2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) | 978-1-6654-4895-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/LICS52264.2021.9470654

Session Logical Relations for Noninterference

Farzaneh Derakhshan
Carnegie Mellon University

Stephanie Balzer
Carnegie Mellon University

Limin Jia
Carnegie Mellon University

***Abstract*—Information flow control type systems statically restrict the propagation of sensitive data to ensure end-to-end confidentiality. The property to be shown is noninterference, asserting that an attacker cannot infer any secrets from made observations. Session types delimit the kinds of observations that can be made along a communication channel by imposing a protocol of message exchange. These protocols govern the exchange along a single channel and leave unconstrained the propagation along adjacent channels. This paper contributes an information flow control type system for linear session types. The type system stands in close correspondence with intuitionistic linear logic. Intuitionistic linear logic typing ensures that process configurations form a tree such that client processes are parent nodes and provider processes child nodes. To control the propagation of secret messages, the type system is enriched with secrecy levels and arranges these levels to be aligned with the configuration tree. Two levels are associated with every process: the maximal secrecy denoting the process' security clearance and the running secrecy denoting the highest level of secret information obtained so far. The computational semantics naturally stratifies process configurations such that higher-secrecy processes are parents of lower-secrecy ones, an invariant enforced by typing. Noninterference is stated in terms of a logical relation that is indexed by the secrecy-level-enriched session types. The logical relation contributes a novel development of logical relations for session typed languages as it considers open configurations, allowing for a more nuanced equivalence statement.**

I. Introduction

Message-passing is a successful concurrency paradigm, adopted by languages such as Erlang, Go, and Rust. In this setting, a program amounts to a number of processes connected via channels, and computation happens by the concurrent exchange of messages along these channels. To prescribe the *protocols* of message exchange and assert their adherence at run-time, *session types* [1], [2] were introduced. Various session-typed programming languages have been designed since then [3]–[5] as well as session type libraries for mainstream languages [6]–[15]. Session types also enjoy a logical foundation by a Curry-Howard correspondence between *linear logic* and the session-typed π-calculus [16]–[18].

Many real-world systems, such as OS processes, Android apps, and web applications, can naturally be modeled with session types. In addition to static protocol assurance, the prevention of *information leakage* is another desirable property in such systems. Accidental or malicious leakage can be prevented by an *information flow control (IFC)* type system. Such a type system restricts the propagation of information and guarantees *noninterference*, asserting that an adversary cannot infer any secrets from observing exchanged messages [19], [20]. While prior work has investigated IFC type systems for process calculi [21]–[31] as well as run-time monitoring for OS processes, Android apps, and web applications [32]–[35], very few IFC session type systems exist [36], [37]. In particular, no one has investigated information flow types in the context of linear binary session types based on intuitionistic linear logic, which naturally accommodate *flow sensitivity*.

978-1-6654-4895-6/21/$31.00 ©2021 IEEE

This paper develops a flow-sensitive IFC session type system for the language $\mathsf{SILL}_{\mathsf{sec}}$ and proves noninterference for $\mathsf{SILL}_{\mathsf{sec}}$ in addition to type safety. $\mathsf{SILL}_{\mathsf{sec}}$ is a terminating language with higher-order channels, allowing channels to be sent along channels. It builds on the Curry-Howard correspondence between *intuitionistic* linear logic and the session-typed π-calculus [16], [38]. The intuitionistic foundation turns run-time configurations of processes into *trees*, connecting a *providing* process with *exactly one client*.

The $\mathsf{SILL}_{\mathsf{sec}}$ type system takes advantage of the tree structure imposed by intuitionism and stratifies process trees according to the security order. Two secrecy levels are associated with each process: the *maximal secrecy*, denoting the maximal level of information the process may ever obtain, and the *running secrecy*, denoting the highest level of information a process has obtained so far and whose changes are tracked by the type system. To align the process tree with the security lattice, typing asserts the following invariant, for any node in the tree: *(i)* the maximal secrecy of a child node is at most as high as the maximal secrecy of the parent node and *(ii)* the running secrecy of the parent node is capped by its maximal secrecy. By complementing the maximal secrecy with a running secrecy, the $\mathsf{SILL}_{\mathsf{sec}}$ type system becomes *flow-sensitive*, allowing more secure programs to successfully type check than would be possible with maximal secrecy alone.

Noninterference of $\mathsf{SILL}_{\mathsf{sec}}$ is stated in terms of a *logical relation* [39], [40]. The use of logical relations for session types has focused predominantly on unary logical relations (predicates) for proving termination [41]–[43], with the exception of a binary logical relation for parametricity [44]. Noninterference, however, demands a more nuanced binary relation, requiring observation of a process' communication not only with its client but also with all the processes it is a client of. We generalize binary logical relations for session-typed languages to support *open configurations*, considering both the antecedent and succedent of the typing judgment.

In summary, the paper makes the following contributions:

- development of an flow-sensitive IFC type system for binary session types, yielding the language $\mathsf{SILL}_{\mathsf{sec}}$;
- proofs of type safety and noninterference of $\mathsf{SILL}_{\mathsf{sec}}$;

- generalization of (binary) logical relations to the session typed setting, supporting open configurations and higher-order channels.

Paper structure: Sect. II familiarizes with information flow control and intuitionistic session-typed programming. Sect. III develops the main ideas underlying the $\mathsf{SILL}_{\mathsf{sec}}$ type system, concretized in Sect. IV. Sect. V develops the main ideas underlying the session logical relation, detailed in Sect. VI. Sect. VII proves type safety and noninterference of $\mathsf{SILL}_{\mathsf{sec}}$. Sect. VIII summarizes related and future work. Further technical developments and proofs can be found in [45].

II. Motivating Example

This section provides an introduction to programming with intuitionistic linear logic session types [3]–[5], [38] based on a banking example and illustrates violations of end-to-end confidentiality. We base the discussion on the language $\mathsf{SILL}_{\mathsf{sec}}$ that we formalize and for which we prove noninterference in the remainder of this paper. $\mathsf{SILL}_{\mathsf{sec}}$ is a terminating language with higher-order channels, allowing channels to be sent over channels.

In $\mathsf{SILL}_{\mathsf{sec}}$, we can define the protocol according to which an authorization process interacts with a customer seeking access to their bank account as follows:

$$\begin{array}{rl}\mathsf{auth} = & \&\{\mathit{tok}_1\colon \oplus\{\mathit{succ}\colon \mathsf{account} \otimes 1, \mathit{fail}\colon 1\}, \ldots, \\ & \mathit{tok}_n\colon \oplus\{\mathit{succ}\colon \mathsf{account} \otimes 1, \mathit{fail}\colon 1\}\}\end{array}$$

The connectives $\&$, $\oplus$, $\otimes$, and 1 can be found in Table I, providing an overview of intuitionistic linear session types and their operational reading. The first column indicates the session type before the message exchange, the second column the session type after the exchange. The corresponding process terms are listed in the third and fourth column, respectively. The fifth column provides the operational meaning of a connective and the last column its *polarity*. Positive connectives have a sending semantics, negative connectives a receiving semantics.

Linearity ensures that a channel connects exactly two processes. An intuitionistic viewpoint moreover allows the distinction of one process as the *provider* and the other as the *client*, where linearity ensures that every providing process has *exactly one* client process. As a result, channels in intuitionistic linear session type languages can be typed with the session type of the providing process. In developments of linear session types based on classical logic [17], the two endpoints of a channel are instead typed separately, using linear negation to make sure that the two endpoint types are dual to each other. The fact that a provider process and client process must behave dually to each other surfaces in an intuitionistic setting at the level of the process terms, which come in matching pairs. Table I lists the process term of a provider in the first line for each connective and the client's term in the second line.

The above session type auth thus requires the client to send their authorization token (tok_i), after which the authorization process will respond with succ in case of successful authorization and fail, otherwise. In the former case, the authorization process sends the channel to the customer's bank account and then terminates, in the latter case it just terminates. A corresponding authorization process is implemented for each customer, accepting only the customer's authorization token. We assume that session type auth includes a label tok_i for every imaginable authorization token.

We complete the example with the addition of the following session types:

$$\begin{array}{ll}\mathsf{customer} & = \mathsf{auth} \multimap 1 \\ \mathsf{account} & = \oplus\{\mathit{high}{:}1,\ \mathit{med}{:}1,\ \mathit{low}{:}1\} \\ \mathsf{rate} & = \&\{\mathit{lowRate}{:}1,\ \mathit{highRate}{:}1\}\end{array}$$

As the names suggest, customer denotes the protocol of a customer process, indicating that it is waiting to receive an authorization channel, after which it eventually terminates. A bank account process (session type account), on the other hand, will indicate whether its balance is high (high), medium (med), or low (low) and then terminate. The last session type rate allows a bank to advertise the current interest rate, for example by displaying it on a bulletin board.

For our example, we assume that the bank has two customers, Alice and Bob, which own accounts with the bank. In a secure system, Alice's account can only be queried by Alice or the bank, but neither by Bob or any walk-in customer. The same must hold for Bob's account. We can express these dependencies by defining corresponding secrecy levels and a lattice on them:

$$\mathbf{guest} \sqsubseteq \mathbf{alice} \sqsubseteq \mathbf{bank} \qquad \mathbf{guest} \sqsubseteq \mathbf{bob} \sqsubseteq \mathbf{bank}$$

We next show the corresponding process implementations concerning Alice. We first define process Alice for the alice customer process:

$$\begin{array}{l}\cdot \vdash \mathsf{Alice} :: y\colon \mathsf{customer}[\mathbf{alice}] \\ y \leftarrow \mathsf{Alice} \leftarrow \cdot = (\ \text{//}\ \cdot \vdash y{:}\mathsf{customer} \\ \quad w \leftarrow \mathbf{recv}\, y; w.\mathit{tok}_j;\ \text{//}\ w\colon \oplus\{\mathit{succ}\colon \mathsf{account} \otimes 1, \mathit{fail}\colon 1\} \vdash y{:}1 \\ \quad \mathbf{case}\, w\, (\mathit{succ} \Rightarrow v \leftarrow \mathbf{recv}\, w;\ \text{//}\ w{:}1, v{:}\mathsf{account} \vdash y{:}1 \\ \qquad\qquad\quad \mathbf{case}\, v\, (\mathit{high} \Rightarrow \mathbf{wait}\, v; \mathbf{wait}\, w; \mathbf{close}\, y \\ \qquad\qquad\qquad\qquad \mid \mathit{med} \Rightarrow \mathbf{wait}\, v; \mathbf{wait}\, w; \mathbf{close}\, y \\ \qquad\qquad\qquad\qquad \mid \mathit{low} \Rightarrow \mathbf{wait}\, v; \mathbf{wait}\, w; \mathbf{close}\, y) \\ \qquad\qquad\quad \mid \mathit{fail} \Rightarrow \mathbf{wait}\, w; \mathbf{close}\, y))@\mathbf{alice}\end{array}$$

The first line of the above process definition denotes the process' signature. It is in line with the process term typing judgment introduced in Sect. IV and indicates that process Alice provides a session of type customer along channel y without being a client of any other sessions (denoted by $\cdot$ on the left of the turnstile). The next line introduces the bindings of channels variables to be used in the body of the process, appearing to the right of the $=$ sign. We generally use the symbol $\leftarrow$ denote variable bindings. For the time being, we ignore the secrecy annotations $[\mathbf{alice}]$ and $@\mathbf{alice}$.

In its body, the Alice process first receives a channel to Alice's authorization process. Along this channel it then sends Alice's authorization token. If that token is correct, the authorization process will respond by sending a channel to Alice's account process. Otherwise, the Alice process waits for the authorization process to terminate and then terminates itself. In case of successful authentication, the Alice process queries its account process for its balance, willing to receive

Session type (current / cont)		Process term (current / cont)		Description	Pol
$x : \oplus\{\ell : A\}_{\ell\in L}$	$x : A_k$	$x.k; P$	P	provider sends label k along x and continues with P	+
		$\mathbf{case}\, x(\ell \Rightarrow Q_\ell)_{\ell\in L}$	Q_k	client receives label k along x and continues with Q_k	
$x : \&\{\ell : A\}_{\ell\in L}$	$x : A_k$	$\mathbf{case}\, x(\ell \Rightarrow P_\ell)_{\ell\in L}$	P_k	provider receives label k along x and continues with P_k	-
		$x.k\ Q$	Q	client sends label k along x and continues with Q	
$x : A \otimes B$	$x : B$	$\mathbf{send}\, y\, x; P$	P	provider sends channel $y{:}A$ along x and continues with P	+
		$z \leftarrow \mathbf{recv}\, x; Q$	$[y/z]\, Q_z$	client receives channel $y{:}A$ along x and continues with Q	
$x : A \multimap B$	$x : B$	$z \leftarrow \mathbf{recv}\, x; P$	$[y/z]\, P_z$	provider receives channel $y{:}A$ along x and continues with P	-
		$\mathbf{send}\, y\, x; Q$	Q	client sends channel $y{:}A$ along x and continues with Q	
$x : 1$	-	$\mathbf{close}\, x$	-	provider sends "end" along x and terminates	+
		$\mathbf{wait}\, x; Q$	Q	client receives "end" along x and continues with Q	

TABLE I: Overview of intuitionistic linear session types in $\mathsf{SILL}_{\mathsf{sec}}$ together with their operational meaning.

any of the labels *high*, *med*, or *low*, and then waits for the authorization and account processes to terminate, before terminating itself.

