A Secure Compiler for Session Abstractions

Ricardo Corin[1,2,3], Pierre-Malo Deniélou[1,2],
Cédric Fournet[1,2], Karthikeyan Bhargavan[1,2], and James Leifer[1]

[1] MSR-INRIA Joint Centre
[2] Microsoft Research
[3] University of Twente

Abstract. Distributed applications can be structured as parties that exchange messages according to some pre-arranged communication patterns. These sessions (or contracts, or protocols) simplify distributed programming: when coding a role for a given session, each party just has to follow the intended message flow, under the assumption that the other parties are also compliant.
In an adversarial setting, remote parties may not be trusted to play their role. Hence, defensive implementations also have to monitor one another, in order to detect any deviation from the assigned roles of a session. This task involves low-level coding below session abstractions, thus giving up most of their benefits.
We explore language-based support for sessions. We extend the ML language with session types that express flows of messages between roles, such that well-typed programs always play their roles. We compile session type declarations to cryptographic communication protocols that can shield programs from any low-level attempt by coalitions of remote peers to deviate from their roles. Our main result is that, when reasoning about programs that use our session implementation, one can safely assume that all session peers comply with their roles—without trusting their remote implementations.

Table of Contents

1 Session types for secure distributed programming

Programming networked, independent systems is complex, because the programmer has little control over the runtime environment. To simplify his task, programming languages and system libraries offer abstractions for common communication patterns (such as private channels or RPCs), with automated support to help the programmer use these abstractions reliably and to relieve him from their low-level implementation details (such as message format and routing). As an example, web services promote declarative types and policies for messaging, with tools that can automatically fetch these declarations and set up proxies with a simple typed programming interface.

From a security perspective, when parts of the system and some of the remote parties are not trusted, communication abstractions can be especially effective: relying on cryptographic protocols, secure implementations of these abstractions can sometimes entirely shield programmers from low-level attacks (such as message interception and rewriting) [1,2]. Unfortunately, this is seldom the case in practice, as security concerns force the programmer to understand low-level implementation issues.

Beyond simple abstractions for communications, distributed applications can often be structured as parties that exchange messages according to some fixed, pre-arranged patterns. These sessions (also named contracts, or workflows, or protocols) simplify distributed programming by specifying the behaviour of each network entity, or *role*. By agreeing in advance on a common session specification, the parties can resolve most of the complexity upfront. Then, when coding a role for a given session, each party just has to follow the agreed message flow for this role, under the assumption that the other parties are also compliant. At run-time, sessions can finally be instantiated by mapping roles to actual principals and their hosts.

Language-based support for sessions is the subject of active research [23,20,9,34,7] [8,15,19,35,37,24]. In particular, several recent type systems statically ensure compliance to session specifications. In their setting, type safety implies that user code that instantiates a session role always behaves as prescribed in the session. Thus, assuming that every distributed program that may participate in a session is well-typed, any run of the session follows its specification.

In an adversarial setting, remote parties may not be trusted to play their role. Hence, defensive implementations of session roles also have to monitor one another, in order to prevent any confusion between parallel sessions, to ensure authentication, correlation, and causal dependencies between messages and, more generally, to detect any deviation from the other assigned roles of a session. Left to the programmer, this task involves delicate low-level coding below session abstractions, which defeats their purpose. Instead, we propose to systematically compile session specifications to cryptographic protocols.

In this paper, we explore language-based support for sessions and their implementations, as follows:

1. We design a small embedded language of types for specifying messages, roles, and sessions, and we identify a secure implementability condition for these sessions.
2. We extend F# [33] (a dialect of ML [27,29]) with distributed communication and sessions, so that type safety yields functional guarantees: in well-typed programs using sessions, any sent message is expected by its receiver, with matching payload types.

3. We compile session types to cryptographic communication protocols, coded in F#, that can shield programs using sessions from any low-level attempt by coalitions of remote peers to deviate from their roles. We thus obtain a secure, functional, distributed implementation of sessions.
4. Our main security theorems state that the safety guarantees implied by session types do not depend on the implementations of any remote peers: from the viewpoint of programs using sessions, any action that may occur with our distributed implementation may also occur in an idealized setting, with a centralized implementation that globally enforces all session types.
5. We report experimental results with a prototype compiler for F+S, including the study and benchmarking of a simplified conference management system (CMS), expressed as a session.

To our knowledge, this paper provides the first secure implementation of session types, both formally and concretely. It relates the semantics of three languages: at the level of types, simple processes to specify communication patterns and payloads; as a source language, a subset of F# with distributed communications and typed sessions; as an implementation language, a subset of F# with distributed communications and cryptography.

Typed session APIs Our compiler extracts session definitions, verifies that they meet the secure implementability condition, generates the corresponding cryptographic protocols, and emits their code as F# modules. However, it leaves the code of programs that use sessions unchanged, treating the session constructs of the extended language as ordinary higher-order function calls to their implementations. Hence, user code calls our generated code to enter a session and then, for each received message, generated code calls back user code and resumes the protocol once user code returns the next message to be sent. Taking advantage of this calling convention, with a separately-typed user-code continuation for each state of each role of the session, we can thus entirely rely on ordinary typing à la ML to enforce session typing in user code. (In the following, as we focus on session security, we treat this important but well-understood aspect of session types informally.)

Cryptographic protocol outline The compiled protocols rely on a combination of standard techniques for authentication and anti-replay protection. The compiler does not introduce any additional message: each abstract session message is mapped to a cryptographic message with the same sender and receiver. Principals are authenticated using X.509 certificates. All messages include a unique session identifier (obtained as the joint cryptographic hash of its session type, its assignment of principals to roles, and a fresh session nonce) and a series of signatures: one signature from the message sender, plus one forwarded signature from each peer involved in the session since the receiver's last message (or the start of the session). At any point in a session, each protocol role knows exactly which messages to expect and what they should contain, so we can use compact wire formats and compile simple, specialized message handlers. Any message that deviates from the expected format can be silently dropped, or reliably detected as anomalous.

Verifying the protocol compiler In our view, the security of automatically-generated cryptographic protocol implementations must rely on formal verification. To this end, our language design and prototype implementation build on the approach of Bhargavan *et al.* [4], which narrows the gap between concrete executable code and its verified model. Our generated code depends on libraries for networking, cryptography, and principals, with dual implementations.

- A concrete implementation of these libraries uses standard cryptographic algorithms and networking primitives; the produced code supports distributed execution.
- A symbolic implementation of these libraries defines cryptography using algebraic datatypes, in Dolev-Yao style; the produced code supports concurrent execution, and is also our formal model.

Except for these libraries, the same code is shared between execution and verification. Thus, our security theorems directly apply to any user code calling any session-implementation code generated by our compiler calling symbolic-library code, within a formal model of a subset of F#. This yields stronger guarantees than those obtainable by studying an abstract, ad hoc model of the protocol loosely related to actual executable code.

Related work Session types have been explored first for process calculi [20,23,37], as a way to control interaction on single channels. Behavioral types [9,25] support more expressive sessions, typed as CCS processes possibly involving multiple channels. Another type system [6] also combines session types and correspondence assertions [22]. Recent works consider applications of session types to concrete settings such as CORBA [34], a multi-threaded functional language [35], and a distributed object-oriented language [15]. In particular, the Singularity OS [19] explores the usage of typed contracts in operating system design and implementation.

In recent, independent work, Carbone *et al.* [7] present a language for describing Web interactions from a global viewpoint and describe their end-point projection to local role descriptions. Their approach is similar to our treatment of session graphs and roles in Section 2; however, their descriptions are executable programs, not types. Honda *et al.* [24] subsequently consider multi-party session types and their local projections for the pi-calculus. More generally, distributed languages such as Acute and HashCaml [31,14,5] also rely on types to provide general functional guarantees for networked programs, in particular type-safe marshalling and dynamic rebinding to local resources.

In all these works, type systems are used to ensure session compliance within fully trusted systems, excluding the presence of an (active, untyped) attacker.

Cryptographic communications protocols have been thoroughly studied, so we focus on related work on their use for securing implementations of programming-language abstractions. They can provide secure implementations for distributed languages with private communication channels [1,2]. They can also help support the distributed implementation of sequential languages such as JIF/Split [38], while preserving high-level, typed-based integrity and secrecy guarantees. In a similar vein, the Fairplay [26] system compiles high-level procedural descriptions toward secure two-party computations. In

other work, type-based secrecy and integrity guarantees are enforced by a combination of static typechecking and compilation to low-level cryptographic operations [18]. In the context of Web Services, secure sessions have been considered for the WSDL and WS-SecureConversation specification languages (see e.g. [3,8]); Bhargavan *et al.* [3] verify security guarantees for session establishment and for sequences of SOAP requests and responses (with no communication types).

Protocol synthesis and transformation have been explored in other settings: for instance, the Automatic Protocol Generation (APG) tool [30] generates authentication protocols then verified using Athena [32] and, more recently, Cortier *et al.* [12] verify the correctness of a generic transformation to protect a protocol from active attacks (but not from compromised participants).

Contents Section 2 defines two views of sessions, as global communication graphs and as local role definitions. Section 3 sets the syntax and semantics for our source and target languages. Section 4 outlines the libraries that embed our assumptions on cryptography and principals, used by our implementation. Section 5 presents our optimized cryptographic protocol, as a refinement of a basic, intuitively secure protocol. Section 6 describes our implementation code for sessions. Section 7 states our main results, formally showing the correctness of the implementation. Section 8.1 describes our prototype implementation and a case study. Section 9 concludes.

The appendix provides additional details on our implementation, including listings for selected libraries, a discussion of correspondence assertions, a detailed programming example, and the proofs.

Parts of this work appear in preliminary form in conference proceedings: a first paper presents our general approach and main theoretical results [11]; a second paper focuses on the implementation and case study [10].

The prototype implementation of our session compiler is available online, at `http://msr-inria.inria.fr/projects/sec/sessions`. It includes all the examples presented in this paper, as well as the source code for the compiler and the runtime libraries.

2 Sessions

In this paper, a *session* is a static description of the valid message flows between a fixed set of roles. Every message is of the form $f(\widetilde{v})$, where f is the message descriptor, or label, and $\widetilde{v}$ is the payload. The label indicates the intent of the message and serves to disambiguate between messages within a session. (Throughout the paper, both $\widetilde{v}$ and $(v_i)_{i<n}$ denote a comma-separated list of values $v_0, \ldots, v_{n-1}$; we use $(v_i)_{i<n}$ instead of $\widetilde{v}$ when we need to refer specifically to indexed values.)

We denote the roles of a session by a set $\mathcal{R} = \{r_0, \ldots, r_{n-1}\}$ for some $n \geq 2$. By convention, the first role (r_0) sends the first message, thereby initiating the session. In any state of the session, at most one role may send the next message—initially r_0, then the role that received the last message. The session specifies which labels and target roles may be used for this next message, whereas the selection of a particular message and payload is left to the programs that implements the roles.

We define two interconvertible representations for sessions. A session is described either globally, as a graph defining the message flow, or locally, as a process for each role defining the schedule of message sends and receives. The graph describes the session as a whole and is convenient for discussing security properties and the secure implementability condition. More operationally, local role processes are the basis of our implementation; they provide a direct typed interface for programming roles.

In the rest of this section, we describe these two representations; we explain how sessions are instantiated at runtime; we discuss session integrity; and we give an implementability condition.

2.1 Global session graphs

We represent sessions as directed graphs where nodes are session states tagged with their active role, and edges are labelled with message descriptors. Formally, a session graph $\mathcal{G} = \langle \mathcal{R}, \mathcal{V}, \mathcal{L}, m_0 \in \mathcal{V}, \mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}, r : \mathcal{V} \to \mathcal{R} \rangle$ consists of a finite set of roles $\mathcal{R} = \{r_0, \ldots, r_{n-1}\}$, a set of nodes $m, m', m_i \in \mathcal{V}$ and a set of labels $f, g, l \in \mathcal{L}$, with initial node m_0, labelled edges $(m, f, m') \in \mathcal{E}$, and a function r from nodes to roles such that $r(m_0) = r_0 \in \mathcal{R}$. We require that session graphs meet the following properties:

1. Edges have distinct source and target roles: if $(m, f, m') \in \mathcal{E}$, then $r(m) \neq r(m')$.
2. Two different edges have distinct labels: if $(m_1, f, m_1') \in \mathcal{E}$ and $(m_2, f, m_2') \in \mathcal{E}$, then $m_1 = m_2$ and $m_1' = m_2'$.

Property 1 disallows a role from sending a message to itself; such a message would be invisible to the other roles and should not be part of the session specification. Property 2 ensures that the intent of each message label is unambiguous; the label uniquely identifies the source and target session states. Note that one can always transform graphs so that they meet Property 2 by renaming message labels that occur on multiple edges.

As usual, a path is a sequence of connected edges. By Property 2 above, a sequence of labels uniquely defines a path, so we just write $\widetilde{f}$ to denote paths. To emphasize the first node of a path, we write a pair $(m, \widetilde{f})$. In particular, paths of the form $(m_0, \widetilde{f})$, where m_0 is the initial node of the graph, are called initial paths; they represent possible message sequences for the session. We say that a role r is active on a path $\widetilde{f}$ when r is the role of any source node of a label of the path.

As a running example, we consider sessions with a customer role C arranging the delivery of an item with a store role S. This arrangement may include several negotiation rounds, until both C and S agree on the details, for instance the delivery date and time. In addition, a third notary officer role O may take part in the session to record the transaction, preventing further disputes. Figure 1 displays three increasingly complex examples of session graphs:

(a) The customer C sends a Request message to store S, which may reply with either an Accept message or a Reject message.
(b) As a refinement to (a), S may either Reject as before, or accept the request and propose a delivery time by sending an Offer message. C may then either Change the delivery time or approve it by sending an Accept message.

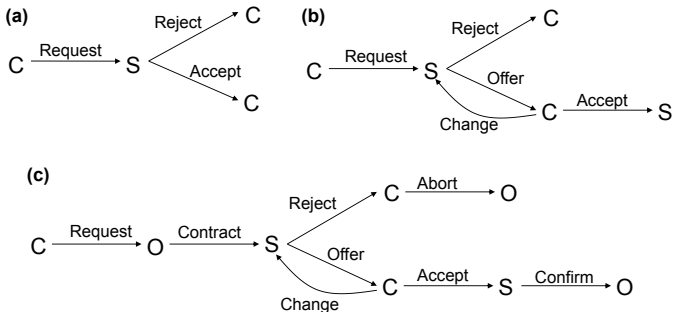

Fig. 1. Graphs for (a) a basic session, (b) a session with a cycle, and (c) a three-party session.

(c) A new officer role O acts as a notary for the transaction. Initially, C sends its Request to O, which forwards this request to S. S negotiates with C as before, and finally O receives either a Confirm from S indicating that the request is successful, or an Abort from C indicating that the request is void.

In session (a), there are only two paths from the initial node; hence, only two message sequences are allowed. In sessions (b) and (c), however, the negotiation can be repeated indefinitely, so the number and the length of possible message sequences are unbounded.

