, we need to close off those channels in the integrity interface that are of high-confidentiality ($c \not\sqsubseteq \xi$). Note that these high-confidentiality channels are of high-integrity ($e \sqsubseteq \xi$). To close off these channels, we compose the open configurations with high-confidentiality clients and providers, possibly different ones for each program run. These high-integrity clients and providers are connected to the open configurations via high-integrity channels and, as a result, may affect the observable outcome of the two runs via synchronization patterns. We therefore require them to be logically equivalent.

Based on this idea, Fig. 14 defines the *bisimulation up to* confidentiality ξ denoted as $\approx^{\xi}_a$, for two well-typed configurations: it first composes the two configurations with high-confidentiality providers (**Hrel-IProvider**) and clients (**H-CClient**), while insisting that the high-integrity parts of the providers and clients are logically equivalent (using the relations **Hrel-IProvider** and **Hrel-IClient**). Then it invokes an asynchronous bisimulation $\approx_a$ on the compositions. The definition uses a projection function $\Downarrow^{\mathsf{cf}}\xi$ to build the confidentiality interface, e.g., $\Delta\Downarrow^{\mathsf{cf}}\xi$ projects out the channels in Δ with confidentiality $c \not\sqsubseteq \xi$.

The *asynchronous bisimulation* $\approx_a$ invoked by the definition in Fig. 14 uses a labeled transition system (LTS) following the standard definition of asynchronous bisimulation [38]. The relation $\mathcal{D}_1 \approx_a \mathcal{D}_2$ states that every internal step or external action of $\mathcal{D}_1$ can be (weakly) simulated by $\mathcal{D}_2$ and vice-versa. For example, when $\mathcal{D}_1$ takes an action by sending output q via an external channel x_α, i.e., $\mathcal{D}_1 \xrightarrow{\overline{x_\alpha}\, q} \mathcal{D}_1'$, the bisimulation ensures that for some $\mathcal{D}_2'$, we have $\mathcal{D}_2 \xRightarrow{\overline{x_\alpha}\, q} \mathcal{D}_2'$ and $\mathcal{D}_1' \approx_a \mathcal{D}_2'$. Here, $\xRightarrow{\overline{x_\alpha}\, q}$ stands for taking zero or more internal steps before outputting q along x_α. The full definition of bisimulation is in the TR.

$\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1\langle c_1, e_1\rangle \approx_a^\xi \Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2\langle c_2, e_2\rangle$ iff
$\mathcal{D}_1 \in \mathsf{Tree}(\Delta_1 \Vdash x_\alpha{:}A_1\langle c_1, e_1\rangle)$ and $\mathcal{D}_2 \in \mathsf{Tree}(\Delta_2 \Vdash y_\beta{:}A_2\langle c_2, e_2\rangle)$ and
$\Delta = \Delta_1 \Downarrow^{\mathsf{cf}} \xi = \Delta_2 \Downarrow^{\mathsf{cf}} \xi$ and $K = y_\beta{:}A_2\langle c_2, e_2\rangle \Downarrow^{\mathsf{cf}} \xi = x_\alpha{:}A_1\langle c_1, e_1\rangle \Downarrow^{\mathsf{cf}} \xi$ and
$\Delta' = \Delta_1 \Downarrow^{\mathsf{ig}} \xi = \Delta_2 \Downarrow^{\mathsf{ig}} \xi$ and $K' = y_\beta{:}A_2\langle c_2, e_2\rangle \Downarrow^{\mathsf{ig}} \xi = x_\alpha{:}A_1\langle c_1, e_1\rangle \Downarrow^{\mathsf{ig}} \xi$ and
$\forall \mathcal{B}_1 \in \textbf{H-CProvider}^\xi(\Delta_1), \mathcal{B}_2 \in \textbf{H-CProvider}^\xi(\Delta_2)$.
$\forall \mathcal{T}_1 \in \textbf{H-CClient}^\xi(x_\alpha{:}A_1\langle c_1, e_1\rangle), \mathcal{T}_2 \in \textbf{H-CClient}^\xi(y_\beta{:}A_2\langle c_2, e_2\rangle)$.
if $(\mathcal{B}_1, \mathcal{B}_2) \in \textbf{Hrel-IProvider}^\xi(\Delta' \backslash \Delta)$ and $(\mathcal{T}_1, \mathcal{T}_2) \in \textbf{Hrel-IClient}^\xi(K' \backslash K)$ then $\mathcal{B}_1\mathcal{D}_1\mathcal{T}_1 \approx_a \mathcal{B}_2\mathcal{D}_2\mathcal{T}_2$.