A distinguishing feature of session type programming is that channels and the processes offering along those channels change their types along with the messages exchange. It is instructive to walk through the body of process Alice to follow these state changes, consulting Table I as needed. We include annotations as comments, indicating the types of all channels existing at the various points in the code[1].

Next, we show the implementation of Alice's authorization process aAuth. This process offers a session of type auth along its offering channel x and uses a process along channel u, which offers a choice between access to Alice's account process (label s) or a terminating process (label f). The aAuth process waits to receive an authorization token along its offering channel. If the sent token is Alice's authorization token (tok_j), the authorization process sends the label *succ* along its offering channel as well as the label s along channel u, after which it sends the channel u providing access to Alice's account process along x and then terminates. Otherwise, the authorization process sends the labels *fail* and f along channel x and u, respectively, waits for u to terminate and then terminates itself.

$$
\begin{array}{l}
u{:}\,\&\{s{:}\mathsf{account}, f{:}1\}[\mathbf{alice}] \vdash \mathsf{aAuth} :: x{:}\mathsf{auth}[\mathbf{alice}] \\
x \leftarrow \mathsf{aAuth} \leftarrow u = (\\
\quad \mathbf{case}\, x\ (tok_j \Rightarrow x.succ; u.s; \mathbf{send}\, u\, x; \mathbf{close}\, x \\
\qquad\qquad \mid tok_{i\neq j} \Rightarrow x.fail; u.f; \mathbf{wait}\, u; \mathbf{close}\, x))@\mathbf{alice}
\end{array}
$$

The implementation of Alice's account process aAcc is finally shown below. We leave it to the reader to walk through the code, consulting Table I as needed.

$$
\begin{array}{l}
\cdot \vdash \mathsf{aAcc} :: u{:}\&\{s{:}\mathsf{account}, f{:}1\}[\mathbf{alice}] \\
u \leftarrow \mathsf{aAcc} \leftarrow \cdot = (\\
\quad \mathbf{case}\, u\, (s \Rightarrow u.high; \mathbf{close}\, u \\
\qquad\qquad \mid f \Rightarrow \mathbf{close}\, u))@\mathbf{alice}
\end{array}
$$

It is instructive to look at the implementation of the bank process, which instantiates our running example. We assume corresponding process definitions for Bob and the rate to be displayed on the bulletin board.

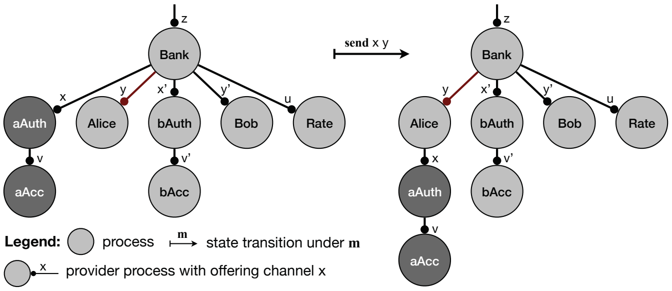


Fig. 1: State transition in process configuration due to $\multimap$.

$$
\begin{array}{l}
x{:}\,\mathsf{auth}[\mathbf{alice}], y{:}\,\mathsf{customer}[\mathbf{alice}], x'{:}\mathsf{auth}[\mathbf{bob}], \\
y'{:}\,\mathsf{customer}[\mathbf{bob}], u{:}\,\mathsf{rate}[\mathbf{guest}] \vdash \mathsf{Bank} :: z{:}1[\mathbf{bank}] \\
z \leftarrow \mathsf{Bank} \leftarrow x, x', y, y', u = (\\
\quad \mathbf{send}\, x\, y; \mathbf{send}\, x'\, y'; u.lowRate; \\
\quad \mathbf{wait}\, y; \mathbf{wait}\, y'; \mathbf{wait}\, u; \mathbf{close}\, z)@\mathbf{guest}
\end{array}
$$

Fig. 1 shows the run-time configuration of processes that exist before and after executing the first statement in the above code. Intuitionistic linear typing imposes a *tree* structure on process configurations such that client processes are parent nodes and provider processes child nodes. Fig. 1 also demonstrates that message exchanges may not only change the type of a channel and its offering process but also the structure of the tree. Changes in the tree structure, in particular, are due to the connectives $\multimap$ and $\otimes$, which make a sibling subtree the child of the recipient and a child subtree a sibling of the sender, respectively.

It is time to ask ourselves whether the Bank process is actually secure. For this purpose we now consider the red secrecy annotations [d]. These annotations indicate the *maximal secrecy* of a process, i.e., the maximal level of secret information the process may ever obtain. As to be expected, the processes Alice, aAuth, and aAcc have maximal secrecy [alice] because they know Alice's authorization token and account balance. Similarly, the processes associated with Bob have maximal secrecy [bob]. The Bank process itself has the highest maximal secrecy of [bank]. The process associated with the rate bulletin board, on the other hand, has the lowest maximal secrecy [guest] because information about interest rates are available to any walk-in customer. Given

[1] We have omitted secrecy annotations for compactness.

these annotations and the security lattice defined earlier, we can conclude that process Bank is secure: it sends Alice's authorization process to Alice and Bob's authorization process to Bob, but not other way around.

Next, let's ask the same question for the below LeakyBank process implementation. As its name suggests, this implementation is not secure. Information is leaked by sending the channel to Alice's authorization process to a customer with a maximal secrecy of [**guest**], potentially allowing such a customer to get access to Alice's bank account.

```
x: auth[alice], y: customer[guest] ⊢ LeakyBank :: z:1[bank]
z ← LeakyBank ← x, y = (
  send x y; // insecure send
  wait y; close z)@guest
```

While process LeakyBank contains what is referred to as a *direct* flow there also exist *indirect* flows, which are more subtle. For example, consider the below process definition SneakyaAuth that not only authenticates Alice but also indirectly leaks information about whether Alice's authorization was successful to the adversary x_1.

```
x1:&{s:1, f:1}[guest],
u: &{s:account, f:1}[alice] ⊢ SneakyaAuth :: x:auth[alice]
x ← SneakyaAuth ← u, x1 = (
   case x (tok_j ⇒ x.succ; u.s; x1.s; // insecure send
                   send u x; wait x1; close x
          | tok_{i≠j} ⇒ x.fail; u.f; x1.f; // insecure send
                         wait u; wait x1; close x))@alice
```

Process SneakyaAuth is not secure because the sends to the adversary x_1 with maximal secrecy [**guest**] happen when branching on channel x whose maximal secrecy is [**alice**].

To rule out indirect information flows in SILL$_{\mathsf{sec}}$, we complement the maximal secrecy of a process with its *running secrecy*, occurring as green process term level annotations @**c**. The running secrecy denotes the highest level of secret information a process has obtained so far. When defining a process, a programmer must indicate the process' maximal secrecy as well as the *initial* running secrecy the process starts out with when spawned. As we will see in Sections IV and III, the SILL$_{\mathsf{sec}}$ type system increases the running secrecy accordingly whenever information of higher secrecy is received and disallows sends from contexts of a higher running secrecy than the one of the receiver.

III. Key Ideas - Part I

This section develops the main ideas underlying the SILL$_{\mathsf{sec}}$ type system.

The banking example discussed in the previous section reveals that a process configuration naturally aligns with the security lattice of the application: processes with higher maximal secrecy are ancestors (direct or transitive parents) of processes with lower or same maximal secrecy. For the Bank configuration shown in Fig. 1, for example, the Bank process has the top maximal secrecy [**bank**] and is the root process of the configuration, whereas all its descendants (direct or transitive children) have a lower maximal secrecy.

We can impose this property as a *presupposition* on the typing judgment for process terms:

$$\Psi; \Delta \vdash P@c :: (x{:}A[d])$$

with presuppositions:

(i) $\forall y{:}B[d'] \in \Delta\ (\Psi \Vdash d' \sqsubseteq d)$
(ii) $\Psi \Vdash c \sqsubseteq d$

The typing judgment states that process P with maximal secrecy $[d]$ and running secrecy $@c$ provides a session of type A along channel variable x, given the typing of sessions offered along channel variables in Δ and given the secrecy levels in the security lattice Ψ. Δ is a *linear* context that consists of a finite set of assumptions of the form $y_i{:}B_i[d'_i]$, indicating for each channel variable y_i its maximal secrecy $[d'_i]$ and the offered session type B_i. Channel variables y_i must be unique in Δ and different from x. This well-formedness condition together with the fact the sequent has exactly one succedent, turns process configurations into trees. The process P under consideration is the parent node of all the processes providing along channels in Δ.

We point out our use of "channel variable" for x and y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables occurring in process terms. As a result, channel variables can be α-varied, as usual. For brevity, we will use the term channel rather than channel variable, whenever the context determines whether a variable or run-time channel is meant.

The presuppositions guarantee that *(i)* the maximal secrecy of a child node is at most as high as the maximal secrecy of the providing (parent) node and that *(ii)* the running secrecy of the providing (parent) node is capped by its maximal secrecy. By transitivity, assertion *(i)* holds equally for any descendant of the providing node. Assertion *(ii)* ensures that a node can never obtain more secrets than it is licensed to. We refer to both assertions as the *tree invariant*. Stating the tree invariant as a presupposition requires the process term typing rules to preserve, but not to establish the invariant. This is sufficient because the tree invariant holds for any well-typed process configuration, as expressed by the configuration typing rules discussed in Sect. IV-C.

The tree invariant is sufficient to rule out any direct flows. For example, the attempt to send Alice's authorization process to a walk-in customer in process LeakyBank (see Sect. II), violates the tree invariant and thus does not type-check. The tree invariant, however, is not sufficient to rule out indirect flows. To tackle indirect flows the type system must make sure that the running secrecy of a process always soundly reflects the level of secret information a process has obtained so far. To this end, it increases the running secrecy upon each receive and correspondingly guards sends, according to the following schema:

(i) **After** receiving a message, the running secrecy of the receiving process must be increased to **at least** the maximal secrecy of the sending process, and

(ii) **before** sending a message, the running secrecy of the sending process must be **at most** the maximal secrecy of the receiving process.

This schema intimately relies on the tree invariant and uses the maximal secrecy as a sound approximation for the running secrecy of a process. We refer to it as the *secrecy pas de deux*. The next section puts the discussed ideas into action.

IV. IFC Session Type System

This section formalizes $\mathsf{SILL}_{\mathsf{sec}}$, giving the process term typing, configuration typing, and asynchronous semantics. The system implements the ideas discussed in the previous section to rule out both direct and indirect information flows. We defer proofs of type safety and noninterference to Sect. VII.

A. Process Typing

Our process typing rules are based on the *sequent calculus*, leading to a left and a right rule for each connective, describing the interaction from the point of view of the provider and client, respectively. We first discuss the rules for the individual connectives in Table I and then conclude with the judgmental rules cut and identity.

1) Internal and External Choice: Internal ($\oplus$) and external ($\&$) choice are the branching constructs, giving the choice to the provider or the client, respectively.

$$\frac{\Psi; \Delta \vdash P@d_1 :: y{:}A_k[c] \qquad k \in L}{\Psi; \Delta \vdash (y.k; P)@d_1 :: y{:} \oplus \{\ell{:}A_\ell\}_{\ell \in L}[c]} \oplus R$$

$$\frac{\begin{array}{c}\Psi \Vdash d_2 = c \sqcup d_1 \\ \Psi; \Delta, x{:}A_k[c] \vdash Q_k@d_2 :: y{:}C[c'] \quad \forall k \in L\end{array}}{\Psi; \Delta, x{:} \oplus \{\ell : A_\ell\}_{\ell \in L}[c] \vdash (\mathbf{case}\, x(\ell \Rightarrow Q_\ell)_{\ell \in L})@d_1 :: y{:}C[c']} \oplus L$$

$$\frac{\Psi; \Delta \vdash Q_k@c :: y{:}A_k[c] \qquad \forall k \in L}{\Psi; \Delta \vdash (\mathbf{case}\, y(\ell \Rightarrow Q_\ell)_{\ell \in I})@d_1 :: y{:}\&\{\ell : A_\ell\}_{\ell \in L}[c]} \&R$$

$$\frac{\Psi \Vdash d_1 \sqsubseteq c \qquad \Psi; \Delta, x{:}A_k[c] \vdash P@d_1 :: y{:}C[c'] \qquad k \in L}{\Psi; \Delta, x{:}\&\{\ell : A_\ell\}_{\ell \in I}[c] \vdash (x.k; P)@d_1 :: y{:}C[c']} \&L$$

Let's convince ourselves that the rules preserve the tree invariant. To preserve the invariant, we may assume that the invariant holds for the conclusion and must establish it for the premise. Since the rules do neither add to or remove any channels from Δ, they preserve the invariant by assumption. Let's examine whether the rules implement the secrecy pas de deux. In case of a receive, the running secrecy of the continuation must be increased to at least the maximal secrecy of the sending channel. In $\oplus L$, the premise $d_1 \sqcup c$ makes this adjustment. In $\&R$, no explicit adjustment is needed because the new running secrecy $d_1 \sqcup c$ amounts to c, by the tree invariant. In case of a send, on the other hand, the send is only admissible if the running secrecy of the sender is at most the maximal secrecy of the receiving channel. In $\oplus R$, this guard ($d_1 \sqsubseteq c$) is already established by the tree invariant. $\&L$ explicitly establishes the guard with the premise $\Psi \Vdash d_1 \sqsubseteq c$.