2.2 Local session roles

We also define a syntax for sessions, as a map from roles to *role processes* that specify the local operational behaviour of each role in the session:

$\tau ::=$	Payload types
$\quad$ int $\mid$ string	$\quad$ base types
$p ::=$	Role processes
$\quad !(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	$\quad$ send
$\quad ?(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	$\quad$ receive
$\quad \mu\chi.p$	$\quad$ recursion declaration
$\quad \chi$	$\quad$ recursion
$\quad 0$	$\quad$ end
$\Sigma ::=$	Sessions
$\quad (r_i : \widetilde{\tau_i} = p_i)_{i<n}$	$\quad$ initial role processes p_i for the roles r_i

Role processes can perform two communication operations: *send* (!) and *receive* (?). When sending, the process performs an internal choice between the labels f_i for $i = 0, \ldots, k-1$ and then sends a message $f_i(\widetilde{v})$ where the payload $\widetilde{v}$ is a tuple of values of types $\widetilde{\tau_i}$, a possibly empty tuple of int or string types. Conversely, when receiving, the process accepts a message with any of the receive labels f_i (thus resolving an external choice). The $\mu\chi$ construction sets a recursion point which may be reached by the process χ; this corresponds to cycles in graphs. Finally, 0 represents a completion of

```
session S1 =
  role customer = !Request:string; ?(Reject + Accept)
  role store:string = ?Request:string; !(Reject + Accept)

session S2 =
  role customer = !Request:string;mu X.
    ?(Reject + Offer:string;!(Change:string;X + Accept))
  role store:string = ?Request:string;mu X.
    !(Reject + Offer:string;?(Change:string;X + Accept))
```

Fig. 2. Local session roles for the session graphs of Figure 1(a,b) (files `S1.session` and `S2.session`)

the role for the session. On completion, a session role returns values whose types $\widetilde{\tau}_i$ are specified on the process role $r_i : \widetilde{\tau}_i = p_i$. For convenience, we often omit type annotations when the payload or return type tuple is empty. Our concrete syntax uses the keyword 'mu' for μ and keywords 'session' and 'role' in front of session and role definitions.

Figure 2 illustrates our local role syntax for the session graphs of Figure 1(a,b). (We also provide the names of the corresponding files included in our prototype compiler distribution, at `http://msr-inria.inria.fr/projects/sec/sessions`.) The concrete syntax for the session graph of Figure 1(c) appears in Appendix D.
Session S1 corresponds to graph (a), with role customer standing for C and role store standing for S. Session S2 uses recursion to represent the negotiation loop of graph (b).

We equip role processes with a simple labelled semantics that describes their execution, with labels η that range over f, $\overline{f}$ with f a message label. We identify roles up to μ-unfolding, so our semantics has just two rules for sending and receiving:

$$\text{(SEND)}\ !(f_i\!:\!\widetilde{\tau}_i\ ;\ p_i)_{i<k} \xrightarrow{\overline{f}_i}_{\mathsf{r}} p_i \qquad \text{(RECEIVE)}\ ?(f_i\!:\!\widetilde{\tau}_i\ ;\ p_i)_{i<k} \xrightarrow{f_i}_{\mathsf{r}} p_i$$

Traces of the labelled semantics represent possible series of actions for these roles. For example, the customer process in S1 can perform the following two sequences of actions:

$$!\text{Request:string; }?(\text{Reject + Accept}) \xrightarrow{\overline{\text{Request}}}_{\mathsf{r}} ?(\text{Reject + Accept}) \xrightarrow{\text{Accept}}_{\mathsf{r}} 0$$
$$!\text{Request:string; }?(\text{Reject + Accept}) \xrightarrow{\overline{\text{Request}}}_{\mathsf{r}} ?(\text{Reject + Accept}) \xrightarrow{\text{Reject}}_{\mathsf{r}} 0$$

Given the role processes for a session (Σ), if the sends and receives are correctly matched, we can construct a corresponding session graph ($\mathcal{G}$). Appendix A details this construction—implemented as the first step in our prototype session compiler—and gives a reverse construction from $\mathcal{G}$ to Σ.

2.3 Distributed session runs

At runtime, a session run involves processes for each of its roles, running on hosts connected through an untrusted network. Each process runs on behalf of a principal. In

general, a principal may be engaged in multiple sessions with other principals, may play multiple roles within a session run, and may also communicate with other principals outside the session.

A run of a session begins as a principal a_0 initiates it, taking its initial role r_0, selecting other principals to play the other roles, and sending a first message. If a_0 picked the principal a_i to play role r_i, then a_i joins the session run in role r_i only when it receives the first message sent to this role. The session run proceeds by exchanging messages between these principals until all role processes have completed, at which point the run terminates. We consider implementations that enjoy "message transparency", that is, every message send in a session is implemented as a single (asynchronous) message send on the network.

As an example, a principal Alice may begin a run of session S1 as a customer. Alice computes a unique session run identifier s, picks the principal Bob to play the role of the store, and sends the first message Request(v), for some string v, to Bob. (All messages implicitly contain the session identifier s.) On receiving the message, Bob joins the running session s as a store, sends either a Reject or an Accept message back to Alice, and thereby completes its role for the session. After receiving the response, Alice also completes its role, and the session run s is terminated.

So far, we described session executions in which every principal is compliant. If a principal is malicious, however, it may deviate from its role. We consider a threat model where some of the principals participating in a session may be malicious and may collude with an attacker that also controls the network, and can thus intercept, modify, and replay any messages.

2.4 Session integrity

We say that a distributed session implementation preserves session integrity if, during every run, regardless of the behaviour of the malicious principals and the network, the process states at the compliant principals are consistent with a run where all principals seem to comply with all sessions. Intuitively, every time a compliant principal sends or accepts a message in a session run, such a message must be allowed by the session graph; conversely, every time a malicious principal tries to derail the session by sending or replaying an incorrect message, this message must be ignored.

Session integrity requires that all message sequences exchanged at compliant principals are consistent and comply with the session graph. For instance, in a run of the session graph of Figure 1(c), a compliant officer Charlie should accept a Confirm from a store Bob only if a customer Alice previously sent an Accept for the same session run to the store Bob. Such properties on message sequences can also be interpreted as injective correspondences between message events [36] (see Section 7.1).

Conversely, if in a session run, some malicious principals, possibly in collusion with the network-based opponent, succeed in confusing a compliant principal into accepting or generating a message sequence that deviates from the session, we say that this run constitutes an *attack* against session integrity. In the example above, if the store Bob is malicious, it may Confirm a transaction to Charlie, without ever completing its negotiation with Alice, hence attempting to lead the compliant principals Alice and Charlie to

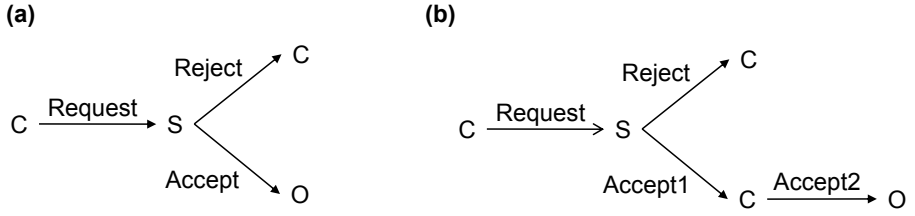

Fig. 3. (a) A session graph with a vulnerable fork and (b) its safe counterpart.

inconsistent session states. To avoid this attack, Charlie's implementation will require further cryptographic evidence from Alice and Bob.

Even if all principals are compliant, a network-based opponent could still try to confuse them by mixing messages from different session runs, or by replaying old messages. In the example above, assume the customer Alice sends a Change to store Bob, who then sends a Reject to officer Charlie. A network-based opponent may intercept the Reject message and replay instead a previous Offer message, to trigger a new iteration of the loop. Such attacks, as well as simpler attacks on the integrity of message payloads, are reminiscent of common attacks against (flawed variants of) cryptographic protocols, in the style of Dolev and Yao [16].

A secure implementability condition for sessions For some session graphs, it is difficult to rule out certain attacks without either trusting some principals, or introducing additional messages, or relying on a trusted party.

Consider for example the session of Figure 3(a), where S may send either a Reject message to C or an Accept message to O. Unless C and O exchange some information, they cannot prevent a malicious S from sending both messages, thereby breaking session integrity.

To avoid such cases, we formalize a secure implementability condition as a third property of session graphs, in addition to Properties 1 and 2 given above:

3. For any two paths $\widetilde{f}_1$ and $\widetilde{f}_2$ starting from the same node and ending with roles r_1 and r_2, if neither r_1 nor r_2 are in the active roles of $\widetilde{f}_1$ and $\widetilde{f}_2$, then $r_1 = r_2$.

Property 3 is trivially met for sessions with two roles; it excludes only some particular sessions where choices between different messages are not seen by all roles.

The session of Figure 3(a) does not meet this condition: the principal instantiating the (single) role S active on the paths $\widetilde{f}_1 =$ Reject and $\widetilde{f}_2 =$ Accept, may form a "coalition" of one, against $r_1 =$ C and $r_2 =$ O by contacting them simultaneously. Nevertheless, such vulnerable session graphs can always be transformed to functionally equivalent ones that meet Property 3, at the cost of inserting additional messages (see Appendix B for the general case). Figure 3(b) shows a safe counterpart for the vulnerable session of Figure 3(a): the message Accept is split into two messages, Accept1 and Accept2, so that S is obliged to contact C irrespective of the branch is selects.

In the rest of this paper, we consider sessions that meet Properties 1–3, and we describe distributed implementations that preserve their integrity.

3 Language specification

We now embed sessions within ML. We adopt the concrete syntax of F#, a dialect of ML, to which we add the syntax for session type definitions. Formally, we give a semantics only to a subset of this language (which we call F+S) with primitives for both session-based and channel-based communications. We compile programs in this language to a language without session constructs (which we call F).

$T ::=$	Type expressions
$\quad t$	type variable
$\quad \mathsf{int}, \mathsf{string}, \mathsf{unit}$	base types
$\quad T\ \mathsf{chan}$	channel type
$\quad T_1 \to T_2$	arrow type
$v ::=$	Values (also used as Patterns)
$\quad x$	variable
$\quad 0, 1, \ldots, \mathsf{Alice}, \mathsf{Bob}, \ldots, ()$	constants for base types
$\quad l, c, n, \ldots$	names for functions, channels, nonces
$\quad f(v_1, \ldots, v_k)$	constructed term (when f has arity k)
$e ::=$	Expressions
$\quad v$	value
$\quad l\ v_1 \ldots v_k$	function application
$\quad \mathsf{match}\ v\ \mathsf{with}\ (\lvert v_i \to e_i)_{i<k}$	value matching
$\quad \mathbf{0}$	inert expression
$\quad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	value definition
$\quad \mathsf{let}\ (l_i\ x_0 \ldots x_{k_i} = e_i)_{i<k}\ \mathsf{in}\ e$	mutually-recursive function definition
$\quad \mathsf{type}\ (t_i = (\lvert f_{j_i}\ \mathsf{of}\ \widetilde{T}_{j_i})_{j_i<k_i})_{i<k}\ \mathsf{in}\ e$	mutually-recursive datatype definition
$\quad \mathsf{session}\ S = \Sigma\ \mathsf{in}\ e$	session type definition
$\quad S.r^b\ \widetilde{v}\ (v)$	session entry
$\quad s.p(e)$	session role (run-time only)
$E[\cdot] ::=$	Evaluation contexts
$\quad [\cdot]$	top level
$\quad \mathsf{let}\ x = E[\cdot]\ \mathsf{in}\ e_2$	sequential evaluation
$\quad s.p(E[\cdot])$	in-session evaluation (run-time only)
$P ::=$	Processes
$\quad e$	running thread
$\quad P \mid P$	parallel composition

The grammar defined for T, v, and e (except for $\mathbf{0}$ and the final three session-related constructs) generates a simple subset of ML; this is the language we call F. Type expressions T include constructed types t, base types int, string and unit, channel types T chan (with payload type T), and arrow types. Channel types T chan are included only for compatibility with the concrete F# language; our formal semantics is in fact untyped; this allows us to reason about arbitrary opponents. Values v include constants, functions, and terms built with type constructors. We assume given a finite set of principal constants, such as Alice and Bob, which are implemented as strings. Expressions

include an inert expression, $\mathbf{0}$, with no reduction, that abstractly represents thread termination.

Our language has four pi-calculus-like primitive functions: new, send, recv, and fork, to which we give a semantics below. It also has simple core libraries for functional data, including booleans, tuples, lists and functional records (as syntactic sugar for tuples). We omit their standard definitions.

Our language embeds the session types Σ of Section 2 as follows: F+S code can define named session types $S = \Sigma$ in expressions; F+S code can enter such sessions in a given role r using the expression $S.r^b\ \widetilde{a}\ (e)$.

In case r is the initial role of the session, the first argument $\widetilde{a}$ is a tuple of principals that binds all roles for the session and e is a message send; otherwise, $\widetilde{a}$ is the single principal that attempts to join the session in role r and e is a message handler. A *message handler* is a tuple (concretely implemented as a record) of continuations for each message that the role may receive in its current state, whereas a *message send* is an expression that yields a pair of a message to be sent and a message handler to receive the next message, if any. Their structure is illustrated in the example below.

Our syntax for session-entry expressions $S.r^b\ \widetilde{a}\ (e)$ in F+S coincides with the syntax for ordinary function application in F#, where S is the name of the module that implements the session, where r is a function provided by this module (for each role), and where e is built using ordinary datatype constructors. (The optional mark b in $S.r^b$ will be set to $\bullet$ to record that the session role is entered by the opponent; this mark is used only to specify session-integrity despite the compromise of some principals.)

At runtime, session-entry expressions reduce to active roles $s.p(e')$, where s ranges over unique session identifiers, p is the current role process, and e' is the current expression for the role: either a message-send expression or a message-handler value, depending on p.

We illustrate our syntax by giving an expression that initiates session S1 of Section 2 as a customer (file `customer.fs`):

```
let handle_accept a r = printf "The request has been accepted." in
let handle_reject a r = printf "The request has been rejected." in
S1.customer
    {customer=''Alice''; store=''Bob''}
    (Request("12 May 2007", { hAccept = handle_accept; hReject = handle_reject }))
```

In this code, the first argument to the customer role function instantiates the customer and store roles with principals Alice (the running principal) and Bob (some remote store). The second argument is the user code for the customer role: it defines a Request to be sent with payload `"12 May 2007"` and handlers (hAccept, hReject) for each of the two messages Accept and Reject that may be received next.

Semantics We define a labelled semantics for expressions, then for processes; we begin with the rules for plain F, then give the additional rules for sessions in F+S.

Our semantics has an explicit store, ranged over by ρ, that keeps track of generated names, defined functions, defined types, and (in F+S only) defined session types and information about running sessions. Syntactically, ρ contains names n; types $(t_i = (|f_{j_i}\ \text{of}\ \widetilde{T}_{j_i})_{j_i<k_i})_{i<k}$; function definitions $(l_i\ x_0 \ldots x_{k_i} = e_i)_{i<k}$; session types $S =$

Σ; and running sessions $s\ \widetilde{a}\ (\delta) : S$, where s is the session identifier, $\widetilde{a}$ are the principals for all roles, δ is a set of roles activated so far, and S is the session type variable name. We use $\uplus$ to express extensions of ρ with disjoint domain. (Before extending ρ, we may use renaming to obtain distinct constructor, function, type, and session type names).

Transitions are either unlabelled (implicitly labelled with the silent action) or labelled with either an input $z\ v$ or an output $\overline{z}\ v$, where z is either a channel name (e.g. c) or a session name concatenated with a message label (e.g. $s\overline{f}, sf$), and where v is a value. We let α, β range over labels, and let φ, ψ range over series of labels.