$\cdot \;\in \textbf{H-CProvider}^\xi(\cdot)$
$\mathcal{B} \in \textbf{H-CProvider}^\xi(\Delta, x_\alpha{:}A\langle c, e\rangle)$ iff
  $c \not\sqsubseteq \xi$ and $\mathcal{B} = \mathcal{B}'\mathcal{T}$ and $\mathcal{B}' \in \textbf{H-CProvider}^\xi(\Delta)$ and $\mathcal{T} \in \mathsf{Tree}(\cdot \Vdash x_\alpha{:}A\langle c, e\rangle)$, **or**
  $c \sqsubseteq \xi$ and $\mathcal{B} \in \textbf{H-CProvider}^\xi(\Delta)$

$\mathcal{T} \in \textbf{H-CClient}^\xi(x_\alpha{:}A\langle c, e\rangle)$ iff
  $c \not\sqsubseteq \xi$ and $\mathcal{T} \in \mathsf{Tree}(x_\alpha{:}A\langle c, e\rangle \Vdash _ : 1\langle\top, \top\rangle)$, **or**
  $c \sqsubseteq \xi$ and $\mathcal{T} = \cdot$

$(\cdot, \cdot) \;\in \textbf{Hrel-IProvider}^\xi(\cdot)$
$(\mathcal{B}_1, \mathcal{B}_2) \in \textbf{Hrel-IProvider}^\xi(\Delta, x_\alpha{:}A\langle c, e\rangle)$ iff
  $e \not\sqsubseteq \xi$ and $\mathcal{B}_i = \mathcal{B}'_i\mathcal{T}_i$ and $(\mathcal{B}'_1, \mathcal{B}'_2) \in \textbf{Hrel-IProvider}^\xi(\Delta)$, **or**
  $e \sqsubseteq \xi$ and $\mathcal{B}_i = \mathcal{B}'_i\mathcal{T}_i$ and $(\mathcal{B}'_1, \mathcal{B}'_2) \in \textbf{Hrel-IProvider}^\xi(\Delta)$ and $\cdot \Vdash \mathcal{T}_1 \equiv_\xi^{\Psi_0} \mathcal{T}_2 :: x_\alpha{:}A\langle c, e\rangle$.

$(\cdot, \cdot) \in \textbf{Hrel-IClient}^\xi(_\langle\top, \top\rangle)$
$(\mathcal{T}_1, \mathcal{T}_2) \in \textbf{Hrel-IClient}^\xi(x_\alpha{:}A\langle c, e\rangle)$ iff
  $e \not\sqsubseteq \xi$ **or** $e \sqsubseteq \xi$ and $x_\alpha{:}A\langle c, e\rangle \Vdash \mathcal{T}_1 \equiv_\xi^{\Psi_0} \mathcal{T}_2 :: _ : 1\langle\top, \top\rangle$

Figure 14 Asynchronous bisimulation up to observations of confidentiality ξ.

Now we are ready to present our adequacy theorem stating that, given an observer level ξ, logically equivalent configurations are bisimilar up to observations of confidentiality ξ. The proof of the theorem relies on a compositionality lemma, which ensures a harmony between asserts and assumes in the value-interpretation of the logical relation.

▶ **Lemma 2** (Compositionality). $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}^\xi_{\Psi_0}[\![\Delta, u_\alpha^{\langle c,e\rangle}{:}T \Vdash K]\!]^m$ and $\forall m. (\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}^\xi_{\Psi_0}[\![\Delta' \Vdash u_\alpha^{\langle c,e\rangle}{:}T]\!]^m$ *if and only if* $\forall k. (\mathcal{T}_1\mathcal{D}_1; \mathcal{T}_2\mathcal{D}_2) \in \mathcal{E}^\xi_{\Psi_0}[\![\Delta', \Delta \Vdash K]\!]^k$.

▶ **Theorem 3** (Adequacy). *If* $(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1\langle c_1, e_1\rangle) \equiv_\xi^{\Psi_0} (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2\langle c_2, e_2\rangle)$ *then* $(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1\langle c_1, e_1\rangle) \approx_a^\xi (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2\langle c_2, e_2\rangle)$.