2) Higher-Order Channels: Tensor ($\otimes$) and lolli ($\multimap$) denote channel output (send) and input (receive), respectively.

$$\frac{\Psi; \Delta \vdash P@d_1 :: y{:}B[c]}{\Psi; \Delta, z{:}A[c] \vdash (\mathbf{send}\, z\, y; P)@d_1 :: y{:}A \otimes B[c]} \otimes R$$

$$\frac{\begin{array}{c}d_2 = c \sqcup d_1 \qquad \Psi' := (\Psi, \psi = c) \\ \Psi'; \Delta, z : A[\psi], x{:}B[c] \vdash P@d_2 :: y{:}C[c']\end{array}}{\Psi; \Delta, x{:}A \otimes B[c] \vdash (z \leftarrow \mathbf{recv}\, x; P)@d_1 :: y{:}C[c']} \otimes L$$

$$\frac{\Psi' := (\Psi, \psi = c) \qquad \Psi'; \Delta, z{:}A[\psi] \vdash P@c :: y{:}B[c]}{\Psi; \Delta \vdash (z \leftarrow \mathbf{recv}\, y; P)@d_1 :: y{:}A \multimap B[c]} \multimap R$$

$$\frac{\Psi \Vdash d_1 \sqsubseteq d \qquad \Psi; \Delta, x{:}B[d] \vdash P@d_1 :: y{:}C[c']}{\Psi; \Delta, z{:}A[d], x{:}A \multimap B[d] \vdash (\mathbf{send}\, z\, x; P)@d_1 :: y{:}C[c']} \multimap L$$

To understand that the rules preserve the tree invariant, it is helpful to remind ourselves that the connectives $\otimes$ and $\multimap$ change the tree structure, making a child a sibling of the sender and a sibling the child of the recipient, respectively. $\otimes R$ preserves the tree invariant without any extra conditions. By assumption we know that the maximal secrecy of the sent channel is equal to the maximal secrecy c of the provider. Also by assumption, we know that the maximal secrecy of the provider is less than or equal to the one of its parent, *ensuring* that the tree invariant is preserved for $\otimes L$ as well. $\otimes R$ also implements the secrecy pas de deux, since $d_1 \sqsubseteq c$ by assumption. While $\otimes R$ license us to *assume* in $\otimes L$ that the maximal secrecy ψ of the received channel z is equal to the maximal secrecy c of the sending channel x, the actual maximal secrecy level of z is statically unknown. As a result, ψ stands for a secrecy *variable*, and we extend the security lattice with $\psi = c$. The premise $d_2 = c \sqcup d_1$ in $\otimes L$ lastly implements the secrecy pas de deux, raising the running secrecy of the continuation P to c, unless $c \sqsubseteq d_1$. The reasoning for $\multimap R$ and $\multimap L$ are analogous, but with the roles reversed.

3) Termination: The multiplicative unit (1) denotes process termination.

$$\frac{}{\Psi; \cdot \vdash (\mathbf{close}\, y)@d_1 :: y{:}1[c]} 1R$$

$$\frac{\Psi \Vdash d_2 = c \sqcup d_1 \qquad \Psi; \Delta \vdash Q@d_2 :: y{:}T[d]}{\Psi; \Delta, x{:}1[c] \vdash (\mathbf{wait}\, x; Q)@d_1 :: y{:}C[d]} 1L$$

$1R$ trivially preserves the tree invariant because there is no continuation and implements the secrecy pas de deux since $d_1 \sqsubseteq c$ by assumption. Similarly, $1L$ preserves the tree invariant by simply removing a channel from the continuation and implements the secrecy pas de deux with the left premise.

4) Identity and Cut: Identity and cut are the two rules that do not result in any communication. Identity amounts to termination after identifying the involved channels and cut to process spawning. For simplicity, we do not support process definitions. The examples from Sect. II can be rewritten by inlining the body of the process definition when called.

$$\frac{}{\Psi; x{:}A[c] \vdash (y \leftarrow x)@d_1 :: y{:}A[c]} \mathsf{Fwd}$$

$$\frac{\begin{array}{c}\Psi \Vdash d_1 \sqsubseteq d_2 \sqsubseteq d' \\ \forall\, z{:}A[c'] \in \Delta_1 .\, \Psi \Vdash c' \sqsubseteq d' \qquad \Psi; \Delta_1 \vdash P@d_2 :: x : B[d'] \\ \Psi \Vdash d' \sqsubseteq d \qquad \Psi; x : B[d'], \Delta_2 \vdash Q@d_1 :: y{:}C[d]\end{array}}{\Psi; \Delta_1, \Delta_2 \vdash ((x^{d'} \leftarrow P)@d_2; Q)@d_1 :: y{:}C[d]} \mathsf{Cut}$$

We briefly comment on Cut. The premise $\Psi \Vdash d' \sqsubseteq d$ establishes the tree invariant for the continuation Q and the premise $\forall\, z{:}A[c'] \in \Delta_1 .\, \Psi \Vdash c' \sqsubseteq d'$ for the spawned process P. The premise $\Psi \Vdash d_1 \sqsubseteq d_2 \sqsubseteq d'$ is vital to prevent any indirect flows from Q via P. It ensures that the newly spawned process has at least the knowledge of secret information that its spawner has. Thanks to this premise the below insecure example, which indirectly leaks information about the success of Alice's authorization to the adversary x_1, is rejected.

$$
\begin{array}{l}
x_1{:}\&\{s{:}1, f{:}1\}[\mathbf{guest}],\\
u{:}\,\&\{s{:}\mathsf{account}, f{:}1\}[\mathbf{alice}] \vdash \mathsf{SneakyaAuth} :: x{:}\mathsf{auth}[\mathbf{alice}]\\
x \leftarrow \mathsf{SneakyaAuth} \leftarrow u, x_1 = (\\
\quad \mathbf{case}\, x\, (tok_j \Rightarrow x.succ; u.s; z_1 \leftarrow S \leftarrow x_1;\ \text{// insecure spawn}\\
\qquad\qquad\qquad\qquad \mathbf{send}\, u\, x; \mathbf{wait}\, z_1; \mathbf{close}\, x\\
\qquad\qquad\quad \mid tok_{i \neq j} \Rightarrow x.fail; u.f; z_1 \leftarrow F \leftarrow x_1;\ \text{// insecure spawn}\\
\qquad\qquad\qquad\qquad\quad \mathbf{wait}\, u; \mathbf{wait}\, z_1; \mathbf{close}\, x))@\mathbf{alice}
\end{array}
$$

$$
\begin{array}{l}
x_1{:}\&\{s{:}1, f{:}1\}[\mathbf{guest}] \vdash \mathsf{S} :: z_1{:}1[\mathbf{alice}]\\
z_1 \leftarrow \mathsf{S} \leftarrow x_1 = (x_1.s; \mathbf{wait}\, x_1; \mathbf{close}\, z_1)@\mathbf{guest}
\end{array}
$$

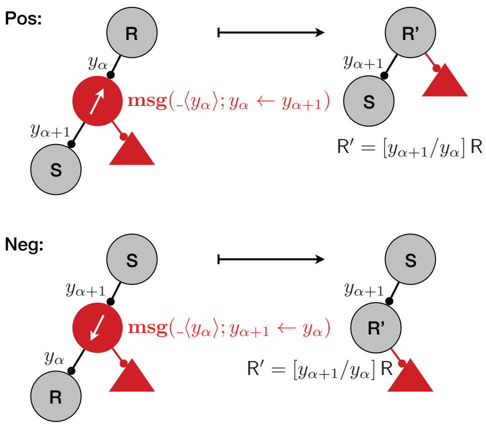
$$
\begin{array}{l}
x_1{:}\&\{s{:}1, f{:}1\}[\mathbf{guest}] \vdash \mathsf{F} :: z_1{:}1[\mathbf{alice}]\\
z_1 \leftarrow \mathsf{F} \leftarrow x_1 = (x_1.f; \mathbf{wait}\, x_1; \mathbf{close}\, z_1)@\mathbf{guest}
\end{array}
$$

B. Asynchronous Dynamics

We define an *asynchronous* dynamics for $\mathsf{SILL}_{\mathsf{sec}}$ because it is not only more practical but also allows for a more accurate statement of noninterference. The dynamics is in line with [5], [46], with the difference that it considers open configurations. The result is shown in Fig. 3. We first convey the main ideas and then comment on selected rules.

In an asynchronous semantics only receivers can be blocked, while senders just output the message and proceed with their continuation. We model such outputted messages as special $\mathbf{msg}(P)$ processes that just contain the particular message. In order to ensure that an outputted message is properly sequenced with the sender's continuation, we use forwarding. Fig. 2 schematically illustrates this idea, showing the case of a *positive* (sending) connective in the first line and the case of a *negative* (receiving) connective in the second line, with S and R standing for the sending and receiving process, respectively. The message process $\mathbf{msg}(P)$ is depicted in red. This process has a subtree, in case of $\otimes$ and $\multimap$. We can think of the message as being spawned by the sender. This results in the allocation of a new generation $y_{\alpha+1}$ of the carrier channel y_α. The forward then links the two generations $y_{\alpha+1}$ and y_α appropriately. In case of a positive connective, $y_{\alpha+1}$ is forwarded to y_α, in case of a negative connective, y_α is forwarded to $y_{\alpha+1}$. Once the message has been received, it terminates and $y_{\alpha+1}$ is substituted for y_α in the receiver's continuation R$'$. Messages can be "queued up" as long as the polarity of the carrier channel stays the same. Session typing ensures that any messages "in flight" must first be received before the polarity of the carrier channel changes.

Fig. 3 defines the asynchronous dynamics in terms of rewriting rules $\mathcal{C} \mapsto_{\Delta \Vdash \Delta'} \mathcal{C}'$ that rewrite open configuration $\mathcal{C}$ with type $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$ to open configuration $\mathcal{C}'$ with type $\Psi; \Delta \Vdash \mathcal{C}' :: \Delta'$. We detail the configuration typing in the next section. Cut allocates a fresh channel x_0 at *generation* 0. This channel is substituted for the channel variable x



Pos:
R
y_α
$\mathbf{msg}(\_\langle y_\alpha\rangle; y_\alpha \leftarrow y_{\alpha+1})$
$y_{\alpha+1}$
S
R'
$y_{\alpha+1}$
S
$\mathsf{R}' = [y_{\alpha+1}/y_\alpha]\,\mathsf{R}$
Neg:
S
$y_{\alpha+1}$
$\mathbf{msg}(\_\langle y_\alpha\rangle; y_{\alpha+1} \leftarrow y_\alpha)$
y_α
R
S
$y_{\alpha+1}$
R'
$\mathsf{R}' = [y_{\alpha+1}/y_\alpha]\,\mathsf{R}$

Fig. 2: Schematic illustration of asynchronous dynamics.

occurring in the process terms P and Q in the post-state. The generation α of a channel y_α is incremented to $\alpha + 1$ whenever a new message is spawned, except for 1 because there is no continuation. Lastly we point out that fwd is not defined for any channels in Δ' because those configurations are considered *poised*, as we discuss in Sect. IV-D.