For F expressions (without sessions yet), our semantics is as follows:

$$
\begin{aligned}
\text{(APPLY)}\ & \rho, l\ v_0 \dots v_k \rightarrow_{\mathsf{e}} \rho, e\{x_0 = v_0; \dots; x_k = v_k\} \\
& \quad \text{when } (l\ x_0 \dots x_k = e) \in \rho \\
\text{(MATCH)}\ & \rho, \text{match } v \text{ with } (|v_i \rightarrow e_i)_{i<k} \rightarrow_{\mathsf{e}} \rho, e_0\gamma \\
& \quad \text{when } v = v_0\gamma \text{ for some substitution } \gamma \\
\text{(MISMATCH)}\ & \rho, \text{match } v \text{ with } (|v_i \rightarrow e_i)_{i<k} \rightarrow_{\mathsf{e}} \\
& \quad \rho, \text{match } v \text{ with } (|v_i \rightarrow e_i)_{0<i<k} \text{ otherwise} \\
\text{(LETVAL)}\ & \rho, \text{let } x = v \text{ in } e \rightarrow_{\mathsf{e}} \rho, e\{x = v\} \\
\text{(LETFUN)}\ & \rho, \text{let } (l_i\ x_0 \dots x_{k_i} = e_i)_{i<k} \text{ in } e \rightarrow_{\mathsf{e}} \\
& \quad \rho \uplus \{(l_i\ x_0 \dots x_{k_i} = e_i)_{i<k}\}, e \\
& \quad \text{up to renamings of } l_i \\
\text{(TYPE)}\ & \rho, \text{type } (t_i = \lambda_i)_{i<k} \text{ in } e \rightarrow_{\mathsf{e}} \rho \uplus \{(t_i = \lambda_i)_{i<k}\}, e \\
& \quad \text{where } \lambda_i = (|f_{j_i} \text{ of } \widetilde{T}_{j_i})_{j_i<n_i} \\
& \quad \text{up to renamings of } t_i, f_{j_i} \\
\text{(FRESH)}\ & \rho, \text{new } () \rightarrow_{\mathsf{e}} \rho \uplus \{n\}, n \\
\text{(SEND)}\ & \rho, \text{send } c\ v \xrightarrow{\overline{c}\ v}_{\mathsf{e}} \rho, () \text{ when } c \in \rho \\
\text{(RECV)}\ & \rho, \text{recv } c \xrightarrow{c\ v}_{\mathsf{e}} \rho, v \text{ when } c \in \rho
\end{aligned}
$$

This small-step semantics is standard; labels are used only to collect calls to send and recv; the rules (FRESH), (LETFUN), and (TYPE) simply extend ρ.

For processes, we have rules for forking new threads and communicating on both sides of a parallel composition.

$$
\text{(EVAL)}\ \frac{\rho, e \xrightarrow{\alpha}_{\mathsf{e}} \rho', e'}{\rho, E[e] \xrightarrow{\alpha}_{\mathsf{P}} \rho', E[e']} \qquad \text{(FORK)}\ \rho, E[\text{fork } l] \rightarrow_{\mathsf{P}} \rho, E[()] \mid l\ ()
$$

$$
\text{(COMMR)}\ \frac{\rho, P \xrightarrow{\overline{z}\ v}_{\mathsf{P}} \rho', P' \quad \rho', Q \xrightarrow{z\ v}_{\mathsf{P}} \rho'', Q'}{\rho, P \mid Q \rightarrow_{\mathsf{P}} \rho'', P' \mid Q'} \qquad \text{(PARR)}\ \frac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q}{\rho, R \mid P \xrightarrow{\alpha}_{\mathsf{P}} \rho', R \mid Q}
$$

$$
\text{(COMML)}\ \frac{\rho, Q \xrightarrow{\overline{z}\ v}_{\mathsf{P}} \rho', Q' \quad \rho', P \xrightarrow{z\ v}_{\mathsf{P}} \rho'', P'}{\rho, P \mid Q \rightarrow_{\mathsf{P}} \rho'', P' \mid Q'} \qquad \text{(PARL)}\ \frac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q}{\rho, P \mid R \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q \mid R}
$$

The communication rules (COMMR) and (COMML) combine matching send and receive actions; in case these actions are session actions, this may in turn may involve session transitions that update ρ (as shown below).

For sessions, we let σ range over $S.r^b\ \widetilde{a}$ and $s.p$, that is, session entries parameterized by principals as well as running sessions. We first define auxiliary transitions that keep track of running sessions in the store, written $\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p$, obtained from the role transitions $p \xrightarrow{\eta}_{\mathsf{r}} p'$ of Section 2 and with the same labels.

$$\text{(INIT)}\ \frac{p_0 \xrightarrow{\overline{g}}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau_i} = p_i)_{i<n} \in \rho \quad s \textit{ fresh}}{\rho, S.r_0^b\ (a_i)_{i<n} \xrightarrow{\overline{g}}_{\mathsf{s}} \rho \uplus \{s\ (a_i)_{i<n}\ \{r_0\} : S\}, s.p'} \qquad \text{(STEP)}\ \frac{p \xrightarrow{\eta}_{\mathsf{r}} p'}{\rho, s.p \xrightarrow{\eta}_{\mathsf{s}} \rho, s.p'}$$

$$\text{(JOIN)}\ \frac{p_j \xrightarrow{f}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau_i} = p_i)_{i<n} \in \rho \quad \rho' = \rho \uplus \{s\ (a_i)_{i<n}\ \delta : S\}}{\rho', S.r_j^b\ a_j \xrightarrow{f}_{\mathsf{s}} \rho \uplus \{s\ (a_i)_{i<n}\ (\delta \uplus \{r_j\}) : S\}, s.p'}$$

Rule (INIT) initiates a session, adding a new record $s\ (a_i)_{i<n}\ \{r_0\} : S$ to ρ with s being a freshly generated session name. Rule (JOIN) requires that (1) r_j for some $j < n$ is a role for the session S; (2) S is the session type of s; (3) the set δ of already-running roles for s does not contain r_j; and (4) the joining principal a_j matches the principal for r_j in s. The label f records the first input label for p_j according to S.

For sessions in expressions (hence in processes), we have:

$$\text{(SESSION)}\ \rho, \text{session}\ S = \Sigma\ \text{in}\ e \rightarrow_{\mathsf{e}} \rho \uplus \{S = \Sigma\}, e \text{ up to renamings of } S$$

$$\text{(SENDS)}\ \frac{\rho, \sigma \xrightarrow{\overline{g}}_{\mathsf{s}} \rho', s.p \quad \text{safe } \sigma}{\rho, \sigma\ (g(\widetilde{v}), w) \xrightarrow{s\overline{g}\ \widetilde{v}}_{\mathsf{e}} \rho', s.p\ (w)}$$

$$\text{(RECVS)}\ \frac{\rho, \sigma \xrightarrow{g}_{\mathsf{s}} \rho', s.p \quad s\ \widetilde{a}\ \delta : S \in \rho \quad \text{safe } \sigma}{\rho, \sigma\ (w) \xrightarrow{sg\ \widetilde{v}}_{\mathsf{e}} \rho', s.p\ (w.g\ \widetilde{a}\ \widetilde{v})} \qquad \text{(ENDS)}\ \rho, s.0\ (v) \rightarrow_{\mathsf{e}} \rho, v$$

where the predicate safe σ (defined only in Section 4) depends on the principal that enters the session. Rule (SESSION) adds a session type definition to ρ; Rules (SENDS) and (RECVS) enable role processes to send and receive messages using the session transitions; Rule (ENDS) returns the final value computed by a role process.

4 Libraries for cryptography and principals

In this section, we describe the design and interfaces of our libraries for cryptography and principals, coded as F# modules. We follow the approach of Bhargavan *et al.* [4] and provide a symbolic implementation in addition to the standard concrete implementation of these libraries. The symbolic implementation, written in the formal subset F of F#, is an important part of our security model. Its code is listed in Appendix C. (Formally, an F# module implementation M is just an expression context that binds types, session types, values, and functions; we write $M\ M'$ as syntactic sugar for $M[M'[\_]]$.)

Cryptography The cryptographic library includes the following types and functions, plus a few auxiliary formatting functions such as concat and utf8.

```
type bytes                                  val genskey: name → keybytes
type keybytes                               val genvkey: keybytes → keybytes
val nonce: name → bytes                     val sign: bytes → keybytes → bytes
val hash: bytes → bytes                     val verify: bytes → bytes → keybytes → bool
```

It has abstract types bytes for bitstrings and keybytes for cryptographic keys, and functions for constructing messages: nonce takes a (typically fresh) name and returns a nonce; hash returns the cryptographic hash of a message; genskey returns the signing

key associated with a name (used as a seed); genvkey returns the verification key associated with a signing key; sign signs a message using a signing key, and verify verifies a signature on a message using a verification key.

The concrete implementation of this library uses standard cryptographic algorithms. For instance, the datatype bytes is implemented as a byte array, and sign is implemented as an asymmetric signing function on the hash of the message (RSA-SHA1).

The symbolic implementation, on the other hand, uses algebraic datatypes and datatype constructors to model cryptographic operations. For example, the type bytes is defined as an algebraic datatype, and sign is implemented as the application of a binary constructor Sign that represents signed bytes. (Both bytes and keybytes types are abstract in the interface, and hence values of these types can be accessed only through the exported functions, preventing e.g. trivial key leakage by pattern matching on signatures.)

Executing code linked with our symbolic libraries is useful for debugging; Appendix D displays a symbolic run for an example. More importantly, the symbolic implementation encodes our formal model of cryptography that is used to establish our security results in the subsequent sections. Specifically, we consider a variant of the standard Dolev-Yao threat model: the opponent can control corrupted principals (that may instantiate any of the roles in a session), intercept, modify, and send messages on public channels, and perform cryptographic computations. However, the opponent cannot break cryptography, guess secrets belonging to compliant principals, or tamper with communications on private channels. (We rely on private channels only for simplicity; we could use instead, for instance, message authentication codes.) For example, the symbolic implementation ensures that 'verify m (sign m′ k) vk = true' only when m = m′ and vk = genvkey k.

Principals This library manages principals and their data; our implementation uses it to exercise the two privileges associated with the principals that play session roles, that is, signing values and receiving messages. Principals are just strings. (For clarity we use the type alias principal instead of type string.) The interface contains:

```
val skey : principal → keybytes
val vkey : principal → keybytes
val psend : principal → bytes → unit
val precv : principal → bytes
val safe : principal → bool
val psend• : (principal ∗ bytes) chan
val chans• : (principal ∗ bytes chan) list
val skeys• : (principal ∗ bytes) list
```

Functions skey and vkey return the signing and verification keys of a principal, respectively. (In the concrete implementation, we fetch keys from a local X.509 store, and return an error if no certificate is available.) Functions psend and precv provide message delivery with replay protection (explained below): 'psend a v' asynchronously sends message v to principal a, whereas 'precv a' receives a message sent to a. Calling 'skey a' and 'precv a' is a's privilege.

In the model, we assume a fixed, finite population of principals and an arbitrary but fixed predicate safe that indicates whether a principal is compliant or possibly corrupted. This predicate is used only to specify the security properties that hold for compliant principals—clearly, our implementation could not guarantee the security of principals whose signing keys are compromised. To this end, in our semantics, only safe

principals may enter a session in compliant code, and only unsafe principals may enter a session in opponent code. Formally, in rules (SENDS) and (RECVS), we let safe σ hold if and only if either $\sigma = S.r\ (a_i)_{i<n}$ and safe a_0, or $\sigma = S.r^\bullet\ (a_i)_{i<n}$ and not safe a_0, or $\sigma = s.p$.

Accordingly, opponent code is not given direct access to psend, precv and skey. Instead, it is given a channel psend$^\bullet$ for sending messages to safe principals, a list chans$^\bullet$ of channels to receive messages sent to unsafe principals, and a list skeys$^\bullet$ of signing keys belonging to unsafe principals. Using these, the attacker can receive messages sent to any unsafe principal and sign any value on their behalf. Hence, the initial knowledge of our Dolev-Yao opponent (called K in Section 7) consists of the values psend$^\bullet$, chans$^\bullet$ and skeys$^\bullet$, and all the functions above except for psend, precv and skey.

Anti-replay cache Like any protocols with responder roles, our protocols rely on dynamic anti-replay protection mechanisms for the messages that may cause principals to join a session, that is, the first messages they may receive in their roles.

To prevent such replays, each principal maintains a cache that records pairs of session identifiers and roles for all sessions it has joined so far. The cache for principal a is used only to filter incoming messages through the call to an auxiliary function 'antireplay' that can determine from the message header whether the message may need replay protection (by checking its header) and, when it is the case, which cache entry is associated with the message. If the message does not need replay protection, it is transmitted; otherwise, if the cache entry already occurs in the cache for a, the message is ignored; otherwise, the message is transmitted and the entry is added to the cache. The code of psend, precv and antireplay in the symbolic implementation is listed in Appendix C. This simple mechanism is thus verified as part of our formal model. Concretely, it may be refined using standard, timestamp-based techniques to bound the size of the cache while preserving its correctness.

5 Protocol outline

This section outlines the security protocol used to enforce session integrity; Section 6 describes its compiled implementation. We present our protocol (the third protocol below) as a refinement of simpler, intuitively secure protocols (the first and second protocols below). Exploiting the session structure and the implementability condition of Section 2, we obtain a final, optimized protocol with compact messages and minimal message processing.

The protocols all implement sends and receives by converting them to and from low-level bytes messages that consist of a session identifier, a payload, and a series of signatures (depending on the protocol, as described below). The identifier is computed as $s = \text{hash}(D\ \widetilde{a}\ N)$, where $D\ \widetilde{a}\ N$ is the tagged concatenation of $D = \text{hash}(S{=}\Sigma)$, a digest of the whole named session-type definition (type_digest: bytes in the implementation); the principals $\widetilde{a}$ assigned to the session roles; and a nonce N freshly generated by the initiator. Every initial message also includes D, $\widetilde{a}$, and N, so that its recipient can verify the identifier hash.

First protocol: signing the full session history In order to prevent any misbehaviour from any of the principals participating in a session, every message may conservatively include a record of the whole session history, countersigned by the sender of every message that extends the session. Every receiver can then verify the validity of incoming messages by replaying the received history on the session graph and verifying all its signatures.

Although intuitively correct, this solution is inefficient, as it requires both senders and receivers to do significant work, since session runs (and hence their records in messages) may be arbitrarily long in the presence of cycles.

Second protocol: signing message labels Since the session type is statically known and Property 2 of Section 2 ensures that every label has unique source and target nodes, each sender may simply sign the message label, rather than countersign the whole session record. Thus, every sender may forward previously-signed labels and append its own signed label to every message. Specifically, every message now carries a series of cryptographic signatures, each computed as $ts = \mathrm{sign}(s\ f\ t, \mathrm{skey}(a))$ where $s\ f\ t$ is the concatenation of the session identifier s, the message label f, and a logical timestamp t and where a is the principal assigned to the sending role of f, determined by s. The timestamp disambiguates signatures for labels occurring in cycles, which may be signed several times within a session; when receiving a message, a series of signatures is accepted only if they have increasing timestamps larger than the last-received message.

Although session records are now more compact, and their processing may be partially cached, receivers still need to dynamically replay session histories.

Third protocol: signing visible labels We can entirely avoid graph computations at runtime, and rely instead on a static notion of *visibility* between nodes and labels.

Let $\widetilde{g}$ be the sequence of labels on a given path from the initial node m_0 to a node m with role r. Let $\widetilde{f}$ be the sequence of labels obtained from $\widetilde{g}$ by erasing every label g (1) whose sending role is r; or (2) that is followed by a label whose sending role is either r or g's sending role. (Thus, $\widetilde{f}$ retains the last label sent by every role other than r, if any, along the path $\widetilde{g}$.) We then say that $\widetilde{f}$ is *visible* from m.

For example, for session (c) of Figure 1, the bottom-right node has a single visible sequence of labels, Accept-Confirm; the central node has two visible sequences, Request-Contract (along the initial path) and Change (through the cycle). Relying on visibility information computed at session-type compile-time, we obtain an efficient protocol with compact messages. To send a message with label f from node m to m' in the session graph, we pre-compute the series of labels $\widetilde{g}f$ that is visible from m' on a path with final label f. The message for f then includes the corresponding series of signatures, consisting of signatures for $\widetilde{g}$ previously received in messages from other roles, plus a new signature for f computed by the sender. Conversely, to verify a message received at node m, we pre-compute all series of visible labels at m, and accept a message only if it is well-formed and has valid signatures that match a series of visible labels. Hence, message size overheads and receiver checks are statically bounded by the number of roles.

