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 $\pi$-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].

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.

This paper develops a flow-sensitive IFC session type system for the language $\text{SILL}_\text{sec}$ and proves noninterference for $\text{SILL}_\text{sec}$ in addition to type safety. $\text{SILL}_\text{sec}$ 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 $\pi$-calculus [16], [38]. The intuitionistic foundation turns run-time configurations of processes into trees, connecting a providing process with exactly one client.

The $\text{SILL}_\text{sec}$ 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 $\text{SILL}_\text{sec}$ type system becomes flow-sensitive, allowing more secure programs to successfully type check than would be possible with maximal secrecy alone.

Noninterference of $\text{SILL}_\text{sec}$ 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 $\text{SILL}_\text{sec}$;
- proofs of type safety and noninterference of $\text{SILL}_\text{sec}$;
• 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 SILL 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 SILL. 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], [3] based on a banking example and illustrates violations of end-to-end confidentiality. We base the discussion on the language SILL that we formalize and for which we prove noninterference in the remainder of this paper. SILL is a terminating language with higher-order channels, allowing channels to be sent over channels.

In SILL, we can define the protocol according to which an authorization process interacts with a customer seeking access to their bank account as follows:

\[
\text{auth} = \&\{ \text{tok}_i : \oplus \{ \text{succ: account} \otimes 1, \text{fail: 1} \}, \ldots, \\
\text{tok}_n : \oplus \{ \text{succ: account} \otimes 1, \text{fail: 1} \} \}
\]

The connectives &, ⊕, ⊗, 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 types 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, ), 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, for every imaginable authorization token.

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

\[
\begin{align*}
\text{customer} & = \text{auth} \rightarrow 1 \\
\text{account} & = \oplus\{ \text{high: 1, med: 1, low: 1} \} \\
\text{rate} & = \&\{ \text{lowRate: 1, highRate: 1} \}
\end{align*}
\]

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:

\[
\begin{align*}
\text{guest} & \sqsubseteq \text{alice} \sqsubseteq \text{bank} \\
\text{guest} & \sqsubseteq \text{bob} \sqsubseteq \text{bank}
\end{align*}
\]

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

\[
\begin{align*}
\text{Alice} & :: y : \text{customer[alice]} \\
& y \leftarrow \text{Alice} \leftarrow \cdot \quad (// \cdot \leftarrow y : \text{customer} \\
& w \leftarrow \text{recv y}; w : \text{tok}_1 ; // w : \oplus\{ \text{succ: account} \otimes 1, \text{fail: 1} \} \leftarrow y : 1 \\
& \text{case w (} \text{succ} \Rightarrow v \leftarrow \text{recv w} ; w : \text{account} \leftarrow y : 1 \\
& \quad \text{case v (} \text{high} \Rightarrow \text{wait v} ; \text{wait w} ; \text{close y} \\
& \quad \quad | \text{med} \Rightarrow \text{wait v} ; \text{wait w} ; \text{close y} \\
& \quad \quad | \text{low} \Rightarrow \text{wait v} ; \text{wait w} ; \text{close y} \\
& \quad \quad | \text{fail} \Rightarrow \text{wait w} ; \text{close y} )) @ Alice
\end{align*}
\]

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 · 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 ← denote variable bindings. For the time being, we ignore the secrecy annotations [alice] and @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
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 exchanged. 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.

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.

\[
\begin{align*}
  &\text{u:}\{s:\text{account}, f:1\}[\text{alice}]\vdash \text{aAuth :: x:auth[alice]} \\
  &x \leftarrow \text{aAuth} \leftarrow u = ( \\
  &\quad \text{case } x (\text{tok}_j \Rightarrow x.\text{succ}; u.s; \text{send } u x; \text{close } x \\
  &\quad \quad | \text{tok}_i \neq j \Rightarrow x.\text{fail}; u.f; \text{wait } u; \text{close } x))@\text{alice}
\end{align*}
\]

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.

\[
\begin{align*}
  &\vdash \text{aAcc :: u:}\{s:\text{account}, f:1\}[\text{alice}] \\
  &u \leftarrow \text{aAcc} \leftarrow = ( \\
  &\quad \text{case } u (s \Rightarrow u.\text{high}; \text{close } u \\
  &\quad \quad | f \Rightarrow \text{close } u))@\text{alice}
\end{align*}
\]

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.