C. Configuration Typing

We use the judgment $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$ to type an open configuration $\mathcal{C}$. An open configuration consists of an open *forest* of processes $\mathbf{proc}(x[d], P@d_1)$ and messages $\mathbf{msg}(P)$. We refer to the configuration $\mathcal{C}$ as *open* because it is typed relative to free channels in Δ. While our logical relation is phrased in terms of an open *tree* — representing the open program under consideration — typing of an open forest is necessitated by the inductive nature of the below rules. The judgment indicates that $\mathcal{C}$ provides sessions in Δ', using sessions in Δ, and given the security lattice Ψ. Both Δ' and Δ are linear contexts, consisting of a finite set of assumptions of the form $y_i{:}B_i[d'_i]$, where y_i denotes an actual *channel* that has been allocated upon spawning a process. For simplicity, we do not display a channel's generation.

$$
\frac{}{\Psi; x{:}A[d] \Vdash \cdot :: (x{:}A[d])}\ \mathbf{emp}_1
\qquad
\frac{}{\Psi; \cdot \Vdash \cdot :: (\cdot)}\ \mathbf{emp}_2
$$

$$
\frac{\begin{array}{c}\Psi \Vdash d_1 \sqsubseteq d \qquad \forall y{:}B[d'] \in \Delta'_0, \Delta\, (\Psi \Vdash d' \sqsubseteq d)\\ \Psi; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi; \Delta'_0, \Delta \vdash P@d_1 :: (x{:}A[d])\end{array}}{\Psi; \Delta_0, \Delta'_0 \Vdash \mathcal{C}, \mathbf{proc}(x[d], P@d_1) :: (x{:}A[d])}\ \mathbf{proc}
$$

$$
\frac{\begin{array}{c}\forall y{:}B[d'] \in \Delta'_0, \Delta\, (\Psi \Vdash d' \sqsubseteq d)\\ \Psi; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi; \Delta'_0, \Delta \vdash P@d :: (x{:}A[d])\end{array}}{\Psi; \Delta_0, \Delta'_0 \Vdash \mathcal{C}, \mathbf{msg}(P) :: (x{:}A[d])}\ \mathbf{msg}
$$

$$
\frac{\Psi; \Delta_0 \Vdash \mathcal{C} :: \Delta \qquad \Psi; \Delta'_0 \Vdash \mathcal{C}_1 :: x{:}A[d]}{\Psi; \Delta_0, \Delta'_0 \Vdash \mathcal{C}, \mathcal{C}_1 :: \Delta, x{:}A[d]}\ \mathbf{comp}
$$

Rule **comp** types an open forest, singling out the open tree $\mathcal{C}_1$ rooted at x. Rules **proc** and **msg** type open trees, singling out their root process or message, respectively. Both rules include sufficient premises to establish the tree invariant. Unlike processes, messages have no running secrecy associated because their running secrecy is determined by the maximal secrecy of the sender. Sect. VII provides further details. Rules $\mathbf{emp}_1$ and $\mathbf{emp}_2$, finally, type an empty open forest.

$$\begin{array}{lr}
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(y_\alpha \leftarrow x_\beta)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1[x_\beta/y_\alpha]\mathcal{C}_2 \qquad (y_\alpha \notin \Delta') & \mathsf{fwd} \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(x^d \leftarrow P)@d_2;Q@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{proc}(x_0[d],([x_0/x]\,P)@d_2)\mathbf{proc}(y_\alpha[c],([x_0/x]\,Q)@d_1)\mathcal{C}_2 \qquad (x_0\,\mathit{fresh}) & \mathsf{Cut} \\
\mathbf{proc}(y_\alpha[c],(\mathbf{close}\,y_\alpha)@d_1)\,\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathbf{msg}(\mathbf{close}\,y_\alpha)\mathcal{C}_2 & 1 \\
\mathcal{C}_1\mathbf{msg}(\mathbf{close}\,y_\alpha)\mathcal{C}'\mathbf{proc}(x_\beta[c'],(\mathbf{wait}\,y_\alpha;Q)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathcal{C}'\mathbf{proc}(x_\beta[c'],Q@(d_1 \sqcup c))\mathcal{C}_2 & 1 \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],y_\alpha.k;P@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{proc}(y_{\alpha+1}[c],([y_{\alpha+1}/y_\alpha]P)@d_1)\mathbf{msg}(y_\alpha.k;y_\alpha \leftarrow y_{\alpha+1})\mathcal{C}_2 & \oplus \\
\mathcal{C}_1\mathbf{msg}(y_\alpha[c].k;y_\alpha \leftarrow v_\delta))\mathcal{C}'\mathbf{proc}(u_\gamma[c'],\mathbf{case}\,y_\alpha((\ell \Rightarrow P_\ell)_{\ell\in L})@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathcal{C}'\mathbf{proc}(u_\gamma[c'],([v_\delta/y_\alpha]P_k)@(d_1 \sqcup c))\mathcal{C}_2 & \oplus \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(x_\beta.k;P)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{msg}(x_\beta.k;x_{\beta+1} \leftarrow x_\beta)\mathbf{proc}(y_\alpha[c],([x_{\beta+1}/x_\beta]P)@d_1)\mathcal{C}_2 & \& \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(\mathbf{case}\,y_\alpha(\ell \Rightarrow P_\ell)_{\ell\in L})@d_1)\mathcal{C}'\mathbf{msg}(y_\alpha.k;v_\delta \leftarrow y_\alpha)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{proc}(v_\delta[c],([v_\delta/y_\alpha]P_k)@c)\mathcal{C}'\mathcal{C}_2 & \& \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(\mathbf{send}\,x_\beta\,y_\alpha;P)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{proc}(y_{\alpha+1}[c],([y_{\alpha+1}/y_\alpha]P)@d_1)\mathbf{msg}(\mathbf{send}\,x_\beta\,y_\alpha;y_\alpha \leftarrow y_{\alpha+1})\mathcal{C}_2 & \otimes \\
\mathcal{C}_1\mathbf{msg}(\mathbf{send}\,x_\beta\,y_\alpha;y_\alpha \leftarrow v_\delta)\mathcal{C}'\mathbf{proc}(u_\gamma[c'],(w_\eta \leftarrow \mathbf{recv}\,y_\alpha;P)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathcal{C}'\mathbf{proc}(u_\gamma[c'],([x_\beta/w_\eta][v_\delta/y_\alpha]P)@(d_1 \sqcup c))\mathcal{C}_2 & \otimes \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(\mathbf{send}\,x_\beta\,u_\gamma;P)@d_1)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{msg}(\mathbf{send}\,x_\beta\,u_\gamma;u_{\gamma+1} \leftarrow u_\gamma)\mathbf{proc}(y_\alpha[c],([u_{\gamma+1}/u_\gamma]P)@d_1)\mathcal{C}_2 & \multimap \\
\mathcal{C}_1\mathbf{proc}(y_\alpha[c],(w_\eta \leftarrow \mathbf{recv}\,y_\alpha;P)@d_1)\mathcal{C}'\mathbf{msg}(\mathbf{send}\,x_\beta\,y_\alpha;v_\delta \leftarrow y_\alpha)\mathcal{C}_2 \;\mapsto_{\Delta\Vdash\Delta'}\; \mathcal{C}_1\mathbf{proc}(v_\delta[c],([x_\beta/w_\eta][v_\delta/y_\alpha]P)@c)\mathcal{C}'\mathcal{C}_2 & \multimap
\end{array}$$

Fig. 3: Asynchronous dynamics of $\mathsf{SILL}_{\mathsf{sec}}$.

D. *Poised Configuration*

What are values in a functional setting are *poised* configurations here. Prior work [47] has defined that notion only for closed configurations, and we generalize it to open configurations. An open configuration $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$ is poised, iff it is empty or none of its processes and messages can communicate with each other and there exists at least one process or message that attempts to communicate along a channel in Δ or Δ'. The formal definition of poised configurations is given below. For more details see Definition A.4 in [45].

Definition IV.1 (Poised Configuration)**.** *A configuration $\Delta_1, \Delta_2 \Vdash \mathcal{C}_1, \mathcal{C}_2 :: \Lambda, w{:}A'[c]$ is poised iff either $\mathcal{C}_1, \mathcal{C}_2$ is empty or $\Delta_1 \Vdash \mathcal{C}_1 :: \Lambda$ is poised and $\Delta_2 \Vdash \mathcal{C}_2 :: w{:}A'[c]$ is poised. The configuration $\Delta_2 \Vdash \mathcal{C}_2 :: w{:}A'[c]$ is poised iff it cannot take any steps and at least one of the following conditions hold:*

- *$\mathcal{C}_2$ is an empty configuration.*
- *$\mathcal{C}_2 = \mathcal{C}'_2\mathbf{msg}(P)\mathcal{C}''_2$ such that $\mathbf{msg}(P)$ is a negative message along $y \in \Delta_2$, i.e. $y{:}\&\{\ell{:}A_\ell\}_{\ell\in L}[c_1] \Vdash \mathbf{msg}(P) :: x{:}A_k[c_1]$ or $y{:}A \multimap B[c_1], z{:}A[c_1] \Vdash \mathbf{msg}(P) :: x{:}B[c_1]$, and both subconfigurations $\mathcal{C}'_2$ and $\mathcal{C}''_2$ are poised.*
- *$\mathcal{C}_2 = \mathbf{proc}(x[c'], P@d_1)\,\mathcal{C}'_2$ such that $\mathbf{proc}(x[c'], P@d_1)$ attempts to receive along a channel $y{\in}\Delta_2$.*
- *$\mathcal{C}_2 = \mathcal{C}'_2\mathbf{msg}(P)$ such that $\mathbf{msg}(P)$ is a positive message sent along $w{:}A'[c]$, i.e. $x{:}A_k[c] \Vdash \mathbf{msg}(P) :: w{:} \oplus \{\ell{:}A_\ell\}_{\ell\in L}[c]$ or $x{:}B[c], z_\gamma{:}A[c] \Vdash \mathbf{msg}(P) :: w{:}A \otimes B[c]$,or $\cdot \Vdash \mathbf{msg}(P) :: w{:}1[c]$, and subconfiguration $\mathcal{C}'_2$ is poised.*
- *$\mathcal{C}_2 = \mathbf{proc}(w[c], P@d_1)\,\mathcal{C}'_2$ such that $\mathbf{proc}(w[c], P@d_1)$ attempts to receive along $w{:}A'[c]$.*
- *$\mathcal{C}_2 = \mathcal{C}'_2\mathbf{proc}(w^c \leftarrow x^c@d_1)\,\mathcal{C}''_2$.*

V. Key Ideas - Part II

This section develops the main ideas underlying the session logical relation used to prove noninterference of $\mathsf{SILL}_{\mathsf{sec}}$. The next section puts these ideas into action.

Noninterference essentially amounts to a *program equivalence up to* the secrecy level ξ of the observer, requiring that two runs of a program may only differ in outputs whose secrecy level is above or incomparable to ξ. The fundamental property of the logical relation for noninterference then is stated for two runs of any open program, showing that the runs are related, if given related inputs.

In a session-typed setting, open programs amount to *open trees* and outputs to *messages sent* from that open tree. Inputs, on the other hand, consist of the *messages received* from any *closing configurations*.

Given these basic correspondences, we can develop our session logical relation for noninterference schematically based on Fig. 4. The figure shows two runs $\mathcal{D}_1$ and $\mathcal{D}_2$ of an open program with closing substitutions $\mathcal{C}_1$, $\mathcal{F}_1$ and $\mathcal{C}_2$, $\mathcal{F}_2$, respectively, and post-states $\mathcal{C}'_1\mathcal{D}'_1\mathcal{F}'_1$ and $\mathcal{C}'_2\mathcal{D}'_2\mathcal{F}'_2$, resulting from a message exchange. The session logical relation now mandates that $\mathcal{D}_1$ and $\mathcal{D}_2$ will send the same messages to $\mathcal{C}_1$, $\mathcal{F}_1$ and $\mathcal{C}_2$, $\mathcal{F}_2$, respectively, provided that $\mathcal{C}_1$, $\mathcal{F}_1$ and $\mathcal{C}_2$, $\mathcal{F}_2$ will send the same messages to $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. This property is expressed as

$$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathcal{V}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!]$$

where Δ amounts to the typing of channels connecting $\mathcal{C}_1$ and $\mathcal{C}_2$ with $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively, and K to the typing of the channel connecting $\mathcal{D}_1$ and $\mathcal{D}_2$ with $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. We refer to Δ and K as the *interface* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

Clearly, the above property can only hold for exchanged messages of at most the observer's secrecy level. We call such messages and their carrying channels *observable*. We thus phrase the logical relation only over observable channels, requiring us to determine $\mathcal{D}_1$ and $\mathcal{D}_2$ for two runs $\mathcal{D}_3$ and $\mathcal{D}_4$ of an open program, with $\Psi; \Delta_3 \Vdash \mathcal{D}_3 :: K_3$ and $\Psi; \Delta_4 \Vdash \mathcal{D}_4 :: K_4$, such that the observable channels defined by the projection $\_ \Downarrow \xi$ are the same, i.e., $\Delta_3 \Downarrow \xi = \Delta_4 \Downarrow \xi = \Delta$ and $K_3 \Downarrow \xi = K_4 \Downarrow \xi = K$. The left-over, non-observable channels in Δ_3, K_3 and Δ_4, K_4 are closed off and internalized into $\mathcal{D}_3$ and $\mathcal{D}_4$, yielding $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively.