6 Compiler implementation

In this section we present a translation from the session definitions of Section 2 to generated code for each of their roles, built on top of the libraries of Section 4. For a given valid session, we describe the generated interface, then present the generated protocol, and finally provide its implementation. To illustrate the translation, Appendix D lists excerpts from the code generated for the example session of Figure 1(c).

Generated session-type interface We first generate type declarations, including a record type principals that maps roles to principals and, for each role, types that reflect the message flow of a session from this role's viewpoint. We generate a type for each message sent or received by the role. For sending, we use a sum type with a constructor for each message that the role may send at this point, along with the corresponding continuation type. For receiving, we use a record type, with a message-handler for each message that the role may receive at this point. These types are mutually recursive when there is a cycle in the graph.

We omit a general definition, and list instead the types for session S2 of Section 2, with two roles (customer and store):

```
type principals = { customer: principal; store: principal }

type msg0 = Request of (string * msg1)
and msg1 = { hReject : principals → unit → unit;
             hOffer : principals → string → msg2}
and msg2 = Change of (string * msg1) | Accept of (unit * unit)

type msg3 = { hRequest : (principals → string → msg4) }
and msg4 = Reject of (unit * string ) | Offer of (string * msg5)
and msg5 = { hChange : (principals → string → msg4) ;
             hAccept : (principals → unit → string) }
```

For each role of the session, we also generate a session-entry function that inputs principal information and the user's message (or message handler). For session S2, these functions have the following types.

```
val customer: principals → msg0 → unit
val store: principal → msg3 → string
```

We rely on ordinary ML typing of the session-entry parameters against the generated types $\text{msg}i_r$ to ensure that the nested messages and handlers provided by the user will comply with role r for the whole session. Hence, inasmuch as all principals enter sessions by calling our typed interface, all their sessions will be correctly executed. In the rest of the section, we describe more dynamic implementation mechanisms that provide guarantees even when some principals are compromised.

Role implementation In our implementation, the dynamic state for each active role consists of a principal assignment prins, a nonce (used in the session identifier), a logical time (the timestamp of the last issued signature), and a record tsigs of the last-received verified signature for each role of the given session type Σ, if any. (In the following, the text in italics included in code specifies how the compiler produces that code.)

type tsig = { tstime: int; tsval: bytes }
type tsigs = { [r: tsig;] *for each* $(r : \widetilde{\tau} = p) \in \Sigma$ }
type state =
  { prins: principals; nonce: bytes; time: int; sigs: tsigs }

In addition, much like in code implementing control automata, we generate distinct, mutually-recursive functions indexed by series of labels, so that the current node and stored signatures for the role are always statically known when we generate the code for each of these functions. To generate a message with label f in a state where $\widetilde{g}$ denotes the series of labels for the signatures currently stored in tsigs, we implement:

val gen_$\widetilde{g}$_f: state $*$ $payload(f)$ $\rightarrow$ bytes
val gensig_p_f: state $\rightarrow$ bytes

The function gensig_p_f computes a signature ts for label f using the currently-stored time, as described above; gen_$\widetilde{g}$_f computes a message that carries some payload for f and includes a series of signatures for the labels visible by the intended receiver, with a last signature computed by gensig_p_f. To check that a received message contains valid signatures for the visible labels $\widetilde{g}'$ in a state with stored labels $\widetilde{g}f$, we also implement

val chk_$\widetilde{g}f$_$\widetilde{g}'$: state $*$ bytes $\rightarrow$ state $*$ $payload(\widetilde{g}')$

where f is the last label sent by the role, and can be omitted when $\widetilde{g}$ is empty (that is, when receiving a first message for the role) and $payload(\widetilde{g}')$ is the payload type for the last label of $\widetilde{g}'$, written $last(\widetilde{g}')$, as specified in the session type. The function chk_$\widetilde{g}f$_$\widetilde{g}'$ updates the state with the new received signatures and updates the stored time with the successor of the latest received timestamp.

For any path in the graph, there is a single active role r, which can send a message to a role r' with label selected from a set $\mathcal{F}$ that collects the possible outgoing labels at this particular node; moreover, we can pre-compute the series of stored labels $\widetilde{g}$ for this active role. For each such $\widetilde{g}$, our compiler generates the following sending and receiving functions:

for all reachable $\widetilde{g}$ *with corresponding* $r, r', \mathcal{F}$:
[[let rec|and] send_$\widetilde{g}$ st msg = match msg with
  for each $f \in \mathcal{F}$: [| f(v,w) $\rightarrow$
    let a' = st.prins.r' in let m = gen_$\widetilde{g}$_f st v in psend a' m ;
    if the next node is terminal: w *else:* recv_$\widetilde{g}f$ st w]]
for all reachable $\widetilde{g}$ *with corresponding* $r, r', \mathcal{F}$ *and for each* $f \in \mathcal{F}$:
[and recv_$\widetilde{g}f$ st w = let a = st.prins.r in let m = precv a in verify_$\widetilde{g}f$ st m w
 and verify_$\widetilde{g}f$ st m w = let path = visible_$\widetilde{g}f$ m in match path with
  for each $\widetilde{g}'$ *such that* $\widetilde{g}f + \widetilde{g}'$ *visible from a receiving node for* r:
  [| t_$\widetilde{g}'$ $\rightarrow$ let st,payload = chk_$\widetilde{g}f$_$\widetilde{g}'$ st m in
    if the next node is terminal: w.$last(\widetilde{g}')$ st.prins payload
    else: let next = w.$last(\widetilde{g}')$ st.prins payload in send_$\widetilde{g}$+$\widetilde{g}'$ st next]]

The function send_$\widetilde{g}$ takes two parameters, st and msg = f(v,w) for some label f in $\mathcal{F}$; it sends f(v) to r' and calls 'recv_$\widetilde{g}f$ st' with the next received message given by 'precv a'

and the message handlers w. (If the process terminates after the send, it simply returns w.) The function recv_$\widetilde{g}f$ st calls 'verify_$\widetilde{g}f$ st' on the message it has received. This call extracts from the message the partial history (i.e. the partial path) it contains and verifies that it matches one of the possible partial paths t_$\widetilde{g}'$ the role can expect to encounter in this state. Specifically, the function visible_$\widetilde{g}f$ matches the message against every acceptable partial history, pre-computed as the visible sequences at node $\widetilde{g}f$ (see Section 5). Finally, the incoming message m is checked to include the corresponding series of valid signatures, and send_$\widetilde{g}$+$\widetilde{g}'$ is invoked to send the next message in the updated state (or, if the role terminates after the receive, the function simply returns the value produced by w). Here, $\widetilde{f}$+$\widetilde{g}$ is the sequence of labels obtained from $\widetilde{f}\widetilde{g}$ by erasing from $\widetilde{f}$ any label that has the same sending role as a label in $\widetilde{g}$. If any test fails while processing the message, the session is stuck and yields the inert expression $\mathbf{0}$.

Relying on these definitions, our implementation exports functions $(r_i)_{i<n}$ and their types; these top-level functions rely on auxiliary functions init and join to initialize the session state when a role initiates or joins the session:

```
val init: principals → unit state
for the initiator role r0:
[let r0 prins msg =
  let st = init prins in send_∅ st msg ]
```

```
val join: bytes → unit state * bytes
for all other roles r:
[let r self w =
  let m0 = precv self in let st,m = join m0 in
  if st.prins.r = self then verify_∅ st m w]
```

7 Correctness results

In order to express and prove the correctness of our implementation, we first use a reduction and testing semantics and then, more precisely, use a labelled semantics that explicitly tracks all interactions with the opponent. This labelled semantics is also used to structure the proofs of our theorems, given in Appendix E.

We relate the behaviour of high-level processes, of the form $L\ \widetilde{S}\ U\ O$ with the F+S semantics to their implementations $L\ M_{\widetilde{S}}\ U\ O'$ with the F semantics, where

- L consists of the symbolic libraries (see Section 4);
- $\widetilde{S}$ is a series of session declarations; $M_{\widetilde{S}}$ is their implementation (see Section 6);
- U ranges over "user" code that may call the session interface but not Prins;
- O ranges over "opponent" code with access to the opponent interface of Prins, including recv$^\bullet$ and skey$^\bullet$ (but not recv and skey), and to $S.r^\bullet$ session entries;
- O' corresponds to O in the implementation, with similar assumptions. We assume that O' does not access $M_{\widetilde{S}}$—this entails no loss of generality, since O' may include its own copy of our implementation code.

Their code can use cryptography and exchange messages on shared channels. This reflects our intuition that U and O, O' may be located on different machines connected by a public network.

7.1 Testing semantics

Our first security theorem is stated in terms of may testing. Testing semantics have a long history, which can be traced back to the Morris equivalence for the lambda calculus [28]. As regards process calculi, may testing has been studied e.g. for CCS by

De Nicola and Hennessy [13]. Informally, by quantifying over any potential user code, we let user code internally implement any test on the behaviour of sessions and decide when to report the result of a test as a "failure"; then, we show that F configurations using our session implementations have no more test failures than F+S configurations.

As is customary in process calculi, we use a special channel ω to mark global failure. We say that a configuration $\emptyset, P$ may fail when $\emptyset, P \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$.

Theorem 1. *If $L\ M_{\widetilde{S}}\ U\ O'$ may fail in* F *for some O' where ω does not occur, then $L\ \widetilde{S}\ U\ O$ may fail in* F+S *for some O where ω does not occur.*

The theorem states that low-level F configurations, where sessions are implemented as described in Section 6, cannot deviate from high-level F+S configurations, where sessions are ideally followed as prescribed by the session semantics of Section 3. Hence, our session implementations preserve session integrity. The proof is given in Section E.7.

A counter-example without the implementability condition. We now show that Theorem 1 does not necessarily hold if we relax our secure implementability condition. For example, recall the session of Figure 3(a), which fails to satisfy Property 3. Assume that the principals for the client and officer are safe and run a single session with an unsafe principal for the store. As mentioned in Section 2, a low-level opponent O' implementing the store can attack the session by sending both Accept and Reject messages. The user code for the client and the officer can then communicate on some auxiliary channel, detect that they both have received a final message, then emit ω in protest. On the other hand, no high-level opponent running the store can cause the same user code to emit ω. We give below some concrete user code (U in Theorem 1) that conducts the test and reports the attack:

```
let pr = { client = "Alice"; server = "Eve"; officer = "Bob"; } in
let x = new() in
let alice () =
  let acceptbranch _ _ = send x "OK" in
  S.client pr (Request (42,{hAccept=acceptbranch})) in
let bob () =
  let rejectbranch pr' _ = if pr = pr' then let _ = recv x in send ω () in
  S.officer "Bob" {hReject=rejectbranch} in
fork bob; alice()
```

In this code, Alice plays the client role and Bob plays the officer role for a single run of the session. These safe principals synchronize using the side communication channel x when they receive both Accept and Reject messages. In that case, Bob fails with ω.

Session integrity and correspondence assertions. Session integrity enforces all causal relationships between message events. In cryptographic protocol analyses, such relationships are typically explicitly written as injective correspondences [36] between events issued by different parties. Intuitively, from a session graph, one can read a series of injective correspondences that hold in any session run. Theorem 1 then guarantees that every such correspondence holds for compliant principals.

To relate these two formulations of authenticity, suppose that before sending each message, each principal issues an event that indicates its current session history; and

suppose that the principal that receives the final message issues this event before terminating the session. As an example, consider a run of session (c) in Figure 1, with identifier id, where the principals a_c, a_s, and a_o play the roles C, S, and O, respectively. Let pr abbreviate the principal-assignment record $\{$ C = a_c; S = a_s; O = a_o; $\}$. Before sending the Contract message, a_o issues the event:

$$\text{SessionHistory}(a_o, \text{id}, pr, [\text{Request}, \text{Contract}])$$

This event indicates that the current session history at a_o for session id has the principal assignment pr and the session label history [Request,Contract].

We can extract from a session graph a set of correspondence assertions on event traces. Since malicious principals may lie about their history, we only make assertions about compliant session participants. First, for each compliant principal, we assert that the history events it may issue must correspond to one of the paths in the graph. Second, for each path followed in a session run, we assert that the histories at compliant principals must be consistent. Traditionally, correspondences do not have recursive operators, so they cannot be used to express properties of graphs with loops. In the following, we use an operator $*$ to express zero or more repetitions of a sequence, and hence, to finitely represent a loop. Still, we do not capture all session integrity properties, such as, for example, that two paths are mutually exclusive in any given session run.

We illustrate each type of assertion for our example. First, the history events issued by a_o are constrained as follows:

$$\begin{aligned}\forall a_c, a_s, a_o, \text{id} : \text{SessionHistory}(a_o, \text{id}, pr, H) \Rightarrow \\ H = [\text{Request}, \text{Contract}] \vee \\ H = [\text{Request}, \text{Contract}] @ [\text{Offer}, \text{Change}] * @ [\text{Reject}, \text{Abort}] \vee \\ H = [\text{Request}, \text{Contract}] @ [\text{Offer}, \text{Change}] * @ [\text{Accept}, \text{Confirm}] \vee \\ \text{Unsafe}(a_o)\end{aligned}$$

Here, @ stands for sequence concatenation; and the event Unsafe(a) indicates that the principal a is compromised (that is, 'safe a' returns false). This assertion says that the SessionHistory events issued by a compliant a_o must be in one of three forms, corresponding to the three nodes where O appears in the graph.

Second, whenever a_o accepts an Abort message, the histories at a_c, a_s, and a_o must be related as follows:

$$\begin{aligned}&\forall a_c, a_s, a_o, id : \\ &\quad \text{SessionHistory}(a_o, \text{id}, pr, [\text{Request}, \text{Contract}] @ H' @ [\text{Reject}, \text{Abort}]) \Rightarrow \\ &\qquad \text{Unsafe}(a_o) \vee \\ &\qquad ((\text{SessionHistory}(a_c, \text{id}, pr, [\text{Request}, \text{Contract}] @ H' @ [\text{Reject}, \text{Abort}]) \vee \text{Unsafe}(a_c)) \wedge \\ &\qquad\ (\text{SessionHistory}(a_s, \text{id}, pr, [\text{Request}, \text{Contract}] @ H' @ [\text{Reject}]) \vee \text{Unsafe}(a_s)))\end{aligned}$$

This assertion says that if the history at a compliant a_o corresponds to a path ending in Abort, then the histories at a_c and a_o must be consistent with this path, unless they are compromised.

Since we can write user code that test these properties, Theorem 1 guarantees that both these assertions, and others read from the session (c) in Figure 1, are preserved by our compiled implementation of the session.

7.2 Labelled semantics for the opponent

Our second security theorem is more precise but also more complex; it provides an explicit operational correspondence between high-level and low-level runs. To this end, we extend our labelled semantics so that it can represent the adversary as an abstract environment, rather than a top-level program (O and O' in Theorem 1).

In the transitions of Section 3, we do not maintain scope for the sessions and values available to the opponent, and maintain instead a global store ρ, using interfaces to ensure that the opponent code cannot access some values. Instead, we now introduce labelled transitions with an abstract environment, representing some unknown opponent, and keep track of the values and sessions available to that environment. As in Section 3, we define two variants of the labelled transition system, for F and for F+S.

We let K range over the knowledge and capabilities available to the environment. Initially, K contains the opponent interfaces of our libraries, including the verification keys of all principals and the signing keys and channels of unsafe principals, some fresh names, as well as any value exported by U. In high-level configurations, K also contains the signing keys of all safe principals and a supplementary set of nonces, $\mathcal{N}$. (These do not give any added power to the high-level attacker but are technically convenient to allow our soundness result, Theorem 2, to be stated without being encumbered by key and nonce renamings.)