![Table I: Overview of intuitionistic linear session types in SILLsec together with their operational meaning.](image)

**Fig. 1:** State transition in process configuration due to $\rightarrow o$.

**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 $\rightarrow o$ 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 associated with Bob have maximal secrecy [bob]. The Bank process itself has the highest maximal secrecy [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

\[x; \text{auth[alice]}, y; \text{customer[alice]}, x'; \text{auth[bob]}, y'; \text{customer[bob]}, u; \text{rate[guest]}; \text{Bank :: z:1[bank]} ~ z \leftarrow \text{Bank} ~ x, x', y, y', u = ( \\
\text{send } x y; \text{send } x'y'; u.\text{lowRate}; \\
\text{wait } y; \text{wait } y'; \text{wait } u; \text{close } z)@\text{guest}\]
implement the maximal secrecy of a process with its secrecy as well as the process, a programmer must indicate the process’ maximal secrecy. We refer to the following tree invariant, 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 $\Delta$ 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 $\Delta$.

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 $x$. To this end, it increases the running secrecy upon each receive 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 $\Delta$.

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) $\forall y:B[d'] \in \Delta (\Psi \vdash d' \subseteq d)$

(ii) $\Psi \vdash c \subseteq d$

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 $\Delta$ and given the secrecy levels in the security lattice $\Psi$. $\Delta$ 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 $\Delta$ 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 $\Delta$.

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 $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i$. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Channels only exist at run-time, being allocated whenever a process is spawned and substituted for the channel variables $x$ and $y_i. Components
(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 **SILL-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 1 and then conclude with the judgmental rules cut and identity.

1) **Internal and External Choice:** Internal (⊕) and external (k) choice are the branching constructs, giving the choice to the provider or the client, respectively.

   \[ \Psi \vdash \Delta \vdash P \otimes\phi_1 :: y:A[k] \quad k \in L \]

   \[ \Psi \vdash \Delta \vdash (y,k):\phi_1 \otimes\phi_2 \vdash y:A[\ell\in\ell_k] \quad \forall k \in L \]

   \[ \Psi \vdash \Delta \vdash x:A[k] \perp Q_k \otimes\phi_2 :: y:C[\psi'] \quad \forall k \in L \]

   \[ \Psi \vdash \Delta \vdash (\text{case} x(\ell \Rightarrow Q_\ell)\otimes\phi_2)[\ell_1 \vdash y:C[\psi'] \quad \forall k \in L \]

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 q, 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 by at least the maximal secrecy of the sending channel. In ⊗L, the premise d_1 ⊆ c makes this adjustment. In &R, no explicit adjustment is needed because the new running secrecy d_1 ⊆ 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 ⊕L, this guard \( d_1 \subseteq c \) is already established by the tree invariant. &L explicitly establishes the guard with the premise \( \Psi \vdash d_1 \subseteq c \).

2) **Higher-Order Channels:** Tensor (⊗) and lolli (−−) denote channel output (send) and input (receive), respectively.

\[ \Psi \vdash \Delta, z:A[\psi'] \vdash (\text{send } y; P)[\otimes\phi_2] :: y:A \otimes B[\psi'] \otimes R \]

\[ d_2 = c \otimes d_1 \quad \Psi' \vdash (\text{recv } x; P)[\otimes\phi_2] :: y:C[\psi'] \]

\[ \Psi \vdash \Delta, x:A[\psi'] \vdash P \otimes\phi_2 :: y:B[\psi'] \otimes L \]

\[ \Psi \vdash (\text{recv } y; P)[\otimes\phi_2] :: y:A \otimes B[\psi'] \rightarrow R \]

\[ \Psi \vdash \Delta, x:A[d], x:A \otimes B[d] \vdash (\text{send } y; P)[\otimes\phi_2] :: y:C[\psi'] \rightarrow L \]