The message exchange depicted in Fig. 4 is a send, denoted by the red node in $\mathcal{D}_1$ and $\mathcal{D}_2$. The node is a message $\mathbf{msg}(P)$, and the figure captures the positive case depicted in Fig. 2. In the post-states $\mathcal{C}'_1\mathcal{D}'_1\mathcal{F}'_1$ and $\mathcal{C}'_2\mathcal{D}'_2\mathcal{F}'_2$, this message is simply pushed into the substitutions $\mathcal{F}'_1$ and $\mathcal{F}'_2$. The value interpretation of $(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2)$ is now phrased in terms of the transition, requiring that

$$\begin{aligned}
&(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathcal{V}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!], \text{ if} \\
&\qquad (\mathcal{C}'_1\mathcal{D}'_1\mathcal{F}'_1, \mathcal{C}'_2\mathcal{D}'_2\mathcal{F}'_2) \in \mathcal{E}^{\xi}_{\Psi}[\![\Delta' \Vdash K']\!]
\end{aligned}$$

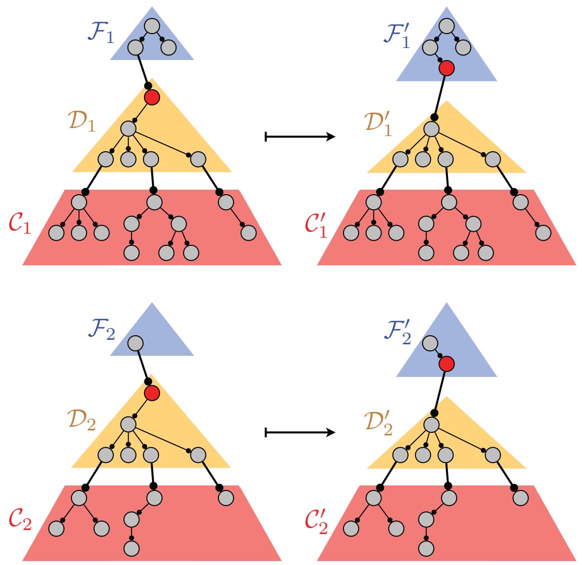


Fig. 4: Session logical relation for noninterference: key ideas.

The post-states $\mathcal{C}'_1\mathcal{D}'_1\mathcal{F}'_1$ and $\mathcal{C}'_2\mathcal{D}'_2\mathcal{F}'_2$ now take any number of internal transitions until $\mathcal{C}''_1$, $\mathcal{D}''_1$, $\mathcal{F}''_1$, $\mathcal{C}''_2$, $\mathcal{D}''_2$, and $\mathcal{F}''_2$ are each individually poised, demanding a message exchange along an observable channel. We thus require this poised configuration to be in the value interpretation

$$(\mathcal{C}''_1, \mathcal{D}''_1, \mathcal{F}''_1; \mathcal{C}''_2, \mathcal{D}''_2, \mathcal{F}''_2) \in \mathcal{V}^{\xi}_{\Psi}[\![\Delta' \Vdash K']\!]$$

The choice to simply push a message $\mathbf{msg}(P)$ across the interface to the recipient, rather then consuming it with a corresponding receiving action, allows for more runs to be soundly equated. In particular, two runs are allowed to differ in the order in which the messages $\mathbf{msg}(P)$ are consumed by the recipients, whenever typing ensures that the recipients can no longer send back any messages to the senders.

Like any logical relation, our session logical relation accounts for the *polarity* of the connectives in Δ and K. Moreover, it considers whether the message is being sent along a channel in Δ or in K. In case the message is being sent along a channel in Δ, we refer to it as communicating on the *left*, otherwise, as communicating on the *right*. These two dimensions span the space of value interpretations of two program runs, requiring *positive* connectives to *assert* the sending of the same message in both runs when communicating on the *right* and to *assume* their existence when communicating on the *left*. Conversely, *negative* connectives can *assume* that the same messages are being sent in both runs when communicating on the *right* and must *assert* sending of the same message in both runs when communicating on the *left*.

VI. Noninterference Logical Relation

In this section we formalize the session logical relation for noninterference as explained in Sect. V. We are interested in a property that asserts that two open programs send the same messages along their observable channels if being closed with any well-typed configurations. The closing configurations are assumed to send the same messages along the observable channels. For this property to hold, the open programs must agree on their set of observable channels and the closing configuration have to be well-typed a priori.

For an open program $\Psi; \Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1[c_1]$ we need two closing configurations. One to provide Δ_1 without using any resources, i.e., $\Psi; \cdot \Vdash \mathcal{C}_1 :: \Delta_1$. The other to use $x_\alpha{:}A_1[c_1]$ as a resource and offer a terminating type, i.e., $\Psi; x_\alpha{:}A_1[c_1] \Vdash \mathcal{F}_1 :: y_\alpha{:}1[c']$. The name and secrecy of the channel provided by $\mathcal{F}_1$ is not significant in our setting; y_α can only send a closing message when all observable channels are already closed. Thus we disregard it and alternatively write $\Psi; x_\alpha{:}A_1[c_1] \Vdash \mathcal{F}_1 :: \cdot$. We keep in mind that a providing type $\cdot$ behaves as a terminating channel. In this paper, we often use $K := x_\alpha[c]{:}A \mid \cdot$ for the providing channel to account for this notation.

Our property of interest is formalized in Def. VI.1.

Definition VI.1 (Equivalence up to Observable Messages)**.** $(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1[c_1]) \equiv^{\Psi}_{\xi} (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2[c_2])$ *is defined as* $\Psi; \Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha{:}A_1[c_1]$ *and* $\Psi; \Delta_2 \Vdash \mathcal{D}_2 :: y_\beta{:}A_2[c_2]$ *and* $\Delta_1 \Downarrow \xi = \Delta_2 \Downarrow \xi = \Delta$ and $x_\alpha{:}A_1[c_1] \Downarrow \xi = y_\beta{:}A_2[c_2] \Downarrow \xi = K$ *and for all* $\mathcal{C}_1, \mathcal{C}_2, \mathcal{F}_1, \mathcal{F}_2$, *with* $\Psi; \cdot \Vdash \mathcal{C}_1 :: \Delta_1$ *and* $\Psi; \cdot \Vdash \mathcal{C}_2 :: \Delta_2$ *and* $\Psi; x_\alpha{:}A_1[c_1] \Vdash \mathcal{F}_1 :: \cdot$ *and* $\Psi; y_\beta{:}A_2[c_2] \Vdash \mathcal{F}_2 :: \cdot$, *we have*

$$(\mathcal{C}_1\mathcal{D}_1\mathcal{F}_1, \mathcal{C}_2\mathcal{D}_2\mathcal{F}_2) \in \mathcal{E}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!].$$

The relation $(\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!]$ is defined in Fig. 5 [2], line 14. It is an apparatus to track the computation of two closed configurations $\mathcal{B}_1$ and $\mathcal{B}_2$ looking for messages being sent and received along their mutual set of observable channels Δ and K. The content of messages sent or received along other channels are not significant and disregarded. In particular, if the offering channels of the two open programs are not observable, we dismiss them from consideration and put $K = \cdot$ as a placeholder in the relation.

To track the observable messages using $\mathcal{E}^{\xi}_{\Psi}$, we need to know that $\mathcal{B}_i$ can be broken down into $\mathcal{C}_i\mathcal{D}_i\mathcal{F}_i$ such that $\Psi; \cdot \Vdash \mathcal{C}_i :: \Delta$, and $\Psi; \Delta \Vdash \mathcal{D}_i :: K$, and $\Psi; K \Vdash \mathcal{F}_i :: \cdot$. We prove that this property holds for any $\mathcal{B}_i$ that is built by closing an open program with observable channels Δ on the left and K on the right. The interested reader can refer to Lemma A.9 and Figure 6 in [45] for further details. The key idea is to internalize any trees rooted at non-observable channels in the bottom closing configuration and the closing non-observable tree constituting the top closing configuration.

After decomposing the configurations $\mathcal{B}_1$ and $\mathcal{B}_2$ into $\mathcal{C}_1\mathcal{D}_1\mathcal{F}_1$ and $\mathcal{C}_2\mathcal{D}_2\mathcal{F}_2$, respectively, we compute each subconfiguration separately. We write $\mathcal{C}, \mathcal{D}, \mathcal{F} \mapsto_{\Delta \Vdash K} \mathcal{C}', \mathcal{D}', \mathcal{F}'$ if (i) $\mathcal{C} \mapsto_{\cdot \Vdash \Delta} \mathcal{C}'$, (ii) $\mathcal{D} \mapsto_{\Delta \Vdash K} \mathcal{D}'$, and (iii) $\mathcal{F} \mapsto_{K \Vdash \cdot} \mathcal{F}'$. We are interested in the state in which none of the subconfigurations can proceed without communicating along an observable channel. This state is closely related to the property of being poised introduced in Sect. IV. We call

[2] For ease of reference in our proofs, we annotate channel names appearing in process terms with their generations (subscript) and maximal secrecy (superscript).