Proof. Recall from Fig. 14 that $\approx_a^\xi$ composes $\mathcal{D}_1$ and $\mathcal{D}_2$ with arbitrary high-confidentiality ($c \not\sqsubseteq \xi$) clients and providers, building a confidentiality interface $[\![\Delta^c \Vdash K^c]\!]$. Let us call the high-confidentiality providers $\mathcal{B}_1$ and $\mathcal{B}_2$ and the high-confidentiality clients $\mathcal{T}_1$ and $\mathcal{T}_2$. We can partition the providers into high-integrity and low-integrity parts to get $\mathcal{B}_1 = \mathcal{B}_i^{HI}\mathcal{B}_i^{LI}$ (similarly for the clients $\mathcal{T}_1 = \mathcal{T}_i^{HI}\mathcal{T}_i^{LI}$), where superscripts HI and LI correspond to high-integrity and low-integrtiy parts, resp. Our goal is to prove $\mathcal{B}_1^{HI}\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{HI}\mathcal{T}_1^{LI} \approx_a \mathcal{B}_2^{HI}\mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{HI}\mathcal{T}_2^{LI}$.

Step 1. The first step is to show that the two compositions are related by the term interpretation as well, i.e., $\forall m.(\mathcal{B}_1^{HI}\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{HI}\mathcal{T}_1^{LI}; \mathcal{B}_2^{HI}\mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{HI}\mathcal{T}_2^{LI}) \in \mathcal{E}[\![\Delta^c \Vdash K^c]\!]$. To do so, we can use the definition from Fig. 13 for $\equiv_\xi^{\Psi_0}$ to compose $\mathcal{D}_1$ and $\mathcal{D}_2$ with given low integrity clients and providers $\mathcal{T}_1^{LI}$, $\mathcal{T}_2^{LI}$, $\mathcal{B}_1^{LI}$, and $\mathcal{B}_2^{LI}$ to build the integrity interface $[\![\Delta^i \Vdash K^i]\!]$ and get $\forall m.(\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{LI}; \mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{LI}) \in \mathcal{E}[\![\Delta^i \Vdash K^i]\!]$. However, this is not enough to achieve our goal as the relation pertains to the integrity interface, and thus, the composition only includes the low integrity providers/clients. To build the confidentiality interface and include the high integrity parts, we use the fact that the high integrity providers $\mathcal{B}_1^{HI}$ and $\mathcal{B}_2^{HI}$ (and clients $\mathcal{T}_1^{HI}$ and $\mathcal{T}_2^{HI}$) are themselves logically equivalent. We use our compositionality lemma (Lem. 2) to compose the high-integrity channels in the integrity interface with these providers/clients and show that the composition results in two logically equivalent configurations, i.e., $\forall m.(\mathcal{B}_1^{HI}\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{HI}\mathcal{T}_1^{LI}; \mathcal{B}_2^{HI}\mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{HI}\mathcal{T}_2^{LI}) \in \mathcal{E}[\![\Delta^c \Vdash K^c]\!]$.

Step 2. We complete the proof by connecting our logically related configurations to an observational equivalence relation for session types [7], which is proved sound and complete for asynchronous bisimulation. We first show that our integrity term interpretation implies the observational equivalence relation in [7] when we consider a confidentiality interface, and then use their soundness result to show that the integrity term interpretation $\forall m.(\mathcal{B}_1^{HI}\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{HI}\mathcal{T}_1^{LI};\mathcal{B}_2^{HI}\mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{HI}\mathcal{T}_2^{LI}) \in \mathcal{E}[\![\Delta^c \Vdash K^c]\!]$ implies bisimilarity $\mathcal{B}_1^{HI}\mathcal{B}_1^{LI}\mathcal{D}_1\mathcal{T}_1^{HI}\mathcal{T}_1^{LI} \approx_a \mathcal{B}_2^{HI}\mathcal{B}_2^{LI}\mathcal{D}_2\mathcal{T}_2^{HI}\mathcal{T}_2^{LI}$.

Next, we briefly explain how our integrity logical relation coincides with the observational equivalence relation (for well-typed configurations) in [7] when considering a confidentiality interface. The observational equivalence in [7] is defined via a logical relation similar to the one developed in this paper, but only considering the confidentiality interface. Let us call the term and value interpretations of our logical relation $\mathcal{E}^i$ and $\mathcal{V}^i$ (defined in Figs. 10–12) and the ones defined in [7] $\mathcal{E}^c$ and $\mathcal{V}^c$, resp. The relation $\mathcal{E}^i$ is invoked for the integrity interface $[\![\Delta^i \Vdash K^i]\!]$ in Fig. 13, and similarly $\mathcal{E}^c$ is invoked for the confidentiality interface $[\![\Delta^c \Vdash K^c]\!]$, where, by definition, Δ^c is a subset of (or equal to) Δ^i and K^c is a subset of (or equal to) K^i. As the integrity logical relation may contain non-observable channels ($c \not\sqsubseteq \xi$), it only insists that the same labels are sent when communication is along observable channels. Concretely, $\mathcal{V}^i$ in lines (l_2), (l_3), (r_2), and (r_3) only insists that the labels k_1 and k_2 sent/received along a channel are the same if $c \sqsubseteq \xi$. However, $\mathcal{V}^c$ always enforces sending the same labels, since a priori the condition $c \sqsubseteq \xi$ holds for all the channels in its interface. In all other regards, $\mathcal{E}^i$ and $\mathcal{V}^i$ have the same definition as $\mathcal{E}^c$ and $\mathcal{V}^c$. As a result, it is straightforward to observe that given an interface $\Delta^c \Vdash K^c$ with only observable channels (channels with $c \sqsubseteq \xi$), we have $\forall m.\,(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{E}^i[\![\Delta^c \Vdash K^c]\!]^m$ iff $\forall m.\,(\mathcal{D}_1;\mathcal{D}_2) \in \mathcal{E}^c[\![\Delta^c \Vdash K^c]\!]^m$. ◀