To understand that the rules preserve the tree invariant, it is helpful to remind ourselves that the connectives ⊗ and −− change the tree structure, making a child a sibling of the sender and a sibling the child of the recipient, respectively. ⊗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 ⊗. As well, ⊗R also implements the secrecy pas de deux, since \( d_1 \subseteq c \) by assumption. While ⊗R licenses us to assume in ⊗L that the maximal secrecy \( \psi \) 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, \( \psi \) stands for a secrecy variable, and we extend the security lattice with \( \psi = c \). The premise \( d_2 = c \otimes d_1 \) in ⊗L lastly implements the secrecy pas de deux, raising the running secrecy of the continuation \( P \) to \( c \), unless \( c \subseteq d_1 \). The reasoning for −−R and −−L are analogous, but with the roles reversed.

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

\[ \Psi \vdash (\text{close } y)[\otimes\phi_2] :: y:1[c] \rightarrow R \]

\[ \Psi \vdash d_2 = c \otimes d_1 \quad \Psi \vdash Q_\phi \otimes\phi_2 :: y:T[d] \rightarrow L \]

\[ \Psi \vdash \Delta, x:1[c] \vdash (\text{wait } x; Q)[\otimes\phi_2] :: y:C[\psi'] \rightarrow L \]

1R trivially preserves the tree invariant because there is no continuation and implements the secrecy pas de deux since \( d_1 \subseteq c \) 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.

\[ \Psi : x:A[\psi'] \vdash (y \leftarrow x)[\otimes\phi_2] :: y:A[\psi'] \text{ Fwd} \]

\[ \forall z:A[c'] \in \Delta_1 : \forall c' \subseteq d' : \Psi \vdash P \otimes\phi_2 :: x:B[\psi'] \rightarrow \Delta_2 \quad \Psi \vdash d' \subseteq d \quad \Psi : x:B[\psi'], \Delta_2 \vdash Q[\otimes\phi_2] :: y:C[\psi'] \rightarrow \Delta_3 \quad \text{Cut} \]

\[ \Psi : \Delta_1, \Delta_2 \vdash ((x') \leftarrow P)[\otimes\phi_2] ; Q[\otimes\phi_2] :: y:C[\psi'] \rightarrow \Delta_3 \]
We briefly comment on Cut. The premise $\Psi \vdash d' \subseteq d$ establishes the tree invariant for the continuation $Q$ and the premise $\forall z:A(c') \in \Delta_1, \Psi \vdash c' \subseteq d'$ for the spawned process $P$. The premise $\Psi \vdash d_1 \subseteq d_2 \subseteq d'$ 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 spawn 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.

$x_1; & (\{x:1,f:1\}[\text{guest}],\text{account}:f:1) & \vdash \text{SneakyAuth} :: x:a\text{uth}[\text{alice}]$

$x \leftarrow \text{SneakyAuth} \leftarrow u, x_1 =$

\[
\left\{\begin{array}{l}
\text{case } x (\text{tok}_j \Rightarrow x, \text{su}\text{ce};u; s; z_1 \leftarrow S; x_1; // \text{insecure spawn}) \\
\quad \text{send } u; x; \text{wait } z_1; \text{close } x \\
\text{| tok_{i\neq j} \Rightarrow x, \text{fail}; u; f; z_1 \leftarrow F; x_1; // \text{insecure spawn})} \\
\quad \text{wait } u; \text{wait } z_1; \text{close } x) @ (\text{alice}
\end{array}\right.
\]