The set K grows as the opponent obtains new values in labelled output transitions. We let $\mathsf{Val}(K, \rho)$ represent values computed from K by repeatedly applying type constructors in ρ to the elements of K and constants in base types.

In F+S, for every running session recorded in ρ, Ks also record the state of the roles instantiated to unsafe principals, written $s.p$. (Hence, if ρ initially has no sessions, K initially records no session states.)

We define auxiliary notations to access these session states: we write $K = K'[\sigma]$ either when K records the state $\sigma = s.p$ for an running session s of ρ, or when $K = K'$ and $\sigma = S.r_i^\bullet\ \tilde{a}$ with safe $a_i = \mathsf{false}$.

We define transitions for F and F+S configurations with K as follows:

$$\text{(KAPPLY)}\ \frac{l_i, v_0, \ldots, v_k \in \mathsf{Val}(K,\rho) \quad (l_i\ x_0 \ldots x_k = e) \in \rho \quad \rho, l_i\ v_0 \ldots v_k \longrightarrow_{\mathsf{e}}^{*} \rho, w}{K, \rho, P \rightarrow_{\mathsf{K}} K \cup \{w\}, \rho, P}$$

$$\text{(KSEND)}\ \frac{\rho, P \xrightarrow{\overline{c}\ v}_{\mathsf{P}} \rho, P' \qquad c \in K}{K, \rho, P \xrightarrow{\overline{c}\ v}_{\mathsf{K}} K \cup \{v\}, \rho, P'} \qquad \text{(KRECV)}\ \frac{\rho, P \xrightarrow{c\ v}_{\mathsf{P}} \rho, P' \quad c, v \in \mathsf{Val}(K,\rho)}{K, \rho, P \xrightarrow{c\ v}_{\mathsf{K}} K, \rho, P'}$$

$$\text{(KSTEP)}\ \frac{\rho, P \rightarrow_{\mathsf{P}} \rho', P'}{K, \rho, P \rightarrow_{\mathsf{K}} K, \rho', P'} \qquad \text{(KSENDS)}\ \frac{\rho, P \xrightarrow{s\overline{g}\ v}_{\mathsf{P}} \rho', P' \quad K, \rho' \xrightarrow{sg}_{\mathsf{o}} K', \rho''}{K, \rho, P \xrightarrow{s\overline{g}\ v}_{\mathsf{K}} K' \cup \{v\}, \rho'', P'}$$

$$\text{(KRECVS)}\ \frac{K, \rho \xrightarrow{s\overline{g}}_{\mathsf{o}} K', \rho' \quad \rho', P \xrightarrow{sg\ v}_{\mathsf{P}} \rho'', P' \quad v \in \mathsf{Val}(K,\rho)}{K, \rho, P \xrightarrow{sg\ v}_{\mathsf{K}} K', \rho'', P'}$$

$$\text{(OSTEP)}\ \frac{\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p}{K[\sigma], \rho \xrightarrow{s\eta}_{\mathsf{o}} K[s.p], \rho'} \qquad \text{(OCOMM)}\ \frac{K, \rho \xrightarrow{s\overline{g}}_{\mathsf{o}} \xrightarrow{sg}_{\mathsf{o}} \xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}{K, \rho \xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}$$

Rule (KAPPLY) lets the environment apply functions; its hypotheses require that both the function and its arguments be known to the environment, and that the function

be pure (since ρ is left unchanged). Additionally, function with side effects, or calls from P to the environment, may be modelled using channel-based communications.

Rules (KSEND) and (KRECV) represent channel-based communications with the environment. Rule (KSTEP) enables P to make silent progress.

Rules (KSENDS) and (KRECVS) represent session steps in the source semantics. They rely on auxiliary transitions $K, \rho \xrightarrow{\alpha}_{\mathsf{o}} K', \rho'$ that represent session operations in the environment. Rule (OSTEP) performs session steps for roles in the environment (inits, joins, sends, and receives). In addition, Rule (OCOMM) accounts for communications between roles in the environment, which may advance the session without involving compliant user code.

The lemma below relates the reduction-based and labelled-based semantics in F+S. A similar lemma holds in F. These lemmas enable us to prove Theorem 1 using inductions on traces. We write $\stackrel{\varphi}{\Rightarrow}_{\mathsf{K}}$ for a series of transitions that consist of observable transitions (with series of labels φ) interleaved with any number of silent transitions. The proof is given in Section E.6.

Lemma 1. *We have transitions $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \stackrel{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ for some fresh names $\widetilde{n}$ where $\omega \notin fn(\psi)$ if and only if $\rho, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ for some process O that does not contain ω, does not match on constructors in ρ, that calls only pure functions of ρ, and whose values defined in ρ are all included in $\mathsf{Val}(K, \rho)$.*

Relating abstract and compiled sessions at runtime The state of a role implementation in F is not entirely determined by high-level configurations $H = K, \rho, P$: in addition to the session definition S and the session state s recorded in ρ, and to active sessions $s.p(e)$ within P, the implementation state records the time, the session nonce, and a sequence of signed timestamped labels $\widetilde{g}$. We let T record this additional information. For every principal a:

- $T.\mathit{cache}(a)$ is the content of the anti-replay cache of principal a: a set of pairs of session identifiers and roles $(\widetilde{sid}, r)$.

For every running session $s\ (a_i)_{i<n}\ \{\widetilde{r}\} : S$ in ρ:

- $T.\mathit{nonce}(s)$ is a term N_s in K;
- $T.\mathit{path}(s)$ is an initial path $\widetilde{f}$ of S, decorated with strictly-increasing integers $\widetilde{j}$, ending by a label sent or received by a safe principal;
- $T.\mathit{stuck}(s)$ is the set of roles of the session that have received a bad input, if any. In that case, the roles are stuck, and so is the whole session.

Finally, T provides an infinite (and co-infinite) set of fresh supplementary names, $\mathcal{N}$, which is formally used to supply a fresh nonce to the high-level environment whenever the low-level environment receives a fresh session nonce.

We say that a signature is exported by T when it is a signature of a running session s of H, with the label and timestamp recorded in $T.\mathit{path}(s)$, signed by a safe principal, and such that this label is visible from a role on that path instantiated to an unsafe principal. Informally, these are the signatures that the opponent has received. (They are not known to user code unless the opponent explicitly pass them to the user.)

We define the translation $[\![K, \rho, P]\!]_T$ from F+S configurations to F configurations, as follows: to translate K:

– we replace session records $s.p$ with the session nonces $T.\mathit{nonce}(s)$;
– we add the signatures exported by T;
– we remove the signing keys of all safe principals and the supplementary nonces $\mathcal{N}$.

To translate the store ρ:

– we replace session type definitions $\widetilde{S}$ with the types and function definitions of $M_{\widetilde{S}}$;
– we remove the session entries and the nonces of $\mathcal{N}$ not in the image of $T.\mathit{nonce}$.

To translate the process P:

– we add the process F = forward ();
– for each principal a, we add the process $P_a = \text{send } \mathit{cache}_a\ T.\mathit{cache}(a)$, where cache_a is the channel of $\rho_{L\widetilde{S}}$ that holds the state of the antireplay cache for a. (The forwarding code and cache management are defined in Appendix C.)
– we translate all running session roles within P as follows:

$$
\begin{array}{lll}
[\![s.p(w)]\!]_T = \mathbf{0} & & \text{if } p\text{'s role is in } T.\mathit{stuck}(s) \\
[\![s.p(w)]\!]_T = \text{S.recv\_}\widetilde{g}\ st\ [\![w]\!]_T & & \text{else if } p \text{ is an input} \\
[\![s.p(e)]\!]_T = \text{let } x_s = [\![e]\!]_T \text{ in S.send\_}\widetilde{g}\ st\ (x_s) & & \text{else if } p \text{ is an output} \\
[\![s.p(e)]\!]_T = [\![e]\!]_T & & \text{else if } p \text{ is } \mathbf{0}
\end{array}
$$

where $\widetilde{g}$ and st are computed from $T.\mathit{nonce}(s)$ and $T.\mathit{path}(s)$. Although we do not translate expressions of the form $S.r\_$, they are now interpreted as function calls, rather than primitive session entries.

Implementation soundness for labelled transitions We let $\rho_{L\widetilde{S}}$ be defined by the deterministic reductions $\emptyset, L\ \widetilde{S}\ [\_] \rightarrow_{\mathsf{P}}{}^{*}\ \rho_{L\widetilde{S}}, [\_]$ (that is, $\rho_{L\widetilde{S}}$ is the store defining our libraries and protocol implementations, after initialization).

An F+S configuration $H = K, \rho, P$ is *valid* with respect to T when

1. ρ includes $\rho_{L\widetilde{S}}$;
2. every value of K defined in $\rho_{L\widetilde{S}}$ is built from the library interfaces of Section 4;
3. every session $s\ (a_i)_{i<n}\ \{\widetilde{r}\} : S$ in ρ has a running session $s.p$ in P for each safe principal plus a running session $s.p$ in K for each unsafe principal, such that their roles either send or receive on $T.\mathit{path}(s)$ with role process p;
4. the signing keys of safe principals occur in P only within signatures exported by T;
5. the other names of the Prins library do not occur in P;
6. K includes all signing keys (including those for safe principals);
7. K and ρ include the supplementary set of nonces $\mathcal{N}$ (discussed above).

An F configuration W is a *valid implementation* of an F+S configuration H when H is valid and $W = [\![H]\!]_T$ for some low-level state T, up to functional steps. Further, W *has no bad inputs* when T has no bad input record for any session.

A low-level trace with labels φ is a direct translation of a high-level trace with labels ψ when φ is ψ after replacing all session inputs and outputs $s\eta\ v$ of ψ with inputs on channel psend$^\bullet$ and outputs on channels in chans$^\bullet$, respectively. We also consider low-level traces where some low-level inputs have been discarded (such as message replays) or have not been processed yet (such as inputs that have passed anti-replay

filtering): a low-level trace is a translation of a high-level trace when it is a direct translation interleaved with additional inputs on channel psend$^\bullet$.

At low-level, an attacker may copy signatures that it receives during session communication from good participants and resend these signatures back via non-session channels. In order to mirror these channel transitions at high-level, we need to allow the high-level environment to construct such signatures since it cannot acquire them as a biproduct of high-level session communication (which does not use any cryptography at all). In valid configurations, we therefore require that the high-level environment K contains the signing keys of all safe principals. The translation removes most of this supplementary knowledge, to prevent the low-level attacker from knowing the signing keys of safe principals. (We do not render the high-level attacker more powerful by providing it with this supplementary knowledge, which plays no part in the high-level semantics: at high level, we make no use of cryptography, so for any sequence of high-level labelled transitions, we could "alpha convert" all the configurations and labelled transitions by renaming the supplementary keys and nonces to entirely unrelated ones.)

Our second security theorem states that all low-level events on the network can be explained by the high-level semantics, thereby ensuring that attackers do not gain anything from trying to break sessions at low level. The proof appears in Section E.5.

Theorem 2. *Let W be a valid implementation of H. For all transitions $W \stackrel{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, there exist a valid implementation W° of H° and a translation φ of ψ such that*

$$H \stackrel{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ \qquad W \stackrel{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{K}} W'' \qquad W' \rightarrow^*_{\mathsf{K}} W''$$

In the statement of the theorem, the first series of transitions gives high-level transitions that simulate the low-level transitions. However, due to silent implementation steps (e.g. security checks), the resulting low-level configurations W' and W° may differ slightly; the second and third series of transitions relate them, showing that both may perform silent step leading to the same low-level configuration.

Conversely, our final theorem expresses functional completeness of our implementation of high-level transitions, under the assumption that no session has silently failed after receiving a bad input. The theorem guarantees, in particular, that our implementation is functionally correct in the absence of an adversary. The proof appears in Section E.1.

Theorem 3. *Let W be a valid implementation of H with no bad inputs. For all transitions $H \stackrel{\psi}{\Rightarrow}_{\mathsf{K}} H'$ in* F+S *such that ψ contains neither any signing key of a safe principal nor any elements of $\mathcal{N}$, there exist a valid implementation W' of H' with no bad inputs and a direct translation φ of ψ such that $W \stackrel{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in* F.

The statement of the theorem excludes from consideration transitions involving signing keys of safe principals or nonces in the $\mathcal{N}$. These keys and most of these nonces are not known to the low-level attacker and therefore high-level transitions containing them could not necessarily be matched at low-level.

8 Experimental results

In this section, we discuss our prototype complier and present a case study in which we model a conference management system using sessions—to our knowledge, the

largest session specification ever formalized in the literature. We describe the session as a graph and as a local roles' declaration written by the user. From this specification, the compiler generates a type interface, which we illustrate with sample user code that may be compiled against that interface to govern the behaviour of the individual roles. We then discuss potential attacks on session integrity which are prevented by the generated cryptographic protocol.

The code described here may be directly compiled by our prototype, `s2ml`, and executed over the network. In order to illustrate the compilation process, we interleave the discussion of the case study with commentary on the operation of `s2ml`. We conclude by presenting execution benchmarks for the case study.

Overview of the compiler Our prototype compiler, `s2ml`, is available at `http://msr-inria.inria.fr/projects/sec/sessions`. The distribution provides sample code, including the code for the case study below.

The compiler reads session declarations, that is, `.session` files like `S1.session` in Figure 2, and works as follows:

- Initially, each local session declaration is transformed out to a global, graph representation (see Appendix A). These graphs are checked to respect correctness and security conditions (properties 1–3 from Section 2). As intermediate output to help visualize the sessions, `s2ml` can output its DOT [17] graph file (e.g., `S1.dot` for `S1.session`).
- Then, `s2ml` generates F# modules (along with their interfaces) for each specified session (e.g., `S1.fs` and `S1.fsi` for `S1.session`).
- Using these generated interfaces and modules, programmers can develop user code using the sessions, then call the F# compiler to typecheck them and produce executable code for each role of the sessions

8.1 Application: a (simplified) conference management system

We now illustrate each of the phases of compilation for our case study, a session for a conference management system (CMS).

Global description Figure 4 shows the graph of a CMS session. There are three roles: pc (the program committee), author, and confman (the submission manager). All messages carry as a payload either a string value (which is used for the call for papers, paper submissions, and so on), or a unit value, when no payload is necessary.

The session proceeds as follows. Initially, the program committee pc sends a call for papers message, Cfp, to the prospective author. (Our session specifications exclude broadcast, so we assume here that the Cfp is sent to a single author, already chosen by the program committee. In any event, we could replicate the program committee to start other sessions with other prospective authors.)

The author then uploads a draft by sending an Upload message to the conference manager confman, which checks whether the draft meets the conference format (e.g. style and length compliance). If the format is invalid, the conference manager replies to the author with a BadFormat message providing some explanation; at this point we

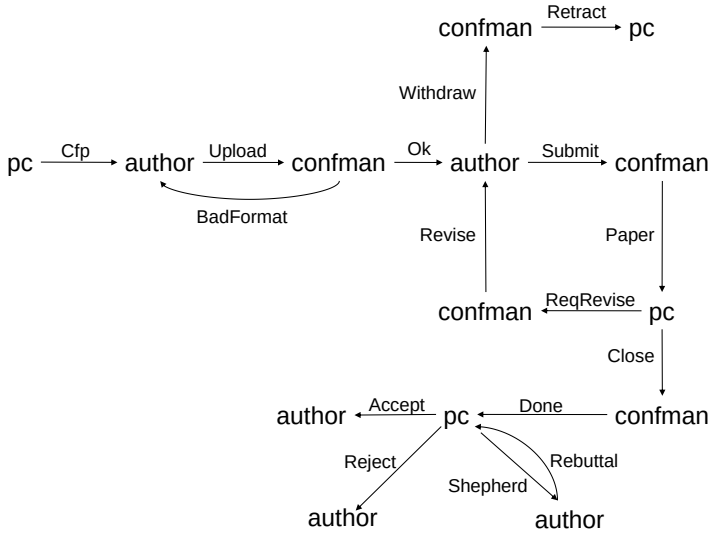

Fig. 4. A conference management system (CMS): Global graph

have a loop in which the author can fix the draft and try again. Eventually the format is valid, and the conference manager replies with an Ok message.