(1)	$(\cdot, \mathcal{D}_1, \mathcal{F}_1; \cdot, \mathcal{D}_1, \mathcal{F}_1) \in \mathcal{V}^\xi_\Psi\llbracket \cdot \Vdash y_\alpha{:}1[c]\rrbracket$	$(\cdot, \mathcal{D}_1, \mathcal{F}_1; \cdot, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\cdot \Vdash y_\alpha : 1[c])$ and $\mathcal{D}_1 = \mathbf{msg}(\mathbf{close}\, y^c_\alpha)$ and $\mathcal{D}_2 = \mathbf{msg}(\mathbf{close}\, y^c_\alpha)$
(2)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket(\Delta \Vdash y_\alpha{:} \oplus \{\ell{:}A_\ell\}_{\ell\in I}[c])\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash y_\alpha : \oplus\{\ell{:}A_\ell\}_{\ell\in I}[c])$ and $\mathcal{D}_1 = \mathcal{D}'_1\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)$ and $\mathcal{D}_2 = \mathcal{D}'_2\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)$ and $(\mathcal{C}_1\mathcal{D}'_1\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_1, \mathcal{C}_2\mathcal{D}'_2\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta \Vdash u_\delta{:}A_k[c]\rrbracket$
(3)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta \Vdash y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}[c]\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}[c])$, and if $(\mathcal{F}_1 = \mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_1$ and $\mathcal{F}_2 = \mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_2)$ then $(\mathcal{C}_1\mathcal{D}_1\mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_1, \mathcal{C}_2\mathcal{D}_2\mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta \Vdash u_\delta{:}A_k[c]\rrbracket$
(4)	$(\mathcal{C}'_1\mathcal{C}''_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}'_2\mathcal{C}''_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta', \Delta'' \Vdash y_\alpha{:}A \otimes B[c]\rrbracket$	$(\mathcal{C}'_1\mathcal{C}''_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}'_2\mathcal{C}''_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash y_\alpha{:}A \otimes B[c])$ and $\mathcal{D}_1 = \mathcal{D}'_1\mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)$ and $\mathcal{D}_2 = \mathcal{D}'_2\mathcal{T}_2\mathbf{msg}(\mathbf{send}, x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)$ and $(\mathcal{C}''_1\mathcal{T}_1\mathcal{C}'_1\mathcal{D}'_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_1,$ $\mathcal{C}''_2\mathcal{T}_2\mathcal{C}'_2\mathcal{D}'_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta'' \Vdash x_\beta{:}A[c]\rrbracket$ and $(\mathcal{C}'_1\mathcal{D}'_1\mathcal{C}''_1\mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_1,$ $\mathcal{C}'_2\mathcal{D}'_2\mathcal{C}''_2\mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta' \Vdash u_\delta{:}B[c]\rrbracket$
(5)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta \Vdash y_\alpha{:}A \multimap B[c]\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash y_\alpha{:}A \multimap B[c])$ and if $(\mathcal{F}_1 = \mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_1$ and $\mathcal{F}_2 = \mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_2)$ then $(\mathcal{C}_1\mathcal{T}_1\mathcal{D}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_1,$ $\mathcal{C}_2\mathcal{T}_2\mathcal{D}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{F}'_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta, x_\beta{:}A[c] \Vdash u_\delta{:}B[c]\rrbracket$
(6)	$(\mathcal{C}_1, \mathcal{F}_1, \mathcal{D}_1; \mathcal{C}_2, \mathcal{F}_2, \mathcal{D}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta, y_\alpha{:}1[c] \Vdash K\rrbracket$	$(\mathcal{C}_1, \mathcal{F}_1, \mathcal{D}_1; \mathcal{C}_2, \mathcal{F}_2, \mathcal{D}_2) \in \mathsf{Tree}_\Psi(\Delta, y_\alpha{:}1[c] \Vdash K)$ and if $(\mathcal{C}_1 = \mathcal{C}'_1\mathbf{msg}(\mathbf{close}\, y^c_\alpha)$ and $\mathcal{C}_2 = \mathcal{C}'_2\mathbf{msg}(\mathbf{close}\, y^c_\alpha))$ then $(\mathcal{C}'_1\mathbf{msg}(\mathbf{close}\, y^c_\alpha)\mathcal{D}_1\mathcal{F}_1, \mathcal{C}'_2\mathbf{msg}(\mathbf{close}\, y^c_\alpha)\mathcal{D}_2\mathcal{F}_2)\mathcal{E}^\xi_\Psi\llbracket\Delta \Vdash K\rrbracket$
(7)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta, y_\alpha : \oplus\{\ell{:}A_\ell\}_{\ell\in I}[c] \Vdash K\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta, y_\alpha{:} \oplus \{\ell{:}A_\ell\}_{\ell\in I}[c] \Vdash K)$ and if $(\mathcal{C}_1 = \mathcal{C}'_1\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)$ and $\mathcal{C}_2 = \mathcal{C}'_2\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta))$ then $(\mathcal{C}'_1\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)\mathcal{D}_1\mathcal{F}_1, \mathcal{C}'_2\mathbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)\mathcal{D}_2\mathcal{F}_2)$ $\in \mathcal{E}^\xi_\Psi\llbracket\Delta, u_\delta{:}A_k[c] \Vdash K\rrbracket$
(8)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta, y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}[c] \Vdash K\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta, y_\alpha{:}\&\{\ell{:}A_\ell\}_{\ell\in I}[c] \Vdash K)$ and $\mathcal{D}_1 = \mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}'_1$ and $\mathcal{D}_2 = \mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}'_2$ and $(\mathcal{C}_1\mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}'_1\mathcal{F}_1,$ $\mathcal{C}_2\mathbf{msg}(y^c_\alpha.k; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}'_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta, u_\delta{:}A_k[c] \Vdash K\rrbracket$
(9)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta, y_\alpha{:}A \otimes B[c] \Vdash K\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta, y_\alpha{:}A \otimes B[c] \Vdash K)$ and if $(\mathcal{C}_1 = \mathcal{C}'_1\mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)$ and $\mathcal{C}_2 = \mathcal{C}'_2\mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)$then $(\mathcal{C}'_1\mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{D}_1\mathcal{F}_1,$ $\mathcal{C}'_2\mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; y^c_\alpha \leftarrow u^c_\delta)\mathcal{D}_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta, x_\beta{:}A[c], u_\delta{:}B[c] \Vdash K\rrbracket$
(10)	$(\mathcal{C}'_1\mathcal{C}''_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}'_2\mathcal{C}''_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta', \Delta'', y_\alpha{:}A \multimap B[c] \Vdash K\rrbracket$	$(\mathcal{C}'_1\mathcal{C}''_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}'_2\mathcal{C}''_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta', \Delta'', y_\alpha{:}A \multimap B[c] \Vdash K)$ and $\mathcal{D}_1 = \mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; ; u^c_\delta \leftarrow y^c_\alpha)\ \mathcal{D}''_1$ and $\mathcal{D}_2 = \mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; ; u^c_\delta \leftarrow y^c_\alpha)\ \mathcal{D}''_2$ and $(\mathcal{C}''_1\mathcal{T}_1\mathcal{C}'_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}''_1\mathcal{F}_1,$ $\mathcal{C}''_2\mathcal{T}_2\mathcal{C}'_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}''_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta' \Vdash x_\beta{:}A[c]\rrbracket$ and $(\mathcal{C}'_1\mathcal{C}''_1\mathcal{T}_1\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}''_1\mathcal{F}_1,$ $\mathcal{C}'_2\mathcal{C}''_2\mathcal{T}_2\mathbf{msg}(\mathbf{send}x^c_\beta\, y^c_\alpha; u^c_\delta \leftarrow y^c_\alpha)\mathcal{D}''_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta'', u_\delta{:}B[c] \Vdash K\rrbracket$
(11)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta \Vdash y_\alpha{:}A[c]\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta', y_\alpha{:}A[c] \Vdash K)$ and $\mathcal{D}_1 = \mathcal{D}'_1\mathbf{proc}(y^c_\alpha, y^c_\alpha \leftarrow x^c_\beta@d_1)$ and $\mathcal{D}_2 = \mathcal{D}'_2\mathbf{proc}(y^c_\alpha, y^c_\alpha \leftarrow x^c_\beta@d_2)$ and $([x^c_\beta/y^c_\alpha]\mathcal{C}_1\mathcal{D}'_1\mathcal{F}_1, [x^c_\beta/y^c_\alpha]\mathcal{C}_2\mathcal{D}'_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta \Vdash x_\beta{:}A[c]\rrbracket$
(12)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in$ $\mathcal{V}^\xi_\Psi\llbracket\Delta, y_\alpha{:}A[c] \Vdash K\rrbracket$	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta', y_\alpha{:}A[c] \Vdash K)$ and if $(\mathcal{C}_1 = \mathcal{C}'_1\mathbf{proc}(y^c_\alpha, y^c_\alpha \leftarrow x^c_\beta@d_1)$ and $\mathcal{C}_2 = \mathcal{C}'_2\mathbf{proc}(y^c_\alpha, y^c_\alpha \leftarrow x^c_\beta@d_2))$ then $([x^c_\beta/y^c_\alpha]\mathcal{C}'_1\mathcal{D}_1\mathcal{F}_1, [x^c_\beta/y^c_\alpha]\mathcal{C}'_2\mathcal{D}_2\mathcal{F}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta, x_\beta{:}A[c] \Vdash K\rrbracket$
(13)	$(\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{V}^\xi_\Psi\llbracket\cdot \Vdash \cdot\rrbracket$	$(\mathcal{B}_1, \mathcal{B}_2) \in \mathsf{Tree}_\Psi\llbracket\cdot \Vdash \cdot\rrbracket$
(14)	$(\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}^\xi_\Psi\llbracket\Delta \Vdash K\rrbracket$	$\mathcal{B}_1 = \mathcal{C}_1\mathcal{D}_1\mathcal{F}_1$ and $\mathcal{B}_2 = \mathcal{C}_2\mathcal{D}_2\mathcal{F}_2$ and $(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash K)$, and $\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1 \mapsto^{\text{poised}}_{\Delta\Vdash K} \mathcal{C}'_1, \mathcal{D}'_1, \mathcal{F}'_1$ and $\mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2 \mapsto^{\text{poised}}_{\Delta\Vdash K} \mathcal{C}'_2, \mathcal{D}'_2, \mathcal{F}'_2$ and $(\mathcal{C}'_1, \mathcal{D}'_1, \mathcal{F}'_1; \mathcal{C}'_2, \mathcal{D}'_2, \mathcal{F}'_2) \in \mathcal{V}^\xi_\Psi\llbracket\Delta \Vdash K\rrbracket$.
(15)	$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_\Psi(\Delta \Vdash K)$	$\Psi; \cdot \Vdash \mathcal{C}_1 :: \Delta$, and $\Psi; \cdot \Vdash \mathcal{C}_2 :: \Delta$, $\Psi; \Delta \Vdash \mathcal{D}_1 :: K$ and $\Psi; \Delta \Vdash \mathcal{D}_2 :: K$ and $\Psi; K \Vdash \mathcal{F}_1 :: \cdot$ and $\Psi; K \Vdash \mathcal{F}_2 :: \cdot$

Fig. 5: Session Logical Relation for Noninterference: property (left), condition (right).

$\mathcal{C}_i, \mathcal{D}_i, \mathcal{F}_i$ poised if its subconfigurations $\mathcal{C}_i$, and $\mathcal{D}_i$, and $\mathcal{F}_i$ are poised. We write $\mathcal{C}_i, \mathcal{D}_i, \mathcal{F}_i \mapsto^{\text{poised}}_{\Delta \Vdash K} \mathcal{C}'_i, \mathcal{D}'_i, \mathcal{F}'_i$ stating that $\mathcal{C}_i, \mathcal{D}_i, \mathcal{F}_i \mapsto^{*}_{\Delta \Vdash K} \mathcal{C}'_i, \mathcal{D}'_i, \mathcal{F}'_i$ and $\mathcal{C}'_i, \mathcal{D}'_i, \mathcal{F}'_i$ is poised. ($\mapsto^{*}_{\Delta \Vdash K}$ refers to zero or more steps taken with $\mapsto_{\Delta \Vdash K}$.)

To relate two poised configurations we use the value relation $\mathcal{V}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!]$. This relation *establishes* equality of the content of every message fired from $\mathcal{D}_1$ and $\mathcal{D}_2$ before adding them to the closing configurations $\mathcal{C}_i$ and $\mathcal{F}_i$ (See Fig. 4). In the case of sending higher order channels (lines 4 and 10 in Fig. 5), we further *assure* that the trees sent along the messages are also related and will behave similarly when received by the closing configuration.

For the messages being fired from the poised closing configurations $\mathcal{C}_i$ and $\mathcal{F}_i$, we *assume* that they have the same content ready to be moved to $\mathcal{D}_1$ and $\mathcal{D}_2$. In particular for higher order channels (lines 5 and 9 in Fig. 5) we *assume* the channels sent by the closing configurations will send the same observable messages too. We add the received channels to the set of observable channels to make sure that the open programs do not send them different messages.

A forwarding process does not send or receive an explicit message. However, when process $\mathbf{proc}(y[c], y_\alpha \leftarrow x_\beta)$ forwards channel (x_β) to an observable channel (y_α) the substitution of x_β for y_α amounts to a broadcast of the name of x_β along the observable channel y_α. The channel x_β has a secrecy level lower than or equal to the observer and now can be observed too. In our relation (line 11 of Fig. 5) we *assert* that such forwarding rules in $\mathcal{D}_1$ and $\mathcal{D}_2$ always broadcast the same names. In the dual case (line 12 of Fig. 5) we *assume* that the closing configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ broadcast the same names. In both cases we continue by monitoring the rest of the computation along x_β. The same holds for a forwarding on the tail of a message.

The well-foundedness of our logical relation is based on a lexicographic order on the structure of observable types and a multiset order $<$ on the size of configurations. The order $<$ is a multiset order on finite multiset $\mathcal{M}$ of the process typing judgments $\Psi; \Delta'' \vdash P :: y_\gamma{:}A[d]$ used in the typing derivation of $\mathcal{C}$. Process typing judgments are ordered based on the size of the process term. As a result, well-foundedness of $<$ follows from the well-foundedness of process terms [48].

VII. Metatheory

In this section we establish the main properties of the system. We show that $\mathsf{SILL}_{\mathsf{sec}}$ is a terminating language with the standard preservation and progress properties. More importantly, we prove that it enjoys the noninterference property.

Theorem VII.1 (Preservation)**.** *If $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \mapsto_{\Delta \Vdash \Delta'} \mathcal{C}'$, then $\Psi; \Delta \Vdash \mathcal{C}' :: \Delta'$. Moreover $\mathcal{C}' < \mathcal{C}$ by the multiset ordering.*

Proof. The proof is by case analysis of $\mathcal{C} \mapsto_{\Delta \Vdash \Delta'} \mathcal{C}'$ and inversion on the typing judgment $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$. See Theorem A.5 in [45] for details. □

Theorem VII.2 (Progress)**.** *If $\Psi; \Delta \Vdash \mathcal{C} :: \Delta'$, then either $\mathcal{C} \mapsto_{\Delta \Vdash \Delta'} \mathcal{C}'$ or $\mathcal{C}$ is poised.*

Proof. The proof is by induction on the configuration typing of $\mathcal{C}$. See Theorem A.6 in [45] for details. □

Termination of $\mathsf{SILL}_{\mathsf{sec}}$ follows from Thm. VII.2 and Thm. VII.1 and well-foundedness of the $<$ order.

The fundamental property of our logical relation is noninterference stated as below.

Theorem VII.3 (Noninterference)**.** *For all security levels ξ and configurations $\Psi; \Delta \Vdash \mathcal{D} :: x_\alpha{:}T[c]$, we have*

$$(\Delta \Vdash \mathcal{D} :: x_\alpha{:}T[c]) \equiv^{\Psi}_{\xi} (\Delta \Vdash \mathcal{D} :: x_\alpha{:}T[c]).$$

Our noninterference theorem asserts that a well-typed open configuration $\mathcal{D}$ is equivalent to itself. It states that if we run a program twice but with different closing configurations, the contents of messages sent by the program along the observable channels will be the same in both runs. The assertion is based on the assumption that the closing configurations send the same messages along the observable channels in both runs.

The two runs start out as $\mathcal{D}$, guaranteeing that their tree structure is identical and their processes are running the same code. The two runs continue to be identical until a process in each run receives a message from a closing configuration along a non-observable channel. The received messages may differ in contents because the carrier channel's maximal secrecy is higher than or incomparable to ξ. Based on the contents of the received message, the two runs may choose different continuations, after which they begin to diverge in their tree structure and the code the individual processes are running. On the other hand, the running secrecy of the receiving processes will be adjusted upon receiving to become higher or incomparable to ξ. This adjustment makes sure that the receiving processes can no longer send any messages along channels of lower or equal secrecy than the observer. In particular, they cannot send a message along an observable channel. We call such processes that can no longer affect any observable messages *irrelevant*.

Throughout the computation, the code and structure of some processes may diverge as they receive non-observable messages. However, the *relevant* processes, i.e., the processes that can affect the contents of observable messages, stay identical. Later in this section, we state the fundamental theorem (Thm. VII.6) that proves two configurations to be equivalent up to observable messages if their relevant processes are identical. The noninterference property is then an immediate corollary of the fundamental theorem.