$x_1; & (\{x:1,f:1\}[\text{guest}] @ S :: z_1:1[@ (\text{alice}]$

\[z_1 \leftarrow S \leftarrow x_1 = (x_1; s; \text{wait } x_1; \text{close } z_1) @ \text{guest}\]

$x_1; & (\{x:1,f:1\}[\text{guest}] @ F :: z_1:1[@ (\text{alice}]$

\[z_1 \leftarrow F \leftarrow x_1 = (x_1; f; \text{wait } x_1; \text{close } z_1) @ \text{guest}\]

B. Asynchronous Dynamics

We define an asynchronous dynamics for SILL$_{ac}$ because it is not only more practical but also allows for a more accurate statement of noninterference. The dynamics is in line with [5], [30], 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 mess$(P)$ 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 mess$(P)$ is depicted in red. This process has a subtree, in case of $\otimes$ and $\rightarrow$. 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 $y_\alpha$. The forward then links the two generations $y_{\alpha+1}$ and $y_\alpha$ appropriately. In case of a positive connective, $y_{\alpha+1}$ is forwarded to $y_\alpha$, in case of a negative connective, $y_\alpha$ is forwarded to $y_{\alpha+1}$. Once the message has been received, it terminates and $y_{\alpha+1}$ is substituted for $y_\alpha$ 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 $\mathcal{C} \Rightarrow \Delta_0 \vdash \mathcal{C}'$ 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 $x$ occurring in the process terms $P$ and $Q$ in the post-state. The generation $\alpha$ of a channel $y_\alpha$ is incremented to $\alpha + 1$ whenever a new message is spawned, except for 1 because there is no continuation. Lastly we point out that $\text{fwd}$ is not defined for any channels in $\Delta'$ because those configurations are considered posed, 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 proc$(x,d), \text{P@}d_1)$ and messages mess$(P)$. We refer to the configuration $\mathcal{C}$ as open because it is typed relative to free channels in $\Delta$. 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 $\Delta'$, using sessions in $\Delta$, and given the security lattice $\Psi$. Both $\Delta'$ and $\Delta$ 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.

\[
\begin{array}{c}
\Psi; x:A[d] \vdash \cdot :: (x:A[d]) & \text{emp}_1 \\
\Psi; \vdash \cdot :: (\cdot) & \text{emp}_2
\end{array}
\]

\[
\begin{array}{c}
\Psi; d_1 \subseteq d \quad \forall y:B[d'] \in \Delta_0, \Delta (\Psi \vdash d' \subseteq d) \\
\Psi; \Delta_0 \vdash \mathcal{C} :: \Delta \\
\Psi; \Delta_0, \Delta \vdash P@d_1 :: (x:A[d]) & \text{proc}
\end{array}
\]

\[
\begin{array}{c}
\forall y:B[d'] \in \Delta_0, \Delta (\Psi \vdash d' \subseteq d) \\
\Psi; \Delta_0 \vdash \mathcal{C} :: \Delta \\
\Psi; \Delta_0, \Delta \vdash P@d :: (x:A[d]) & \text{msg}
\end{array}
\]

\[
\begin{array}{c}
\Psi; \Delta_0 \vdash \mathcal{C} :: \Delta \\
\Psi; \Delta_0, \Delta \vdash C_1 :: x:A[d] & \text{comp}
\end{array}
\]

Rule $\text{comp}$ types an open forest, singling out the open tree $C_1$ rooted at $x$. Rules $\text{proc}$ and $\text{msg}$ 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 $\text{emp}_1$ and $\text{emp}_2$, finally, type an empty open forest.
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 \( \Psi, \Delta \models C :: \Delta' \) 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 \( \Delta \) or \( \Delta' \). 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 \( \Delta_1, \Delta_2 \models C_1, C_2 :: \Lambda, w : A'[c] \) is poised iff either \( C_1, C_2 \) is empty or \( \Delta_1 \not\models C_1 :: \Lambda \) is poised and \( \Delta_2 \models C_2 :: w : A'[c] \) is poised. The configuration \( \Delta_2 \models C_2 :: w : A'[c] \) is poised iff it cannot take any steps and at least one of the following conditions hold:

- \( C_2 \) is an empty configuration.
- \( C_2 = C'_2 \text{msg}(P) C''_2 \) such that \( \text{msg}(P) \) is a negative message along \( y \in \Delta_2 \), i.e. \( y : \{ \ell : A \} \in L[c] \models \text{msg}(P) : x : A[c], \) and both subconfigurations \( C'_2 \) and \( C''_2 \) are poised.
- \( \Delta_2 \models C = C'_2 \text{proc}(x'[c], P@d_1) C''_2 \) such that \( \text{proc}(x'[c], P@d_1) \) attempts to receive along a channel \( y \in \Delta_2 \).
- \( \Delta_2 \models C = C'_2 \text{msg}(P) C''_2 \) such that \( \text{msg}(P) \) is a positive message sent along \( w : A'[c] \), i.e. \( x : A[c] \models \text{msg}(P) : w : \{ \ell : A \} \in L[c] \) or \( x : B[c], \gamma : A[c] \models \text{msg}(P) : w : A \otimes B[c], \) or \( \models \text{msg}(P) : w : 1[c] \), and subconfiguration \( C'_2 \) is poised.
- \( \Delta_2 \models C = C'_2 \text{proc}(w[c], P@d_1) C''_2 \) such that \( \text{proc}(w[c], P@d_1) \) attempts to receive along \( w : A'[c] \).
- \( \Delta_2 \models C = C'_2 \text{proc}(w[c], P@d_1) C''_2 \) such that \( \text{proc}(w[c], P@d_1) \) attempts to receive along \( w : A'[c] \).

V. Key Ideas - Part II

This section develops the main ideas underlying the session logical relation used to prove noninterference of SILL. The next section puts these ideas into action.

**Noninterference** essentially amounts to a program equivalence up to the secrecy level \( \xi \) of the observer, requiring that two runs of a program may only differ in outputs whose secrecy level is above or incomparable to \( \xi \). 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 \( D_1 \) and \( D_2 \) of an open program with closing substitutions \( C_1, F_1 \) and \( C_2, F_2 \), respectively, provided that \( C_1, F_1 \) and \( C_2, F_2 \) will send the same messages to \( D_1 \) and \( D_2 \), respectively.

The property is expressed as

\[
(C_1, D_1, F_1; C_2, D_2, F_2) \in V_{\xi}^\text{s}[\Delta \models K]
\]

where \( \Delta \) amounts to the typing of channels connecting \( C_1 \) and \( C_2 \) with \( D_1 \) and \( D_2 \), respectively, and \( K \) to the typing of the channel connecting \( D_1 \) and \( D_2 \) with \( F_1 \) and \( F_2 \), respectively. We refer to \( \Delta \) and \( K \) as the interface of \( D_1 \) and \( D_2 \).

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 \( D_1 \) and \( D_2 \) for two runs \( D_3 \) and \( D_4 \) of an open program, with \( \Psi \models \Delta_3 \models K_{D} \) and \( \Psi \models \Delta_4 \models K_{D} \), such that the observable channels defined by the projection \( \downarrow \downarrow \xi \) are the same, i.e., \( \Delta_3 \downarrow \downarrow \xi = \Delta_4 \downarrow \downarrow \xi = \Delta \) and \( \Delta_3 \downarrow \downarrow \xi = \Delta_4 \downarrow \downarrow \xi = K \). The left-over, non-observable channels in \( \Delta_3, \Delta_4, K_3, K_4 \) are closed off and internalized into \( D_3 \) and \( D_4 \), yielding \( D_1 \) and \( D_2 \), respectively.