Now the author can submit a paper by sending a Submit message to the conference manager. Alternatively, it can choose to refrain from submitting a paper by sending a Withdraw message, which the conference manager communicates to the program committee by sending a Retract message. If the author indeed submits a paper, the conference manager forwards it to program committee, which then evaluates it. The program committee can ask the author to revise the paper, by sending a ReqRevise message to the conference manager which in turn sends a Revise message to the author. This phase can loop until eventually the program committee reaches a decision, and asks the conference manager to stop receiving revisions by sending a Close message.

The conference manager answers with a Done message, and then the program committee can notify the author of the result, possibly enclosing reviews of the paper. The notification is either an acceptance of the paper (an Accept message), or a rejection (a Reject message), or an exceptional decision to 'shepherd' the paper (a Shepherd message), in which case the author can support her submission by sending a Rebuttal. This can again loop until the program committee decides a final verdict, i.e. either acceptance or rejection the paper. In the case of acceptance, the author sends the program committee a final version of the paper.

As explained in Section 2, not all expressible sessions is safe to implement, as they also have to meet e.g. Property 3. We can check on Figure 4 that this indeed the case for our example: at every fork in the graph, the two edges lead to the same role (e.g., when the conference manager receives an Upload, it always replies to the author).

```
session CMS =
  role pc:string =
    !Cfp:string; mu start.
    ?( Paper:string; !(Close:unit; ?Done:unit; mu discuss.
      !(Accept:string; ?FinalVersion:string
                     + Reject:string
                     + Shepherd:string; ?Rebuttal:string; discuss)
      + ReqRevise:string; start)
    + Retract:unit)
  role author:string =
    ?Cfp:string; mu reformat. !Upload:string;
      ?(BadFormat:string; reformat
      + Ok:unit; mu submission.
        !(Submit:string; mu discuss.
          ?(Accept:string;
            !FinalVersion:string
            + Reject:string
            + Shepherd:string; !Rebuttal:string; discuss
            + Revise:string; submission)
        + Withdraw:unit))
  role confman:string =
    mu uploading. ?Upload:string;
    !(Ok:unit; mu waiting.
      ?(Submit:string; !Paper:string;
        ?(Close:unit; !Done:unit + ReqRevise:string; !Revise:string; waiting)
      + Withdraw:unit; !Retract:unit)
    + BadFormat:string; uploading)
```

Fig. 5. Session specification (file `CMS.session`)

Local processes Figure 5 presents the counterpart of the CMS graph from Figure 4 in terms of local roles. We illustrate user code by describing in detail the behaviour of the author role. From the author's point of view, the session starts by receiving a Cfp message. A recursion point called reformat is created, and then the author checks the paper by sending an Upload message. If a BadFormat message is received, execution jumps back to the reformat point. If an Ok message is received, the author sets a recursion point called submission and then chooses to either send a Submit or a Withdrawal message. For the latter, execution ends. For the former, another recursion step discuss is set, and several messages can be expected: either an Accept, in which case the author ends by sending a FinalVersion, or a Reject which also ends execution, or a Shepherd message to which the author replies with a Rebuttal and then jumps back to discuss; finally, a Revise message may also be received, in which case the author jumps back to submission.

Generated types for the conference management system We run `s2ml` with the CMS example of Figure 5 for the CMS session, on file `CMS.session`. This produces files

`CMS.fs` and `CMS.fsi`. The interface `CMS.fsi` contains a specialized principals record plus generated types and functions for each role (here we show only the ones for the author role):

```
type principal = string
type principals = {pc:principal; author:principal; confman:principal}
type msg9 = { hCfp : (principals → string → msg10)}
and msg10 = Upload of (string * msg11)
and msg11 = { hBadFormat : (principals → unit → msg10) ;
              hOk : (principals → unit → msg12)}
and msg12 = Submit of (string * msg13) | Withdraw of (unit * result_author)
and msg13 = { hAccept : (principals → string → result_author) ;
              hReject : (principals → string → result_author) ;
              hShepherd : (principals → string → msg16) ;
              hRevise : (principals → string → msg12)}
and msg16 = Rebuttal of (string * msg13)
val author : principal → msg9 → result_author
```

The underlying principle for programming session in continuation passing style is that, whenever a message is received by the role, the generated secure implementation calls back the continuation provided by the user and resumes the protocol once user code returns the next message to be sent. Taking advantage of this calling convention, with a separately-typed user-code continuation for each state of each role of the session, we rely on ordinary F# typing to enforce session compliance in user code. The programmer is then free to design the continuations that will be safely executed whenever the chosen role is active. Programming with a session consists then in following the (possibly recursive) generated types by `s2ml`, by filling in the internal choices and payload handling functions (i.e., the continuations).

Programming the author We give sample code (file `author.fs`) for an author that first uploads a paper "First draft", and then, upon receiving a Badformat message, replies with a paper "Submission". It then withdraws the paper with a one-fourth chance, otherwise continues the execution. If the paper enters in shepherding, it answers with a rebuttal message (of payload "Fixed!").

```
let rec handler_response =
{ hAccept = (fun _ comments → FinalVersion("Final", "Accepted!  " ^ comments));
  hReject = (fun _ comments → "Rejected because " ^ comments);
  hShepherd = (fun _ questions →
      Rebuttal("Fixed!", handler_response));
  hRevise = (fun _ reviews → Submit("Paper", handler_response)) }

let rec handler_format =
{ hBadFormat = (fun _ error →
      printf "Formatting error: %s\n" error;
      Upload("Submission", handler_format));
  hOk = (fun _ s →
      if Random.int 4 <> 0
      then Submit("Submission", handler_response)
      else Withdraw((), "Paper withdrawn")) }
```

```
let handler_cfp =
{ hCfp = fun p s → Upload("First draft", handler_format)}

let result = CMS.author "alice" handler_cfp in
printf "Author session complete: %s\n" result
```

In the code, the handler_format record contains two functions: one handles a BadFormat message (which is called back when a BadFormat message is received), prints the error message and sends a Upload message with a different payload and a recursive continuation; the other handles the Ok message and chooses (in an over-simplified way) if the paper has to be withdrawn, i.e. if the next message to be sent is a Submit or a Withdraw. The call to the author role function has thus as arguments the chosen principal, in this case called Alice, and a record handling the first incoming Cfp message.

8.2 Session integrity in the conference management system

In order to illustrate the security properties expected by session users, we discuss some attempts to break integrity on the CMS session. These attacks are instances of those described in Section 2; they are, of course, prevented by the cryptographic protocols generated by `s2ml`. Throughout this discussion, we consider the following assignment of principals to roles: Alice to author, Bob to confman, and Charlie to pc .

Message integrity attacks One could imagine a malicious Alice sending Bob an Upload message even though Charlie never sent a Cfp; if Bob omits to check the required signature from Charlie in Alice's message, session integrity is then be violated. More precisely, consider the first time that the conference manager role gets contacted with an Upload message in Figure 4. At that point, the generated protocol needs to check signatures from the principals playing the roles author and program committee; for our running session with session identifier as above, an incoming message is accepted by Bob as conference manager only if it includes a signature from Charlie (program committee) of a Cfp message, and another signature from Alice (as author) of an Upload message. On the other hand, if Bob as conference manager is at the same node contacted again (e.g. because Bob sent a BadFormat message and entered a loop), in the next incoming message Bob needs to only check a (new) Upload message from Alice, and the Cfp message needs not be forwarded again, as Bob already checked it. Our compiler accounts for both situations, and accordingly outputs specifically tailored functions for message generation and verification.

Session identifier confusions Each session instance needs to have a unique session identifier, as otherwise there could be confusions between different running sessions. For a run of the CMS example, the generated protocol computes as session identifier the hash of the whole session declaration of Figure 5 concatenated with $\widetilde{a} =$ Charlie Alice Bob, and a fresh random nonce N. Since all signatures include the identifier, they cannot be accepted by any other instance of any session.

Session	specification	compiled interface	compiled implementation
S1	4	30	303
S2	6	36	380
S3	19	54	540
CMS (below)	49	83	907

Table 1. Length of sessions specifications and their implementations (lines of code)

	No signatures	Signing, not verifying	Signing & verifying	Standard
first loop	0.231 s	2.79 s	2.95 s	
second loop	0.468 s	5.62 s	6.11 s	
third loop	0.243 s	2.81 s	2.98 s	
total	0.942 s	11.22 s	12.04 s	8.38 s

Table 2. Run-times and cryptographic costs for the CMS session

Intra- and Inter-session replays Message replays against session integrity consist of three categories: (1) a message from one running CMS session can be injected into another running session; (2) an initial message involving a principal can be replayed, trying to re-involve the principal twice; and (3) a message from one running session can be replayed in the same running session (e.g. messages inside any of the three loops of Figure 4, which are particularly vulnerable). These attacks are prevented by using adequately constructed sessions identifiers, anti-replay caches, and virtual clocks.

8.3 Evaluation of the concrete cryptographic protocols

Programs that use generated session interfaces can be linked against networking and cryptographic libraries, to obtain executable code. For portability, our implementation includes two variants of concrete cryptographic libraries: one for F# using the Microsoft .NET cryptographic libraries, and another for Ocaml using the OpenSSL cryptographic libraries. (Unfortunately the two implementations do not fully interoperate, due to incompatibilities between certificate formats.) The data and cryptographic functions we use are as follows. For cryptography, we use SHA1 for hashing, RSASHA1 for signing, and the standard pseudorandom function for nonce generation. Signing uses certificates in '.cer' format for Microsoft .NET and in '.key' format for OpenSSL. For networking, we use Base64 for encoding the messages in a communicable format, and use UDP for communication (although in the future we plan to support TCP-based communication).

The `s2ml` compiler consists of 3352 lines of Ocaml code, plus 289 lines for the OpenSSL bindings and 698 lines for the F# bindings.

Table 1 provides the length of source specifications, generated interfaces, and generated implementations for the sessions presented in this paper. The size of both kinds of generated files is roughly linear in the size of the input, since we generate types and message handlers for all reachable nodes in the graph. On the other hand, our compiler can generate optimized code and message formats for each of these nodes.

Table 2 reports run-times for user code that iterate the loops for the CMS session example. All roles run on a single machine, equipped with a Pentium D at 3.0 GHz running linux-2.6.17-x86_64. We measure the time it takes to run 500 iterations for each of the three loops of the session, thereby processing 4000 message sends and receives. For comparison, we also measure variants of the implementation that omit part of the public-key cryptographic operations. As could be expected, these results show that performance is dominated by the cost of signing messages, whereas message-processing and networking represent less than 10% of run-time. This confirms the benefits of compact, specialized message handlers (and also suggests that an implementation partly based on symmetric session keys would be more efficient.) The last column, labelled 'Standard', compares our implementation to sending messages secured using standard OpenSSL 0.9.8e, reporting the time it takes to send 4000 single-character messages using the command-line tool from the distribution. Our implementation deals with more complex messages but achieves comparable performance.

9 Conclusions and future work

We present a simple language for specifying sessions between roles, and implement it as an extension of ML, with protocol support for running secure distributed sessions. Although session types have been thoroughly studied, and sometimes implemented, we believe this paper is the first to address their secure implementation. Our compiler generates custom cryptographic protocols that guarantee global compliance to the session specification for the principals that use our implementation, with no trust assumptions for the principals that do not. Our theorems relate the runs and labelled traces of a source semantics with primitive sessions to those of an implementation semantics using ordinary communications and cryptographic primitives. Thus, we obtain a full-fledged implementation for distributed sessions with strong security guarantees.

Discussion In terms of protocol verification, our results hold for any number of session declarations and any number of principals, some of them controlled by the adversary, running in parallel any number of instances of these sessions. Even for a single fixed session, we believe such results are beyond automated tools for verifying cryptographic protocols as soon as the session uses loops and branching. Moreover, our result holds for a realistic model—except for the cryptographic primitives, the model is a functional reference implementation.

Cryptographically, our results hold within a symbolic model à la Dolev-Yao. Although a probabilistic polynomial semantics of ML is clearly outside the scope of this paper, we believe our session-authentication mechanisms are also correct under standard, concrete cryptographic hypotheses. Specifically, our usage of signing keys in generated protocols complies with the rules of unforgeability under adaptive chosen-message attacks [21].

We do not consider other session security properties such as confidentiality, left for future work. Moreover, we do not treat important liveness properties, such as progress, global termination, and resistance to denial of service. This is in line with typical security protocol analyses, where the opponent may block all messages anyway.

Prior work consider secure implementation for small process calculi. In comparison, our host language is more expressive and realistic. Hence, we have a running implementation for a language very close to the formal language of the theorems, Also, we rely on this additional expressiveness: we use higher-order functions (and typing, informally) to enforce the session discipline, and use standard functional programming for processing messages. Although we could compile F+S to some process calculus, this would considerably complicate our formalization and proofs.

Overall, we believe that our work illustrates a compelling alternative to protocol handcrafting. For any distributed application that fits our session language, a few lines of high level code can yield a complete distributed implementation with authentication guarantees. In comparison, for session graphs with a dozen of nodes, the design, implementation, and verification of an adequate ad hoc protocol is a challenging task, even for security experts, even if one assumes that all point-to-point communications are already secure.

Future work We are exploring variants of our design to increase the expressiveness of sessions, with extended compiler and proof support. In particular, we are considering session-scoped data bindings, to ensure that the same values are passed in a series of messages, as well as more dynamic principal-joining mechanisms, to enable new principals to enter a role by agreement among the current principals. More generally, we would like to integrate sessions with other language-based security mechanisms, such as secure marshalling for richer types. It would also be interesting (and delicate) to develop secure implementations for existing session-description languages such as BPEL.

Another direction for future work is to extend sessions with more explicit security requirements and relax our message-transparency principle. For instance, one may distinguish "critical messages" with strong authenticity and atomicity, and support them by running a complex subprotocol, such as Byzantine agreement or fair signing. (In principle, our language F+S already enables this approach, as principals may run their own auxiliary protocols on communication channels, but it does not offer linguistic support for them.) However, such extensions would also unavoidably complicate our security model for session programmers.

Acknowledgements This work benefited from discussions with Cosimo Laneve and Jean-Jacques Lévy and from comments of anonymous reviewers.

A Constructing session graphs from role processes, and vice versa

Session graphs and syntactic sessions are interconvertible. Given a session graph and a mapping from labels and roles to their types, we can construct role processes for each role by translating each edge in the graph to dual send and receive operations. Conversely, given the role processes for a session, if the sends and receives are correctly matched, we can construct the corresponding graph, as detailed below.

Given a session $\Sigma = (r_i : \widetilde{\tau_i} = p_i)_{i<n}$, we build the graph $G(\Sigma) = \langle \mathcal{R}, \mathcal{V}, \mathcal{L}, m_0, \mathcal{E}, r \rangle$ as follows.

- $\mathcal{V}$, m_0: we create a node $m_{(f_i)_{i<k}}$ for every sending subprocess $!(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$ within Σ; in particular, we let m_0 be the node for the process p_0 (which must be a send). We similarly create a node for each $\mathbf{0}$ subprocess after a receive.
- $\mathcal{E}$: we create an edge $(m_{\widetilde{f}}, f_i, m)$ for every label f_i, where m depends on the subprocess q of Σ after receiving f_i. (The subprocess q must exist and be unique.) If q is a send, or $q = \mathbf{0}$, we use the corresponding node created in $\mathcal{V}$; if q is $\mu\chi.q'$, we use q' instead of q; if $q = \chi$, we use q' in the binding $\mu\chi.q'$ within Σ. (This binding must exist.)