Before stating the fundamental theorem, we need to define the notion of a relevant process. We discussed earlier that a process with running secrecy higher or incomparable to the observer's secrecy level is irrelevant. An irrelevant process can no longer spawn any observable messages. Moreover, if it sends a message along a non-observable channel, then the receiver becomes irrelevant too. There is another form of irrelevant process with running secrecy less than or equal to ξ but with paths to observable channels passing through channels with maximal secrecy level higher than or incomparable to ξ.

These channels block the flow of information because any process receiving along such a channel becomes irrelevant.

To establish a sound definition of relevant processes and messages for the asynchronous semantics, we need a lookahead for the running secrecy. Consider a process $\textbf{case}\, y^c(\cdots)@d_1$ in the open program and its counterpart $\textbf{case}\, y^c(\cdots)@d_1$ in the other run. They both have running secrecy $d_1 \sqsubseteq \xi$, and are ready to receive a label along a non-observable channel $y[c]$. By the previous discussion, right after receiving a label the two processes become irrelevant. The non-observable messages may not be ready at the same time. For example, the process in the first run may receive the message right away and become irrelevant, while the other process may need to wait for a while. This results in a discrepancy between relevant processes in the two runs. However, these processes cannot affect any observable channels even before they receive a channel. Based on their code they can only receive in the current step and right after the receive they become irrelevant. To account for delays in the receives, we label these two processes as irrelevant even before they receive, using a lookahead called *quasi running secrecy*. The quasi running secrecy of a receiving process is defined as its running secrecy at the next step, i.e., right after the receive.

We determine the running secrecy of a message to be the maximal secrecy of the channel that the message is sent along. However, messages are only temporary holders of a label or tree that they transport. Unless a message is observable, i.e., sent to a closing configuration, its contents can only affect an observable channel *after* it is received by a process. The quasi running secrecy of a message accounts for this and reflects the future potential of a message once it is received and is determined by examining the running secrecy of the recipient. In case of a negative message (see Fig. 2), the receiver is a child of the message. By the tree invariant, the running secrecy of the child is less than or equal to the maximal secrecy of the carrier channel. After receiving the message the running secrecy of the receiver will be equal to the maximal secrecy of the carrier channel. As a result, the quasi running secrecy of a negative message amounts to the maximal secrecy of the channel along which the message is sent. In case of a positive message (see Fig. 2), the receiver is the parent of the message. The running secrecy d_1 of the parent may be higher or incomparable to the maximal secrecy c of the carrier channel. After the message is received, the running secrecy of the parent is adjusted to at least $c \sqcup d_1$. As a result, we determine the quasi running secrecy of a positive message to be the running secrecy of its parent after the message has been received $(c \sqcup d_1)$.

The notions of quasi running secrecy and relevancy are formally defined in Def. VII.4 and Def. VII.5

Definition VII.4 (Quasi Running Secrecy)**.** *In the configuration tree, the quasi running secrecy of a message or process is determined based on its running secrecy, its process term, and the running secrecy of its parent.*

- *If the node is a process with a process term other than* $\textbf{recv}$ *or* $\textbf{case}$*, then its quasi running secrecy is equal to its running secrecy.*
- *If the process term is of the form* $\textbf{case}\, y^c_\alpha(\ell \Rightarrow P_\ell)_{\ell \in L}@d_1$ *or* $x^\psi \leftarrow \textbf{recv}\, y^c_\alpha; P_x@d_1$*, then its quasi running secrecy is* $d_1 \sqcup c$*.*
- *If the node is a message of a negative type along channel* y^c_α*, its quasi running secrecy is* c*.*
- *If the node is a message of a positive type along channel* y^c_α *and it has a parent with quasi running secrecy* d_1*, its quasi running secrecy is* $d_1 \sqcup c$*.*

The quasi running secrecy can be determined by traversing the tree top to bottom.

Definition VII.5 (Relevant Channels and Processes)**.** *Consider configuration* $\Delta \Vdash \mathcal{D} :: K$ *and observer level* ξ*. A channel is relevant in* $\mathcal{D}$ *if 1) it is has a maximal secrecy level lower than or equal to* ξ*, and 2) it is either an observable channel or it shares a process or message with quasi running secrecy less than* ξ *with a relevant channel. (A channel shares a process with another channel if they are siblings or one is the parent of another.)*

The set of all relevant channels can be found by traversing the tree bottom-up. If K *is observable, then by the tree invariant, every channel in* $\mathcal{D}$ *will be relevant.*

A relevant process or message has quasi running secrecy less than or equal to ξ *and at least one relevant channel.* $\mathcal{C}{\Downarrow}\xi$ *are the relevant processes and messages in* $\mathcal{C}$*. We write* $\mathcal{C}_1{\Downarrow}\xi =_\xi \mathcal{C}_2{\Downarrow}\xi$ *if they are identical up to renaming of channels with higher or incomparable secrecy than the observer.*

The fundamental theorem is stated as below.

Theorem VII.6 (Fundamental Theorem)**.** *For all security levels* ξ*, and configurations* $\Psi; \Delta_1 \Vdash \mathcal{D}_1 :: u_\alpha{:}A_1[c_1]$ *and* $\Psi; \Delta_2 \Vdash \mathcal{D}_2 :: v_\beta{:}A_2[c_2]$ *with* $\mathcal{D}_1{\Downarrow}\xi = \mathcal{D}_2{\Downarrow}\xi$*,* $\Delta_1 \Downarrow \xi = \Delta_2 \Downarrow \xi$*, and* $u_\alpha{:}A_1[c_1] \Downarrow \xi = v_\beta{:}A_2[c_2] \Downarrow \xi$ *we have*

$$(\Delta_1 \Vdash \mathcal{D}_1 :: u_\alpha{:}A_1[c_1]) \equiv^{\Psi}_{\xi} (\Delta_2 \Vdash \mathcal{D}_2 :: v_\beta{:}A_2[c_2]).$$

Proof. The proof is by induction on the type structure and the multiset ordering. For the details see Theorem A.13 in [45]. □

To prove that our fundamental theorem entails the desired property, we define an alternative stepping definition $\hookrightarrow_{\Delta \Vdash K}$ for a closed configuration $\mathcal{C}, \mathcal{D}, \mathcal{F} \in \mathsf{Tree}(\Delta \Vdash K)$. Where $\mathcal{C}, \mathcal{D}, \mathcal{F} \in \mathsf{Tree}(\Delta \Vdash K)$ is defined as $\cdot \Vdash \mathcal{C} :: \Delta$, and $\Delta \Vdash \mathcal{D} :: K$, and $K \Vdash \mathcal{F} :: y{:}1[c]$. The idea is to run this closed configuration to completion, while accumulating the messages exchanged between the open program $\mathcal{D}$ and closing configurations $\mathcal{C}$ and $\mathcal{F}$ in a queue. To this end, we define $\mathcal{C}, \mathcal{D}, \mathcal{F} \hookrightarrow_{\Delta \Vdash K} \mathsf{queue}$ by first stepping $\mathcal{C}, \mathcal{D}, \mathcal{F}$ in terms of $\mapsto^{\mathsf{poised}}_{\Delta \Vdash K}$ until we get a configuration $\mathcal{C}', \mathcal{D}', \mathcal{F}'$ that is poised (i.e., $\mathcal{C}, \mathcal{D}, \mathcal{F} \mapsto^{\mathsf{poised}}_{\Delta \Vdash K} \mathcal{C}', \mathcal{D}', \mathcal{F}'$) and then append to the queue the message that is waiting to be sent. For example, $\mathcal{C}, \mathcal{D}''\textbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta), \mathcal{F} \hookrightarrow_{\Delta \Vdash y_\alpha:\oplus\{\ell:A_\ell\}_{\ell \in L}[c]} \mathsf{queue}'\textbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)$, where queue' is a recursive invocation (i.e., $\mathcal{C}, \mathcal{D}'', \textbf{msg}(y^c_\alpha.k; y^c_\alpha \leftarrow u^c_\delta)\mathcal{F} \hookrightarrow_{\Delta \Vdash u_\delta:A_k[c]} \mathsf{queue}'$). We use an overline to indicate the direction of a message: $\mathbf{msg}(\_)$ denotes a message sent from $\mathcal{D}$ to $\mathcal{C}$ or $\mathcal{F}$ and $\overline{\mathbf{msg}}(\_)$ denotes a message sent from $\mathcal{C}$ or $\mathcal{F}$ to $\mathcal{D}$. The complete definition of $\hookrightarrow_{\Delta \Vdash K}$ is given in Figure 7 in [45].

It is straightforward to show that $\mathcal{CDF} \mapsto^{*}_{\cdot \Vdash \cdot}$ $\mathbf{msg}(\mathbf{close}\_)$ if and only if for some queue, we have $\mathcal{C}, \mathcal{D}, \mathcal{F} \hookrightarrow_{\Delta \Vdash K}$ queue, where queue is the list of observable messages being exchanged between $\mathcal{D}$ and the closing configurations $\mathcal{C}$ and $\mathcal{F}$ along Δ and K.

Definition VII.7. *Define* $\mathsf{queue}_1 =_{\xi} \mathsf{queue}_2$ *as either*

- $\mathsf{queue}_1 = \mathsf{queue}_2 = \cdot$, *or*
- $\mathsf{queue}_1 = q_1\, \mathsf{queue}_1'$, *and* $\mathsf{queue}_2 = q_2\, \mathsf{queue}_2'$, *and* $q_1 = q_2$, *and* $\mathsf{queue}_1' =_{\xi} \mathsf{queue}_2'$, *or*
- $\mathsf{queue}_1 = \overline{q_1}\, \mathsf{queue}_1'$, *and* $\mathsf{queue}_2 = \overline{q_2}\, \mathsf{queue}_2'$, *and if* $q_1 = q_2$ *then* $\mathsf{queue}_1' =_{\xi} \mathsf{queue}_2'$.

Theorem VII.8. *For* $(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1; \mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \in \mathsf{Tree}_{\Psi}(\Delta \Vdash K)$, *if* $(\mathcal{C}_1\mathcal{D}_1\mathcal{F}_1; \mathcal{C}_2\mathcal{D}_2\mathcal{F}_2) \in \mathcal{E}^{\xi}_{\Psi}[\![\Delta \Vdash K]\!]$, *then*

$$(\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1) \hookrightarrow_{\Delta \Vdash K} \mathsf{queue}_1 \text{ and } (\mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2) \hookrightarrow_{\Delta \Vdash K} \mathsf{queue}_2$$

such that $\mathsf{queue}_1 =_{\xi} \mathsf{queue}_2$.

Proof. The proof is straightforward by matching the cases in the definition of $\mathcal{E}$ with the cases in the definition of $\hookrightarrow_{\Delta \Vdash K}$. □

VIII. Related and Future Work

Related work can be categorized along the following axes:

a) IFC Type Systems for Functional and Imperative Languages: Volpano et al.'s seminal paper [19] has spurred a multitude of work on IFC type systems for sequential programs (c.f., [20]). Our noninterference definition is inspired by Bowman et al.'s work, which also defines noninterference in terms of a logical relation [49], albeit in a sequential context.

b) IFC Type Systems for Process Calculi: IFC type systems have also been explored for process calculi with the goal to prevent information leakage through process communications [21]–[31]. Many of these works associate a security label either with types or channels. Yoshida et al. associate the security label with actions [22]; Hennessy and Riely associate read and write policies with channels [27], [28]; and Crafa et al. associate a security label with the process and capabilities with expressions [23]. In contrast, our system associates two security labels, the running secrecy and the maximal secrecy, with every process. The use of two labels also sets our system apart from prior work in that it is flow-sensitive: the running secrecy increases as the process receives more information. Flow sensitivity of our system can elegantly be accommodated by our tree invariant, given a formulation based on intuitionistic linear logic session types. Some of the existing work also consider declassification [25], [26], which we leave as future work.