The message exchange depicted in Fig. 4 is a send, denoted by the red node in \( D_1 \) and \( D_2 \). The node is a message \( \text{msg}(P) \), and the figure captures the positive case depicted in Fig. 2. In the post-states \( C'_1 \text{D}_1 F'_1 \) and \( C'_2 \text{D}_2 F'_2 \), this message is simply pushed into the substitutions \( F'_1 \) and \( F'_2 \). The value interpretation of \( (C_1, D_1, F_1; C_2, D_2, F_2) \) is now phrased in terms of the transition, requiring that

\[
(C_1, D_1, F_1; C_2, D_2, F_2) \in V_{\xi}^\text{s}[\Delta \models K], \text{if} \quad (C'_1 \text{D}_1 F'_1, C'_2 \text{D}_2 F'_2) \in E_{\xi}^s[\Delta \models K']
\]
Fig. 4: Session logical relation for noninterference: key ideas.

The post-states $C_1', D_1', F_1'$ and $C_2', D_2', F_2'$ now take any number of internal transitions until $C_1'', D_1'', F_1''$, $C_2'', D_2'', F_2''$, and $F_2''$ are each individually poised, demanding a message exchange along an observable channel. We thus require this poised configuration to be in the value interpretation

$$(C_1'', D_1'', F_1'', C_2'', D_2'', F_2'') ∈ V_Φ[Δ' ⊨ K']$$

The choice to simply push a message $msg(P)$ across the interface to the recipient, rather than 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 $msg(P)$ 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. VII. 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 $Ψ; Δ_1 ⊨ D_1 :: x_α; A_1[c_1]$ we need two closing configurations. One to provide $Δ_1$ without using any resources, i.e., $Ψ; Δ_1 ⊨ C_1 :: Δ_1$. The other to use $x_α; A_1[c_1]$ as a resource and offer a terminating type, i.e., $Ψ; x_α; A_1[c_1] ⊨ F_1 :: y_β; 1[c']$. The name and secrecy of the channel provided by $F_1$ 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 $Ψ; x_α; A_1[c_1] ⊨ F_1 :: ∅$. We keep in mind that a providing type $Δ$ behaves as a terminating channel. In this paper, we often use $K := x_α[c]; A | ·$ 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).**

$$(Δ_1 ⊨ D_1 :: x_α; A_1[c_1]) ≜ (Δ_2 ⊨ D_2 :: y_β; A_2[c_2])$$

is defined as $Ψ; Δ_1 ⊨ D_1 :: x_α; A_1[c_1]$ and $Ψ; Δ_2 ⊨ D_2 :: y_β; A_2[c_2]$ and $Δ_1 ⊩ Δ_2$ and $Δ_1 ⊩ Δ_2$. The name and secrecy of the channels provided by $F_1$ and $F_2$ are not significant in our setting; $y_α$ and $y_β$ can only send closing messages when all observable channels are closed. Thus we disregard them and consider $K := K$ as a placeholder in the relation.

To track the observable messages using $E_Φ^c$, we need to know that $B_i$ can be broken down into $C_1, D_1, F_1$ such that $Ψ; Δ ⊨ C_1 :: Δ$, and $Ψ; Δ ⊨ D_1 :: K$, and $Ψ; Δ ⊨ F_1 :: ∅$. We prove that this property holds for any $B_i$ 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 $B_1$ and $B_2$ into $C_1, D_1, F_1$ and $C_2, D_2, F_2$, respectively, we compute each subconfiguration separately. We write $C, D, F ⊨ C', D', F'$ if (i) $C ⊨ C'$, (ii) $D ⊨ D'$ and (iii) $F ⊨ F'$. 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. VII. We call...
Fig. 5: Session Logical Relation for Noninterference: property (left), condition (right).
$C_i, D_i, F_i$ poised if its subconfigurations $C_i$, and $D_i$, and $F_i$ are poised. We write $C_i, D_i, F_i \vdash \Delta_i \rightarrow K_i$ stating that $C_i, D_i, F_i \rightarrow \Delta_i \rightarrow K_i$ and $C_i', D_i', F_i'$ is poised. ($\rightarrow K_i$ refers to zero or more steps taken with $\rightarrow \Delta_i$).