The definitions for $\mathcal{R}$, $\mathcal{L}$, and r are straightforward. The construction fails if any of the conditions above fail, e.g. if there is a send without a corresponding receive.

From session graphs to local session roles We now build a session Σ from a valid session graph $\mathcal{G}$ plus a type assignment from labels and roles to their payload types $\widetilde{\tau}$ and return types $\widetilde{\tau}$, respectively.

We build the role processes using a graph traversal, starting from m_0. For every new node m with an outgoing edge, we grow a process $\mu\chi_m.!(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$ in $r(m)$, with a branch f_i for each outgoing edge of m, then we traverse the k target nodes. For every node with no outgoing edges, we use the process $\mathbf{0}$. For every already-visited node m, we use the process χ_m.

Interconvertibility In order to reason about the correctness of these transformations, we first define equivalence on graphs: two graphs $\mathcal{G}, \mathcal{G}'$ are equivalent, written $\mathcal{G} \approx \mathcal{G}'$, when

- they have the same set of roles and labels: $\mathcal{R} = \mathcal{R}', \mathcal{L} = \mathcal{L}'$
- they have the same source and target roles for each label: if $(m_1, l, m_2) \in \mathcal{E}$ and $(m_1', l, m_2') \in \mathcal{E}'$, then $r(m_1) = r'(m_1')$ and $r(m_2) = r'(m_2')$
- for every path $(m_0, \widetilde{l})$ in $\mathcal{G}$, there is a path $(m_0', \widetilde{l})$ in $\mathcal{G}'$ and vice versa.

We obtain:

Remark 1. For all valid $\mathcal{G}$, there exists a syntactic session Σ such that $\mathcal{G} \approx G(\Sigma)$.

This is shown by building the session Σ directly from $\mathcal{G}$ using the Global to Local transformation, and then transform it back to $G(\Sigma)$ using the Local to Global transformation. Graphs $\mathcal{G}$ and $G(\Sigma)$ are equivalent since only a renaming on nodes is introduced by the transformations.

We present several examples of these interconversions in the paper. The syntactic sessions S1 and S2 in Section 2 are the result of converting the session graphs in Figure 1(a,b), and vice versa.

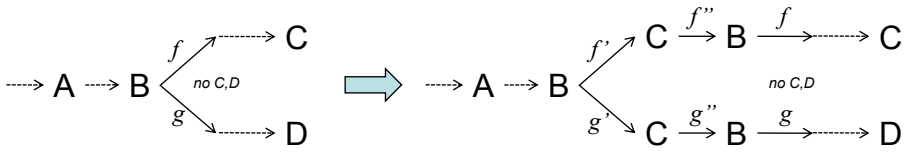

Fig. 6. Eliminating blind forks

B Transforming session graphs to meet Property 3

We say that a sessions graph has a *blind fork* for each two paths that violate Property 3. We show how to eliminate blind forks.

Suppose a graph $\mathcal{G} = \langle \mathcal{R}, \mathcal{V}, \mathcal{L}, m_0, \mathcal{E}, r \rangle$ has a blind fork for the paths $(m, \widetilde{f})$ and $(m, \widetilde{g})$, ending in nodes m_1 and m_2 respectively. Hence, the roles $r(m_1)$ and $r(m_2)$ are distinct, and not active on $\widetilde{f}$ and $\widetilde{g}$. In particular, $\widetilde{f}$ is not a prefix of $\widetilde{g}$, and vice versa. Let m_{fork} be the last common node on two paths; we call it the *forking node*. To eliminate this blind fork, we use the following transform:

- for each edge $(m_{\mathit{fork}}, l, m''') \in \mathcal{E}$, introduce two new nodes $m', m'' \notin \mathcal{V}$ and two new labels $l', l'' \notin \mathcal{L}$; replace $(m_{\mathit{fork}}, l, m''')$ with the three new edges $(m_{\mathit{fork}}, l', m')$, (m', l'', m'') and (m'', l, m'''); and extend r with $r(m') = r(m_1)$ and $r(m'') = r(m_{\mathit{fork}})$.

We check that this transform introduces no new blind fork at m_{fork} and does not affect Property 3 at any other node. Hence, by repeated application of this transform, we can eliminate all blind forks.

Figure 6 illustrates the transform for a sample graph with a blind fork: the graph on the left has two paths ending in roles C and D with a forking node at B; the transformed graph eliminates this fork by inserting C on all paths leading out of the forking node; moreover, by inserting B on each path, the transformed graph maintains the same source and destination roles for all the original labels.

We say that a session graph $\mathcal{G}'$ is a valid transformation of $\mathcal{G}$ when $\mathcal{G}'$ is valid and

- the roles of $\mathcal{G}$ and $\mathcal{G}'$ are the same: $\mathcal{R} = \mathcal{R}'$
- the labels of $\mathcal{G}$ are included in $\mathcal{G}'$ and have the same source and target roles for each label: $\mathcal{L} \subseteq \mathcal{L}'$ and if $(m_1, l, m_2) \in \mathcal{E}$ and $(m_1', l, m_2') \in \mathcal{E}'$, then $r(m_1) = r'(m_1')$ and $r(m_2) = r'(m_2')$
- for every path (m_0, f) in $\mathcal{G}$, there is a path $(m_0', \widetilde{g})$ in $\mathcal{G}'$ and vice versa, where $\widetilde{f}$ is the same as $\widetilde{g}$ after erasing all labels not in $\mathcal{L}$.

Remark 2. For all session graphs $\mathcal{G}$, the graph $\mathcal{G}'$ obtained by repeatedly applying the transform is a valid transformation of $\mathcal{G}$.

The transformed graph $\mathcal{G}'$ is valid because it no longer violates Property 3. Moreover, the transform only add new labels to and from new nodes; hence, the source and destination roles of old labels remain the same, and the new paths are extensions of the old paths (with possibly some of the new labels inserted in them).

C Symbolic code for the libraries

In this appendix we provide the symbolic implementations for the libraries described in Section 4. (These implementations are available as files `crypto.ml` and `prins.ml` in the `lib/symbolic` directory of our prototype implementation.) Our code relies on syntactic sugar: if ... then ... else is a shortcut for standard pattern-matching on the result of the test; the semi-colon, which expresses sequentiality, can be written with our let construct; function application where arguments are not values can be unfolded using let bindings; and anonymous functions introduced with fun can be replaced by a freshly named let binding for the function body.

Symbolic code for the Crypto library We first list the Crypto library, which implements cryptographic types as algebraic datatypes:

```
type keybytes = SKey of name | VKey of keybytes
type bytes = Nonce of name
           | Hash of bytes
           | Concat of bytes * bytes
           | Sign of bytes * keybytes
           | Utf8 of string

let nonce (n: name) : bytes = Nonce n
let genskey (n: name) : keybytes = SKey n
let genvkey (n:keybytes) : keybytes =
  match n with
   | SKey _ → VKey n
let hash (b:bytes) : bytes = Hash b
let concat (m1 : bytes) (m2 : bytes) : bytes = Concat (m1, m2)
let sign (m : bytes) (k : keybytes) : bytes = Sign (m, k)
let verify (m : bytes) (s : bytes) (k : keybytes) : bool =
  match s with
    | Sign (mm, sk) →
        if k = VKey sk && mm = m then true else false
    | _ → 0
let iconcat (m : bytes ) : (bytes * bytes) =
  match m with
    | Concat (m1, m2) → (m1, m2)
let utf8 (s:string) = Utf8 s
let iutf8 (m: bytes) = match m with | Utf8 m1 → m1
```

Symbolic code for the Prins library In our model, the implementation of the Prins library is parameterized by a finite list of principals and a safety predicate on those principals. (In contrast, our concrete prototype implementation retrieves cryptographic materials from a partial database and does not serve the opponent!)

```
let prins = . . . (* a fixed list of all principals *)
let safe (a:principal) = . . . (* a fixed predicate on principals *)
let skeys = List.map (fun a → (a, genskey (new()))) prins
let skey (a : principal) = List.assoc a skeys
```

```
let vkey (a : principal) = genvkey (skey a)
let chans = List.map
    (fun a → let (n:name) = new() in (a, n)) prins

type cache_contents = (bytes * int) list
type cache_result = Stale | Fresh of cache_contents

let asend m a = fork (fun () → send a m)
let caches = List.map
    (fun a → let (n:name) = new() in (a, n)) prins
let _ = map (asend []) caches (* caches init *)

let header s =
  let (msg, sigs) = iconcat (ibase64 s) in
  let (joinflag,header,payload) = iconcat3 (msg) in
  let (host2, dest2, sid) = iconcat3 header in
  let join = if (iS (iutf8 joinflag)) = "J" then true else false in
    ((int_of_string (iS (iutf8 dest2)),sid),join)

let antireplay old a msg =
  let ((sid, r) as k), joining = header message in
  if joining then
    if List.mem k old then Stale
    else Fresh(k::old)
  else Fresh(old)

let psend (a : principal) (m : bytes) =
  let ch = List.assoc a chans in
  let cache = List.assoc a caches in
  let oldcache = recv cache in
  let r = antireplay oldcache a m in
  match r with
  | Fresh(newcache) → asend cache newcache; send ch m
  | Stale → asend cache oldcache

let precv (a : principal) = recv (List.assoc a chans)

(* for modelling the opponent's knowledge only: *)
let psend• = new()
let rec forward () =
  let a,m = recv psend• in
  fork forward; psend a m in
fork forward
let chans• = List.filter (fun (a,n) → not (safe a)) chans
let skeys• = List.filter (fun (a,k) → not (safe a)) skeys
```

The psend• channel implements a small server that receives requests to call psend; this enables the opponent to send messages to safe principals, but not to receive such messages sent by our implementation, by calling psend.

The opponent is given access to prins, safe, vkey, psend•, chans•, and skeys•. Our generated protocol implementations access safe, skey, vkey, psend, and precv. User code is given access only to principal constants.

D Example code

We present in detail the code for the example session from Figure 1(c), as a concrete illustration of the code produced by our session compiler. The example is also included in the prototype distribution, in directory `examples/ex3`.

We first define the session S3 using local roles, then user code that uses this session. Session S3's compilation is sketched next; we first report on the generated interface, and then show excerpts of generated module S3; finally, we include an excerpt of the symbolic run of this example.

Syntactic session Session S3 is defined as follows:

```
session S3 =
  role customer =
   !Request:string; mu start.
   ?( Offer:string;
        !( Change:string; start + Accept; )
    + Reject; !Abort )
  role officer =
    ?Request:string; !Contract:string;
    ?( Confirm + Abort )
  role store:string =
    ?Contract:string; mu start.
    !( Offer:string;
         ?( Change:string; start
         + Accept; !Confirm )
    + Reject )
```

User code The user code defines implementations of each role; an office that forwards messages and print the outcome of the session, a store that offers delivery locations and times (Redmond 8am-9m, by default, then 3pm-4pm, then Orsay at lunchtime, and finally Cambridge at 6pm-7pm). Finally, the code of the customer asks for a meeting on 12 March 2007, and when given an offer of Redmond 8am-9am it changes to Cambridge, otherwise it accepts.

```
open Printf
open S3

// office
let say s = printf "\nOffice: %s.\n\n" s
let log s id () = say s
let request id offer =
  say ("running with "^id.customer^" and "^id.store);
  Contract(offer,
```

```
  { hConfirm = log "run confirmed";
    hAbort = log "run aborted"; })

do Pi.fork (fun () →
  officer "charlie" {hRequest = request})

// store
let offer loc = List.assoc loc
  [ "Default", "Redmond, 8am-9am";
    "Redmond", "Redmond, 3pm-4pm";
    "Orsay", "Orsay, lunchtime";
    "Cambridge", "Cambridge, 6pm-7pm" ]

let server prins req =
  printf "Server: session starting for %s.\n\n" req;
  let rec new_offer prins (loc:string) =
  try
    let o = offer loc in
    Offer(o, {
      hChange = new_offer;
      hAccept = (fun _ () → Confirm((),"in "^o)); })
  with _ → Reject((),"no offer available") in
  new_offer prins "Default"

do Pi.fork(fun () →
  let status = store "bob" { hContract = server; } in
  printf "Store: Done! %s.\n\n" status)

// customer code (non-interactive)
type ChoiceOnOffer = UChange of string | UAccept
let Offer_ui offer =
  if offer = "Redmond, 8am-9am"
  then UChange "Cambridge" else UAccept

do
  let prins = {
    customer = "alice";
    officer = "charlie";
    store = "bob"; } in
  let r = "12 March 2007" in
  let rec msg1 = {
    hReject = (fun _ _ → Abort((),()));
    hOffer = (fun _ offer →
      match Offer_ui offer with
      | UChange location → Change(location,msg1)
      | UAccept → Accept((),()))} in
  let msg0 = Request(r,msg1) in
  let worker () =
    customer prins msg0;
        printf "Customer: session complete.\n\n" in
```

```
worker()
```

Generated interface The generated interface by our compiler for session S3 above is:

```
type principal=string
type principals= {customer:principal; officer:principal; store:principal}
type result_customer = unit

type msg0 =
    Request of (string * msg1)
and msg1 = {
    hOffer : (principals → string → msg2) ;
    hReject : (principals → unit → msg4)}
and msg2 =
    Change of (string * msg1)
  | Accept of (unit * result_customer)
and msg4 =
    Abort of (unit * result_customer)

val customer : principals → msg0 → result_customer

(* Proxy function for officer *)
type result_officer = unit
type msg6 = {
    hRequest : (principals → string → msg7)}
and msg7 =
    Contract of (string * msg8)
and msg8 = {
    hConfirm : (principals → unit → result_officer) ;
    hAbort : (principals → unit → result_officer)}

val officer : principal → msg6 → result_officer

(* Proxy function for store *)
type result_store = string

type msg11 = {
    hContract : (principals → string → msg12)}
and msg12 =
    Offer of (string * msg13)
  | Reject of (unit * result_store)
and msg13 = {
    hChange : (principals → string → msg12) ;
    hAccept : (principals → unit → msg14)}
and msg14 =
    Confirm of (unit * result_store)

val store : principal → msg11 → result_store
```