Timing channels and race conditions can contribute to information leakage. Unlike prior work [29], [30], our linear types ensure progress, termination, and freedom of race conditions; and therefore do not need additional checks to rule out such leaks. Prior work also proposed different noninterference definitions, relying on barbed-congruence, P-congruence, may-testing and must-testing, per-models, and trace equivalence. Our noninterference definition is based on a novel binary session logical relation. It is closest to barbed-congruence definitions and entails trace equivalences. Since our processes' behavior is finite, we do not need co-inductive definitions.

c) IFC for Multiparty Session Types: Only recently, have researchers investigated incorporating information flow security into session types [36], [37], [50], [51]. In addition to developing information flow session type systems that allow declassification [36], [37], researchers also designed flexible run-time monitoring techniques for preventing information leakage [50], [51], all in the context of multiparty session types. Ours is the first information flow binary session type system. Again, our flow-sensitive type system and logical relation-based definition for noninterference sets us apart from existing work.

d) Hybrid Logic Modal Worlds in Session Types: Our typing judgment includes world modalities from hybrid logic as syntactic objects in propositions, where worlds amount to secrecy levels. A hybrid logic approach has been used in prior work on binary session types to ensure deadlock-freedom of shared binary session types [52] and accessibility in linear binary session types [53]. Our work differs not only in the established property of interest (noninterference) but also in the use of a novel binary relation for session types. Possible worlds have also been used by Brookes [54] to give a denotational semantics of parallel Algol with shared variables, allowing reasoning about safety and liveness properties of parallel programs.

e) Logical Relations for Session Types: The application of logical relations to session types has focused predominantly on unary logical relations (predicates) for proving termination [41]–[43] with the exception of a binary logical relation for parametricity [44]. Noninterference, however, demands a more nuanced binary relation, requiring observing a process' communication not only with its client but also with all the processes it is a client of. Our work generalizes binary logical relations for session typed languages to support *open configurations*, considering both the antecedent and succedent of the typing judgment. Interestingly, Kavanagh [55] makes a similar generalization, albeit in the context of bisimulations and barbed congruences. While we have defined the logical relation for noninterference, we believe that the technical developments in this paper can serve as a stepping stone for future explorations.

f) Kripke Logical Relations: Conceptually, our work seems related to Kripke logical relations [56] and in particular the works that use possible worlds [57] and state machines [58] to impose invariants on program heaps. In our setting, the program heap is a configuration of processes. Session types constrain how the configuration can evolve, and configuration typing asserts that configurations align with the security lattice. It seems that our secrecy-level-enriched session types internalize Kripke logical worlds into the type system. We would like to explore this connection in future work.

References

[1] K. Honda, “Types for dyadic interaction,” in *4th International Conference on Concurrency Theory (CONCUR)*, ser. Lecture Notes in Computer Science, vol. 715. Springer, 1993, pp. 509–523. [Online]. Available: https://doi.org/10.1007/3-540-57208-2_35

[2] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *7th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 1381. Springer, 1998, pp. 122–138. [Online]. Available: https://doi.org/10.1007/BFb0053567

[3] B. Toninho, L. Caires, and F. Pfenning, “Higher-order processes, functions, and sessions: A monadic integration,” in *22nd European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 7792. Springer, 2013, pp. 350–369. [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6_20

[4] B. Toninho, “A logical foundation for session-based concurrent computation,” Ph.D. dissertation, Carnegie Mellon University and New University of Lisbon, 2015.

[5] S. Balzer and F. Pfenning, “Manifest sharing with session types,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 37:1–37:29, 2017. [Online]. Available: https://doi.org/10.1145/3110281

[6] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou, “Session types for object-oriented languages,” in *20th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 4067. Springer, 2006, pp. 328–352. [Online]. Available: https://doi.org/10.1007/11785477_20

[7] R. Pucella and J. A. Tov, “Haskell session types with (almost) no class,” in *1st ACM SIGPLAN Symposium on Haskell (Haskell)*. ACM, 2008, pp. 25–36. [Online]. Available: https://doi.org/10.1145/1411286.1411290

[8] K. Imai, S. Yuen, and K. Agusa, “Session type inference in haskell,” in *3rd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, ser. EPTCS, vol. 69, 2010, pp. 74–91. [Online]. Available: https://doi.org/10.4204/EPTCS.69.6

[9] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, “Session types for Rust,” in *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*. ACM, 2015, pp. 13–22. [Online]. Available: https://doi.org/10.1145/2808098.2808100

[10] S. Lindley and J. G. Morris, “Embedding session types in Haskell,” in *9th International Symposium on Haskell (Haskell)*. ACM, 2016, pp. 133–145. [Online]. Available: https://doi.org/10.1145/2976002.2976018

[11] A. Scalas and N. Yoshida, “Lightweight session programming in Scala,” in *30th European Conference on Object-Oriented Programming (ECOOP)*, ser. LIPIcs, no. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 21:1–21:28. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[12] L. Padovani, “A simple library implementation of binary sessions,” *Journal of Functional Programming*, vol. 27, p. e4, 2017. [Online]. Available: https://doi.org/10.1017/S0956796816000289

[13] K. Imai, N. Yoshida, and S. Yuen, “Session-ocaml: A session-based library with polarities and lenses,” *Science of Computer Programming*, vol. 172, pp. 135–159, 2019. [Online]. Available: https://doi.org/10.1016/j.scico.2018.08.005

[14] W. Kokke, “Rusty variation: Deadlock-free sessions with failure in Rust,” in *12th Interaction and Concurrency Experience (ICE)*, ser. EPTCS, vol. 304, 2019, pp. 48–60. [Online]. Available: https://doi.org/10.4204/EPTCS.304.4

[15] R. Chen and S. Balzer, “Ferrite: A judgmental embedding of session types in Rust,” *CoRR*, vol. abs/2009.13619, 2020. [Online]. Available: https://arxiv.org/abs/2009.13619

[16] L. Caires and F. Pfenning, “Session types as intuitionistic linear propositions,” in *21th International Conference onf Concurrency Theory (CONCUR)*, ser. Lecture Notes in Computer Science, vol. 6269. Springer, 2010, pp. 222–236. [Online]. Available: https://doi.org/10.1007/978-3-642-15375-4_16

[17] P. Wadler, “Propositions as sessions,” in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2012, pp. 273–286. [Online]. Available: https://doi.org/10.1145/2364527.2364568

[18] W. Kokke, F. Montesi, and M. Peressotti, “Better late than never: A fully-abstract semantics for classical processes,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 24:1–24:29, 2019. [Online]. Available: https://doi.org/10.1145/3290337

[19] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *J. Comput. Secur.*, vol. 4, no. 2–3, p. 167–187, Jan. 1996.

[20] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, p. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[21] K. Honda, V. T. Vasconcelos, and N. Yoshida, “Secure information flow as typed process behaviour,” in *9th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 1782. Springer, 2000, pp. 180–199. [Online]. Available: https://doi.org/10.1007/3-540-46425-5_12

[22] K. Honda and N. Yoshida, “A uniform type structure for secure information flow,” in *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2002, pp. 81–92. [Online]. Available: https://doi.org/10.1145/503272.503281

[23] S. Crafa, M. Bugliesi, and G. Castagna, “Information flow security for boxed ambients,” *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 3, pp. 76–97, 2002. [Online]. Available: https://doi.org/10.1016/S1571-0661(04)80417-1

[24] S. Crafa and S. Rossi, “A theory of noninterference for the π-calculus,” in *International Symposium on Trustworthy Global Computing (TGC)*, ser. Lecture Notes in Computer Science, vol. 3705. Springer, 2005, pp. 2–18. [Online]. Available: https://doi.org/10.1007/11580850_2

[25] ——, “P-congruences as non-interference for the pi-calculus,” in *ACM Workshop on Formal Methods in Security Engineering (FMSE)*. ACM, 2006, pp. 13–22. [Online]. Available: https://doi.org/10.1145/1180337.1180339

[26] ——, “Controlling information release in the π-calculus,” *Information and Computation*, vol. 205, no. 8, pp. 1235 – 1273, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089054010700003X

[27] M. Hennessy and J. Riely, “Information flow vs. resource access in the asynchronous pi-calculus,” *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 5, p. 566–591, Sep. 2002. [Online]. Available: https://doi.org/10.1145/570886.570890

[28] M. Hennessy, “The security pi-calculus and non-interference,” *The Journal of Logic and Algebraic Programming*, vol. 63, no. 1, pp. 3 – 34, 2005, special issue on The pi-calculus. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832604000049

[29] N. Kobayashi, “Type-based information flow analysis for the pi-calculus,” *Acta Inf.*, vol. 42, no. 4, p. 291–347, Dec. 2005.

[30] S. Zdancewic and A. C. Myers, “Observational determinism for concurrent program security,” in *16th IEEE Computer Security Foundations Workshop (CSFW)*, 2003, pp. 29–43.

[31] F. Pottier, “A simple view of type-secure information flow in the π-calculus,” in *Proceedings 15th IEEE Computer Security Foundations Workshop (CSFW-15)*, 2002, pp. 320–330.

[32] D. Stefan, E. Z. Yang, B. Karp, P. Marchenko, A. Russo, and D. Mazières, “Protecting users by confining JavaScript with COWL,” in *Proc. OSDI*, 2014.

[33] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, “Run-time monitoring and formal analysis of information flows in chromium,” in *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[34] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, “Run-time enforcement of information-flow properties on android (extended abstract),” in *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[35] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, “Information flow control for standard OS abstractions,” in *Proc. SOSP*, 2007.

[36] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk, “Session types for access and information flow control,” in *21th International Conference on Concurrency Theory (CONCUR)*, 2010, pp. 237–252. [Online]. Available: https://doi.org/10.1007/978-3-642-15375-4_17

[37] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini, “Typing access control and secure information flow in sessions,” *Information and Computation*, vol. 238, pp. 68–105, 2014. [Online]. Available: https://doi.org/10.1016/j.ic.2014.07.005

[38] L. Caires, F. Pfenning, and B. Toninho, “Linear logic propositions as session types,” *Mathematical Structures in Computer Science*,

vol. 26, no. 3, pp. 367–423, 2016. [Online]. Available: https://doi.org/10.1017/S0960129514000218

[39] W. W. Tait, "Intensional interpretations of functionals of finite type I," *The Journal of Symbolic Logic*, vol. 32, no. 2, pp. 198–212, 1967. [Online]. Available: http://www.jstor.org/stable/2271658

[40] R. Statman, "Logical relations and the typed λ-calculus," *Information and Control*, vol. 65, no. 2/3, pp. 85–97, 1985. [Online]. Available: https://doi.org/10.1016/S0019-9958(85)80001-2

[41] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho, "Linear logical relations for session-based concurrency," in *21st European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 7211. Springer, 2012, pp. 539–558. [Online]. Available: https://doi.org/10.1007/978-3-642-28869-2_27

[42] ——, "Linear logical relations and observational equivalences for session-based concurrency," *Information and Computation*, vol. 239, pp. 254–302, 2014. [Online]. Available: https://doi.org/10.1016/j.ic.2014.08.001

[43] H. DeYoung, F. Pfenning, and K. Pruiksma, "Semi-axiomatic sequent calculus," in *5th International Conference on Formal Structures for Computation and Deduction (FSCD)*, ser. LIPIcs, vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 29:1–29:22. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSCD.2020.29

[44] L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho, "Behavioral polymorphism and parametricity in session-based communication," in *22nd European Symposium on Programming (ESOP)*, 2013, pp. 330–349. [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6_19

[45] F. Derakhshan, S. Balzer, and L. Jia, "Session logical relations for noninterference," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2104.14094

[46] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar, "Resource-aware session types for digital contracts," in *34th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2021.

[47] F. Pfenning and D. Griffith, "Polarized substructural session types," in *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, ser. Lecture Notes in Computer Science, vol. 9034. Springer, 2015, pp. 3–22. [Online]. Available:

[57] A. Ahmed, D. Dreyer, and A. Rossberg, "State-dependent representation independence," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2009, pp. 340–353. [Online]. Available: https://doi.org/10.1145/1480881.1480925

https://doi.org/10.1007/978-3-662-46678-0_1

[48] J.-P. Jouannaud and H. Kirchner, "Completion of a set of rules modulo a set of equations," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 1155–1194, 1986.

[49] W. J. Bowman and A. Ahmed, "Noninterference for free," in *20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2015, pp. 101–113. [Online]. Available: https://doi.org/10.1145/2784731.2784733

[50] I. Castellani, M. Dezani-Ciancaglini, and J. A. Pérez, "Self-adaptation and secure information flow in multiparty communications," *Formal Aspects of Computing*, vol. 28, no. 4, pp. 669–696, 2016. [Online]. Available: https://doi.org/10.1007/s00165-016-0381-3

[51] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini, "Information flow safety in multiparty sessions," *Mathematical Structures in Computer Science*, vol. 26, no. 8, p. 1352–1394, 2016.

[52] S. Balzer, B. Toninho, and F. Pfenning, "Manifest deadlock-freedom for shared session types," in *28th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 11423. Springer, 2019, pp. 611–639. [Online]. Available: https://doi.org/10.1007/978-3-030-17184-1_22

[53] L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho, "Domain-aware session types," in *30th International Conference on Concurrency Theory (CONCUR)*, ser. LIPIcs, vol. 140. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 39:1–39:17. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2019.39

[54] S. D. Brookes, "The essence of parallel algol," *Information and Computation*, vol. 179, no. 1, pp. 118–149, 2002. [Online]. Available: https://doi.org/10.1006/inco.2002.2995

[55] R. Kavanagh, "Fairness and observed communication semantics for session-typed languages," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2104.01065

[56] A. M. Pitts and I. Stark, "Operational reasoning for functions with local state," *Higher Order Operational Techniques in Semantics (HOOTS)*, pp. 227–273, 1998.

[58] D. Dreyer, G. Neis, and L. Birkedal, "The impact of higher-order state and control effects on local relational reasoning," in *15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2010, pp. 143–156. [Online]. Available: https://doi.org/10.1145/1863543.1863566