7 Related work

IFC type systems using linearity. Conceptually most closely related to our work is the work by Zdancewic and Myers on *ordered linear continuations* [46, 47]. The authors consider continuation-passing style (CPS) security-typed languages to verify noninterference not only for source-level programs but also compiled code. The authors observe that the possibility to lower the pc label upon exiting control flow constructs, present in imperative source-level languages, is no longer available in a CPS language. To rectify the loss of flexibility they introduce ordered linear continuations. Similar to our pattern checks, ordered linear continuations allow downgrading of the pc label after branching on high, because linearity enforces the continuations to be used in every branch, in the order prescribed. In contrast to our work, the authors only consider a sequential language and only prove PINI. Our work moreover establishes the connection to integrity, facilitating regrading policies that are polymorphic in the security lattice for ultimate flexibility.

In another line of work Zdancewic and Myers again employ linearity for increased flexibility and a stronger noninterference statement [48]. The authors consider a concurrent language with a store and first-class channels. Their main focus is observational determinism, ensuring immunity to internal timing attacks and attacks that exploit information about thread scheduling. To this end the authors introduce linear channels and a race freedom analysis. Given that SINTEGRITY enjoys confluence, like other linear session type languages, it rules out timing attacks that exploit the relative order of messages, which seems to be a stronger property than immunity to internal timing attacks considered by the authors. Moreover, we establish PSNI for SINTEGRITY, whereas the authors only prove PINI.

IFC session type systems. In terms of underlying language, the work most closely related to ours is the one by Derakhshan et al. [7, 20]. The authors develop an IFC type system for the same family of linear session types but only consider confidentiality. Their system annotates the process term judgments with running and max confidentiality. Their typing rules only ensure that the running confidentiality (aka taint level) is updated correctly after each receive and that a tainted process does not leak information via send. In particular, the rules do not allow decreasing the taint level at any point. As a result the authors' type system suffers from the same restrictiveness as other IFC type systems for concurrent languages, requiring each loop iteration to run at the maximal confidentiality reached throughout an arbitrary iteration. For example, the authors' IFC type system rejects the banking example in § 5.4: as soon as the bank receives a message from one customer, say Alice, it will be tainted and cannot send a message to any other customer, say Bob. In fact, the authors' IFC type system rejects all well-typed examples presented in this paper even though they enjoy PSNI. We make our IFC type system more flexible by designing synchronization policies to enable regrading of the taint level and using integrity labels to make the policies composable, both of which are novel to our system. Designing these composable policies was an intricate task, particularly due to dealing with both concurrency and general recursion.

Our logical relation for integrity is inspired by Balzer et al.'s [7] logical relation for equivalence. The logical relation for equivalence is defined based on the confidentiality interface. Our logical relation, however, is based on the larger integrity interface to enable the proof of the fundamental theorem. We prove our adequacy theorem by proving compositionality for our logical relation, which then allows us to recast our logical relation in terms of the logical relation for equivalence by the authors, delivering adequacy as a corollary.

IFC type systems for multiparty session types and process calculi. IFC type systems have also been explored for multiparty session types [10–13]. These works explore declassification [10, 12] and flexible runtime monitoring techniques [11, 13]. Our work not only differs in use of session type paradigm (i.e., binary vs. multiparty) but also in use of a logical relation for showing noninterference. Our work is more distantly related with IFC type systems for process calculi [16–18, 25, 25, 28, 29, 31, 34, 48]. These works prevent information leakage by associating a security label with channels/types/actions [29], read/write policies with channels [25, 25], or capabilities with expressions [16]. Honda et al. [29] also use a substructural type system and prove a sound embedding of Dependency Core Calculus (DCC) [2] into their calculus. Our work sets itself apart in its use of session types and meta theoretic developments based on logical relations. Moreover, our IFC type system is more permissive as it allows for regrading of the taint level, while preserving noninterference.