To relate two poised configurations we use the value relation $\forall \delta_\xi \in [\Delta_i \vdash K_i]$. This relation establishes equality of the content of every message fired from $D_1$ and $D_2$ before adding them to the closing configurations $C_i$ and $F_i$. (See Fig. 4). In the case of sending higher order channels (lines 4 and 10 in Fig. 5), we assume 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 $C_i$ and $F_i$, we assume they have the same content ready to be moved to $D_1$ and $D_2$. 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 $\text{proc}(\xi | c | y_o \leftarrow x_\beta)$ forwards channel $(x_\beta)$ to an observable channel $(y_o)$, the substitution of $x_\beta$ for $y_o$ amounts to a broadcast of the name of $x_\beta$ along the observable channel $y_o$. The channel $x_\beta$ 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 assume that such forwarding rules in $D_1$ and $D_2$ always broadcast the same names. In the dual case (line 12 of Fig. 5) we assume that the closing configurations $C_i$ and $C_2$ broadcast the same names. In both cases we continue by monitoring the rest of the computation along $x_\beta$. 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 $M$ of the process typing judgments $\Psi; \Delta \vdash P :: \Delta$ used in the typing derivation of $C$. 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.

**VII. Metatheory**

In this section we establish the main properties of the system. We show that SILL-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 $\Psi; \Delta \vdash C :: \Delta'$ and $C \rightarrow \Delta \rightarrow \Delta'$, then $\Psi; \Delta \vdash C' :: \Delta'$. Moreover $C' < C$ by the multiset ordering.

**Proof.** The proof is by case analysis of $C \rightarrow \Delta \rightarrow \Delta'$ of $C'$ and inversion on the typing judgment $\Psi; \Delta \vdash C :: \Delta'$. See Theorem A.5 in [45] for details.

**Theorem VII.2 (Progress).** If $\Psi; \Delta \vdash C :: \Delta'$, then either $C \rightarrow \Delta \rightarrow \Delta'$ or $C$ is poised.

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

Termination of SILL-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 $\xi$ and configurations $\Psi; \Delta \vdash D :: x_o : T[c]$, we have $(\Delta \vdash D :: x_o : T[c]) \equiv^\Psi_v (\Delta \vdash D :: x_o : T[c])$.

Our noninterference theorem asserts that a well-typed open configuration $D$ 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 $D$, 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 $\xi$. 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 $\xi$. 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 $\xi$ but with paths to observable channels passing through channels with maximal secrecy level higher than or incomparable to $\xi$. 

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 case \( y' \) in the open program and its counterpart case \( y' \) in the other run. They both have running secrecy \( d_1 \subseteq \xi \), 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 of the parent may be higher or incomparable to the maximal secrecy 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 recv or case, then its quasi running secrecy is equal to its running secrecy.
- If the process term is of the form case \( y''_o(\ell \Rightarrow P_1)_{\ell \in L} \otimes d_1 \) or \( x^{\psi} \leftarrow \text{recv } y''_c ; P_2 \otimes d_1 \), then its quasi running secrecy is \( d_1 \sqcup c \).
- If the node is a message of a negative type along channel \( y''_c \), its quasi running secrecy is \( c \).
- If the node is a message of a positive type along channel \( y''_c \) 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 bottom to top.

**Definition VII.5 (Relevant Channels and Processes).** Consider configuration \( \Delta \models D \models K \) and observer level \( \xi \). A channel is relevant in \( D \) if 1) it is has a maximal secrecy level lower than or equal to \( \xi \), and 2) it is either an observable channel or it shares a process or message with quasi running secrecy less than \( \xi \) 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 \( D \) will be relevant.