Declassification. Our notion of regrading may seem related to declassification, which has extensively been studied for IFC type systems for functional and imperative languages [1,6,15, 22,32,33,44,45,49] and allows an entity to downgrade its level of confidentiality. However, our work significantly differs from declassification as it preserves PSNI, whereas declassification systems *deliberately* release information and thus intentionally *weaken* noninterference.

In particular, robust declassification [6,15,33,44,45,49] prevents adversaries from exploiting downgrading of confidentiality, by complementing confidentiality with integrity. It uses integrity to ensure that downgrading decisions can be trusted, i.e., cannot be influenced by an attacker. As such, only high-integrity data can influence the taint level to be lowered. This is similar to our system, where the higher the integrity of a process, the lower level it can regrade the taint level. The difference, however, is that we enforce extra synchronization

policies on our high-integrity processes to ensure that they cannot induce information leaks by lowering the taint level. This contrasts with work on robust declassification, which introduces leakage intentionally and thus compromises noninterference.

References

1. Martín Abadi. Secrecy by typing in security protocols. In *TACS*, volume 1281 of *LNCS*, pages 611–638. Springer, 1997.
2. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160. ACM, 1999.
3. Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, volume 3924 of *LNCS*, pages 69–83. Springer, 2006.
4. Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA):118:1–118:26, 2018.
5. Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.
6. Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Log. Methods Comput. Sci.*, 7(3), 2011.
7. Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency. *CoRR*, abs/2309.00192, 2023. `arXiv:2309.00192`.
8. Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Electronic Systems Division, Air Force Systems Command, United States Air Force, 1977.
9. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
10. Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 238:68–105, 2014.
11. Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, 26(8):1352–1394, 2016. `doi:10.1017/S0960129514000619`.
12. Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In *CONCUR*, pages 237–252, 2010.
13. Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects Comput.*, 28(4):669–696, 2016.
14. Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Inf. Comput.*, 207(10):1044–1077, 2009.
15. Stephen Chong and Andrew C. Myers. Decentralized robustness. In *CSFW*, pages 242–256. IEEE, 2006.
16. Silvia Crafa, Michele Bugliesi, and Giuseppe Castagna. Information flow security for boxed ambients. *Electronic Notes in Theoretical Computer Science*, 66(3):76–97, 2002.
17. Silvia Crafa and Sabina Rossi. A theory of noninterference for the π-calculus. In *TGC*, volume 3705 of *LNCS*, pages 2–18. Springer, 2005.
18. Silvia Crafa and Sabina Rossi. Controlling information release in the pi-calculus. *Inf. Comput.*, 205(8):1235–1273, 2007.
19. Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI*, pages 50–63. ACM, 1999.
20. Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. Session logical relations for noninterference. In *LICS*, pages 1–14. IEEE, 2021.
21. Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. Regrading policies for flexible information flow control in session-typed concurrency. *CoRR*, 2024.

22 Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *IEEE Symposium on Security and Privacy*, pages 130–140. IEEE, 1997.

23 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

24 Daniel Heidin and Andrei Sabelfeld. A perspective on information flow control. Technical report, Marktoberdorf, 2011.

25 Matthew Hennessy. The security pi-calculus and non-interference. *J. Log. Algebraic Methods Program.*, 63(1):3–34, 2005.

26 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

27 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

28 Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *ESOP*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.

29 Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL*, pages 81–92. ACM, 2002.

30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

31 Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4):291–347, December 2005.

32 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4):410–442, 2000.

33 Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, 2006.

34 François Pottier. A simple view of type-secure information flow in the π-calculus. In *CSFW-15*, pages 320–330. IEEE, 2002.

35 Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *SAS*, volume 2477 of *LNCS*, pages 376–394. Springer, 2002.

36 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

37 Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214. IEEE, 2000.

38 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

39 Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM, 1998.

40 Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, pages 201–214. ACM, 2012.

41 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.

42 Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.

43 Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.

44 Steve Zdancewic. A type system for robust declassification. In *MFPS*, volume 83 of *Electronic Notes in Theoretical Computer Science*, pages 263–277. Elsevier, 2003.

45 Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW*, pages 15–23. IEEE, 2001.

46 Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *ESOP*, volume 2028 of *LNCS*, pages 46–61. Springer, 2001.

47 Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *High. Order Symb. Comput.*, 15(2-3):209–234, 2002.

48 Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 1–15. IEEE, 2003.

49 Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *TOCS*, 20(3):283–328, 2002.