A relevant process or message has quasi running secrecy less than or equal to \( \xi \) and at least one relevant channel. \( C \models \xi \) are the relevant processes and messages in \( C \). We write \( C_1 \models \xi =_\xi C_2 \models \xi \) 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 \( \xi \) and configurations \( \Psi; \Delta_1 \models D_1 \models u_1; A_1[1] \) and \( \Psi; D_2 \models u_2; A_2[2] \) with \( D_1 \models \xi = D_2 \models \xi \) and \( u_1; A_1[1] \downarrow \xi = u_2; A_2[2] \downarrow \xi \) we have

\[
(\Delta_1 \models D_1 \models u_1; A_1[1]) \equiv^y (\Delta_2 \models D_2 \models u_2; A_2[2]).
\]

**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 \( \rightarrow_{\Delta \models K} \) for a closed configuration \( C, D, F \in \text{Tree}(\Delta \models K) \). Where \( C, D, F \in \text{Tree}(\Delta \models K) \) is defined as \( \cdot \models C \(models ) \Delta \), and \( \Delta \models D \models K \) and \( K \models F \models y[1] \). The idea is to run this closed configuration to completion, while accumulating the messages exchanged between the open program \( D \) and closing configurations \( C \) and \( F \) in a queue. To this end, we define \( C, D, F \rightarrow_{\Delta \models K} \) queue by first stepping \( C, D, F \) in terms of \( \rightarrow_{\text{poised}} \) until we get a configuration \( C', D', F' \) that is poised (i.e., \( C, D, F \rightarrow_{\text{poised}} C', D', F' \)) and then append to the queue the message that is waiting to be sent. For example, \( C, D', \text{msg}(y''_c; k; y''_c \leftarrow u''_c) \rightarrow_{\Delta \models K} \text{queue'} \text{msg}(y''_c; k; y''_c \leftarrow u''_c) \), where queue’ is a recursive invocation (i.e., \( C, D', \text{msg}(y''_c; k; y''_c \leftarrow u''_c) \rightarrow_{\Delta \models K} \text{queue'} \)). We use an overline to indicate the direction of a
It is straightforward to show that \( CDF \mapsto \mathfrak{m} \) if and only if for some queue, we have \( C, D, F \mapsto \mapsto_{\Delta \vdash K} \) queue, where queue is the list of observable messages being exchanged between \( C \) and the closing configurations \( D \) and \( K \).

**Definition VII.7.** Define \( \text{queue}_1 =_{\xi} \text{queue}_2 \) as either

- \( \text{queue}_1 = \text{queue}_2 = \cdot \), or
- \( \text{queue}_1 = q_1 \text{queue}_1', \) and \( \text{queue}_2 = q_2 \text{queue}_2', \) and \( q_1 = q_2, \) and \( \text{queue}_1' =_{\xi} \text{queue}_2', \) or
- \( \text{queue}_1 = \overline{q} \text{queue}_1', \) and \( \text{queue}_2 = \overline{q} \text{queue}_2', \) and if \( q_1 \neq q_2 \) then \( \text{queue}_1' =_{\xi} \text{queue}_2' \).

**Theorem VII.8.** For \((C_1, D_1, F_1; C_2, D_2, F_2) \in \text{Tree}_{\psi}(\Delta \vdash K), \) if \((C_1, D_1, F_1; C_2, D_2, F_2) \in \mathcal{E}_{\psi}^{\xi} \Delta \vdash K \), then

\[ (C_1, D_1, F_1) \mapsto_{\Delta \vdash K} \text{queue}_1 \text{ and } (C_2, D_2, F_2) \mapsto_{\Delta \vdash K} \text{queue}_2 \]

such that \( \text{queue}_1 =_{\xi} \text{queue}_2 \).

**Proof.** The proof is straightforward by matching the cases in the definition of \( \mathcal{E} \) with the cases in the definition of \( \mapsto_{\Delta \vdash K} \).

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 intutionistic 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.