Generated code excerpts In turn, the compiled module for session S3 above is as follows (we only show one example of a sending and receiving function here, for label Request and Accept):

```
(* gen_[Request ]_Request = sendWiredRequest ... *)
(* gensig_customer_Request = sendlabel ... *)
let sendWiredRequest (host:int) (dest:int) (prin_list: principals_list)
        : (wired0 → bytes list) = function (x:wired0) →
    match x with
    | WiredRequest (host, dest, localtime, sid, last_sig, payl) →
      let _ = Printf.printf "Preparing message: Request(%s)\n" (payl) in
      let header = concat3 (utf8 (S (string_of_int host)))
                                        (utf8 (S (string_of_int dest))) sid in
      let tag = utf8 (S "Request") in
      let payload = concat tag (utf8 (S payl)) in
      let vision = [(0,"Request")] in
      let (newLast_sig, sigs) = send_label prin_list sid host dest
                                        "Request" last_sig localtime vision in
      sendMsg host dest prin_list sid header payload
                                        newLast_sig sigs (joining "Request")
    | _ → assert false
...
(* recv_Accept *)
let receiveWired12 (host:int) (prin_list: principals_list) running_sid last_sig
                                    : (unit → wired12) = function ((():unit) →
  let (tag, host2, dest2, sid, raw_payload, sigs) =
                receiveMsg host prin_list running_sid in
  match tag with
  | "Reject" → let payload = () in
    let _ = Printf.printf "Accepting message: Reject(%s)\n" ("") in
    let vision = [(2,"Reject")] in
    let newLast_sig = receive_label prin_list (sid) host "Reject"
                                    sigs last_sig localtime vision in
    WiredReject(host2, dest2, localtime, sid, newLast_sig, payload)
  | "Offer" → let payload = iS (iutf8 raw_payload) in
    let _ = Printf.printf "Accepting message: Offer(%s)\n" (payload) in
    let vision = [(2,"Offer")] in
    let newLast_sig = receive_label prin_list (sid) host "Offer"
                        sigs last_sig localtime vision in
    WiredOffer(host2, dest2, localtime, sid, newLast_sig, payload)
...
let customer (prin_list: principals) (user_input : msg0) =
...
(* sending Request *)
and customer_msg0 sid (last_sig:bytes list) : msg0 → result_customer = function
  | Request(x,next) →
     let dest = 1 in
    let newLast_sig = sendWiredRequest host dest (genprin prin_list)
                        (WiredRequest (host,dest,localtime,sid,last_sig,x)) in
    customer_msg1 sid newLast_sig next
```

```
...
(* recv Accept or Change *)
and customer_msg1 sid (last_sig:bytes list)
                        : msg1 → result_customer = function handlers →
  let r = receiveWired12 host (genprin prin_list) sid last_sig () in
    match r with
    | WiredOffer (host,dest,localtime,sid,newLast_sig,x) →
    let next = handlers.hOffer prin_list x in
    customer_msg2 sid newLast_sig next
    | WiredReject (host,dest,localtime,sid,newLast_sig,x) →
    let next = handlers.hReject prin_list x in
    customer_msg4 sid newLast_sig next
in
 let nonce = mkNonce () in
    let f s a = concat s (utf8 (S a)) in
    let sid = concat3 session (make_principals (genprin prin_list)
                                                nb_parties) nonce in
 let emptyLast_sig = populate_emptysigs nb_parties in
 Printf.printf "Executing role customer with
                      principal %s...\n" (List.nth (genprin prin_list) host) ;
  customer_msg0 sid emptyLast_sig user_input
```

Symbolic run Finally, we report on the execution of the above code for session S3:

```
Executing role store with principal bob
Executing role officer with principal charlie...
Executing role customer with principal alice...
...
Preparing message: Request(12 March 2007)
alice sent NJ | 0 | 1 | SHA1(S3,(14,(Confirm,unit),9)(2,(Accept,unit),14)
(2,(Change,string),12)(4,(Abort,unit),10)(12,(Reject,unit),4)(12,(Offer,string),2)
(7,(Contract,string),12)(0,(Request,string),7)) | alice | charlie | bob |
nonce6 | Request | 12 March 2007 | 0 | RSA−SHA1{rsa_secret3}
[SHA1(S3,(14,(Confirm,unit),9)(2,(Accept,unit),14)(2,(Change,string),12)
(4,(Abort,unit),10)(12,(Reject,unit),4)(12,(Offer,string),2)
(7,(Contract,string),12)(0,(Request,string),7)) | alice | charlie |
bob | nonce6 | Request | 0] | 0 | |

Receiving message: Request(12 March 2007)

Office: running with alice and bob.

Preparing message: Contract(12 March 2007)
charlie sent NJ | 1 | 2 | SHA1(S3,(14,(Confirm,unit),9)(2,(Accept,unit),14)
(2,(Change,string),12)(4,(Abort,unit),10)(12,(Reject,unit),4)
(12,(Offer,string),2)(7,(Contract,string),12)(0,(Request,string),7)) |
alice | charlie | bob | nonce6 | Contract | 12 March 2007 | 0 |
RSA−SHA1{rsa_secret5}[SHA1(S3,(14,(Confirm,unit),9)(2,(Accept,unit),14)
(2,(Change,string),12)(4,(Abort,unit),10)(12,(Reject,unit),4)
(12,(Offer,string),2)(7,(Contract,string),12)(0,(Request,string),7)) |
```

alice | charlie | bob | nonce6 | Contract | 0] | 0 |
RSA−SHA1{rsa_secret3}[SHA1(S3,(14,(Confirm,unit),9)(2,(Accept,unit),14)
(2,(Change,string),12)(4,(Abort,unit),10)(12,(Reject,unit),4)
(12,(Offer,string),2)(7,(Contract,string),12)(0,(Request,string),7)) |
alice | charlie | bob | nonce6 | Request | 0] |
...
Store: Done! in Cambridge, 6pm−7pm.
Accepting message: Confirm()
Office: run confirmed.

E Proofs for Section 7

We first describe series of low-level transitions that implement each high-level transition, which essentially provides the proof of Theorem 3. We then use this description as the basis of the case analysis in the proof of Theorem 2. We finally prove Lemma 1 and obtain Theorem 1 as a corollary of Theorem 2.

E.1 Proof of Theorem 3

Lemma 2. *Let W be a valid implementation of H. For every transition $H \xrightarrow{\alpha}_{\mathsf{K}} H'$ in* F+S*, where α does not contain any signing key of a safe principal nor any element of $\mathcal{N}$, there exists a valid implementation W' of H such that $W \overset{\varepsilon}{\Rightarrow}_{\mathsf{K}} W'$ in* F.

Proof. Except for the transition steps that involve sessions, every high-level step carries over to a low-level step with the same label, as their redexes are preserved by the translation. We distinguish seven kinds of high-level session transitions, depending on the base rules they use: rules KSENDS (using INIT or STEP), KRECVS (using JOIN or STEP), and KSTEP (using COMML, COMMR and ENDS).

Unfolding the function definitions given in Section 6 and Appendix C, we exhibit a series of low-level transitions that implement these high-level session transitions, leading to a valid implementation of H' for some possibly updated state T' .

We first set up auxiliary notations. We write the elements of high-level configurations without subscripts: $H = K, \rho, P$ or $H' = K', \rho', P'$; we use subscripts for the elements of low-level configurations: $W_0 = K_0, \rho_0, P_0$ or $W_1 = K_1, \rho_1, P_1$. We also use italics for metavariables.

Cache transitions We begin with auxiliary low-level transitions for filtering messages through the anti-replay cache. We write $W_1 = K_1, \rho_1, P_1 \mid E_1$ for the low-level adversary knowledge, environment, processes and expression context. For a given safe principal a, we let $P_a =$ send cache_a $\mathit{content}_a$ abbreviate the message that holds the state of the cache for principal a on channel cache_a: hence, $\mathit{content}_a$ is the list of session identifiers and roles already seen by a, recorded in $T.\mathit{cache}(a)$.

Starting from a message m sent to a in expression context E_1, under the premise that 'header m' evaluates either to '(sid,r),true' with sid,$r \notin \mathit{content}_a$ or to '_,false', by definition of psend we have transitions:

$$K_1, \rho_1, P_1 \mid P_a \mid E_1[\text{psend } a\ m]$$

$\xrightarrow{\text{APPLY}(psend)}\xrightarrow{\text{APPLY,LETVAL}}{}^{*}$

$K_1, \rho_1, P_1 \mid P_a \mid E_1[\text{let oldcache = recv } cache_a \text{ in let isreplay = [...] in [...]}]$

$\xrightarrow{\text{COMML}}$

$K_1, \rho_1, P_1 \mid () \mid E_1[\text{let oldcache = } content_a \text{ in let isreplay = [...] in [...] }]$

$\xrightarrow{\text{LETVAL}}$

$K_1, \rho_1, P_1 \mid () \mid E_1[\text{let isreplay = antireplay } content_a\ a\ m \text{ in match [...]}]$

$\xrightarrow{\text{APPLY,LETVAL}}{}^{*}\xrightarrow{\text{MATCH,MISMATCH}}{}^{*}$

$K_1, \rho_1, P_1 \mid () \mid E_1[\text{match } isreplay \text{ with [...] }]$ (1)

$\xrightarrow{\text{MATCH}}$

$K_1, \rho_1, P_1 \mid () \mid E_1[\text{asend } cache_a\ newcache \text{ ; send } ch_a\ m\ ]$

$\xrightarrow{\text{APPLY}}\xrightarrow{\text{FORK}}\xrightarrow{\text{LETVAL}}$

$K_1, \rho_1, P_1 \mid () \mid \text{send } cache_a\ newcache \mid E_1[\text{send } ch_a\ m\ ]$

where $newcache = (sid,r)::content_a$ if 'header m' evaluates to '(sid,r),true', and $newcache = content_a$ otherwise. We let $\xrightarrow{\text{Cache}}$ abbreviate this sequence of transitions, which represents successful anti-replay filtering.

Conversely, if 'header m' evaluates to '(sid,r),true' with $sid,r \in content_a$, then after (1) we have the transitions:

$$(1) \xrightarrow{\text{MATCH,MISMATCH}}{}^{*} K_1, \rho_1, P_1 \mid () \mid E_1[\text{asend } cache_a\ content_a] \xrightarrow{\text{APPLY,LETVAL}}{}^{*} K_1, \rho_1, P_1 \mid () \mid P_a \mid E_1[()]$$

(This case plays a role in the proof of Theorem 2, but not in the proof of Theorem 3.) In the rest of the proof, we may omit inert threads consisting of just the () expression, such as the thread left after P_a in the transitions above.

Session transitions We now translate the high-level session transitions.

ENDS: The transition is trivially simulated: by definition, we have $[\![s.0(e)]\!]_T = [\![e]\!]_T$, so no low-level transition step is needed to simulate this step.

INIT: The high-level transition is of the form

$$K, \rho, P \mid E[S.r_0^b\ (a_i)_{i<n}(g(\widetilde{v}), w)] \xrightarrow{s\overline{g}\ \widetilde{v}}_{\mathsf{K}} K \cup \{\widetilde{v}\}, \rho', P \mid E[s.p'\ (w)]$$

Let $P_0 \mid P_a \mid E_0[S.r\ prins\ (g(\widetilde{v}), w)]$ match $[\![P \mid E[S.r_0^b\ (a_i)_{i<n}(g(\widetilde{v}), w)]]\!]_T$. Let also $K_0 = [\![K]\!]_T$ and $\rho_0 = [\![\rho]\!]_T$. We have the following sequence of transitions:

$K_0, \rho_0, P_0 \mid P_a \mid E_0[S.r\ prins\ (g(\widetilde{v}), w)]$

$\xrightarrow{\text{APPLY}}$

$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let st = init } prins \text{ in send\_}\emptyset \text{ st } (g(\widetilde{v}), w)]$ (2)

$\xrightarrow{\text{APPLY,LETVAL}}{}^{*}$

$$K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{send\_}\emptyset\ st\ (g(\widetilde{v}), w)]$$
$$\xrightarrow{\text{APPLY}} \xrightarrow{\text{MISMATCH}}{}^{*} \xrightarrow{\text{MATCH}} \xrightarrow{\text{LETVAL}}$$
$$K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{let m = gen\_}\emptyset\text{\_}g\ st\ \widetilde{v}\ \text{in psend}\ a\ \text{m ; recv\_}g\ st\ w]$$
$$\xrightarrow{\text{APPLY,LETVAL}}{}^{*}$$
$$K_0, \rho_1, P_0 \mid P_a \mid E_0[\text{psend}\ a\ m\ ;\ \text{recv\_}f\ st\ w|]$$
$$\xrightarrow{\text{Cache}} \xrightarrow{\text{LETVAL}}$$
$$K_0, \rho_1, P_0 \mid \text{send}\ cache_a\ ((sid,r)::content_a) \mid E_0[\text{send}\ ch\ m\ ;\ \text{recv\_}g\ st\ w]$$
$$\xrightarrow{\overline{ch}\ m(\text{KSEND})}{}_{\mathsf{K}} \xrightarrow{\text{LETVAL}}$$
$$K_0 \cup \{m\}, \rho_1, P_0 \mid \text{send}\ cache_a\ ((sid,r)::content_a) \mid E_0[\text{recv\_}g\ st\ w]$$

where (sid,r) comes from the evaluation of 'header m'. In (2), the init function has some side-effects, transforming ρ_0 into $\rho_1 = [\![\rho']\!]_T$.
Also, the message m added to the low-level K is not the exact translation of the updated high-level K, which contains the message payload v, new signatures, and the nonce. To obtain matching knowledges, we apply KAPPLY steps at the low level to retrieve these values from m.

KSENDS: The high-level transition is of the form

$$K, \rho, P \mid E[s.p\ (g(\widetilde{v}), w)] \xrightarrow{s\overline{g}\ \widetilde{v}}_{\mathsf{K}} K \cup \{\widetilde{v}\}, \rho, P \mid E[s.p'\ (w)]$$

We write $K_0 = [\![K]\!]_T$, $\rho_0 = [\![\rho]\!]$ and $P_0 \mid P_a \mid E_0[\text{let x}_s\text{ = }(g(\widetilde{v}), w)\text{ in send\_}\widetilde{g}\ st\ \text{x}_s] = [\![P \mid E[s.p\ (g(\widetilde{v}), w)]]\!]_T$. We have the following sequence of transitions:

$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let x}_s\text{ = }(g(\widetilde{v}), w)\text{ in send\_}\widetilde{g}\ st\ \text{x}_s]$$
$$\xrightarrow{\text{LETVAL}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{send\_}\widetilde{g}\ st\ (g(\widetilde{v}), w)]$$
$$\xrightarrow{\text{APPLY}} \xrightarrow{\text{MISMATCH}}{}^{*} \xrightarrow{\text{MATCH}} \xrightarrow{\text{LETVAL}}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{let m = gen\_}\widetilde{g}\text{\_}g\ \text{st}\ \widetilde{v}\ \text{in psend}\ a\ \text{m ; recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\text{APPLY,LETVAL}}{}^{*}$$
$$K_0, \rho_0, P_0 \mid P_a \mid E_0[\text{psend}\ a\ m\ ;\ \text{recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\text{Cache}} \xrightarrow{\text{LETVAL}}$$
$$K_0, \rho_0, P_0 \mid \text{send}\ cache_a\ newcache \mid E_0[\text{send}\ ch\ m\ ;\ \text{recv\_}\widetilde{g}g\ st\ w]$$
$$\xrightarrow{\overline{ch}\ m(\text{KSEND})}{}_{\mathsf{K}} \xrightarrow{\text{LETVAL}}$$
$$K_0 \cup \{m\}, \rho_0, P_0 \mid \text{send}\ cache_a\ newcache \mid E_0[\text{recv\_}\widetilde{g}g\ st\ w]$$

We apply KAPPLY steps as above, to obtain matching knowledges.

JOIN: The high-level transition is of the form

$$K, \rho, P \mid E[S.r_0^b\ a_j(w)] \xrightarrow{sg\ \widetilde{v}}_{\mathsf{K}} K', \rho', P \mid E[s.p'\ (w.g\ \widetilde{a}\ \widetilde{v})]$$

We write $K_0 = [\![K]\!]_T, \rho_0 = [\![\rho]\!]$ and $P_0 \mid P_a \mid \mathrm{F} \mid E_0[S.r\ \mathit{self}\ w] = [\![P \mid E[S.r_0^b\ a_j(w)]]\!]_T$.
We also note F = forward () the forward process and $P_1 = P_0 \mid$ send $\mathit{cache}_a\ \mathit{content}_a$.
The translation of this transition is the following sequence of transitions:

$$
\begin{array}{l}
K_0, \rho_0, P_0 \mid P_a \mid \mathrm{F} \mid E_0[S.r\ \mathit{st}\ w] \\
\xrightarrow{\textsc{Apply}} \\
K_0, \rho_0, P_0 \mid P_a \mid \text{let a,m = recv psend}^\bullet \text{ in [...]} \mid E_0[S.r\ \mathit{self}\ w] \\
\xrightarrow{\text{psend}^\bullet\ (a,m0)(\textsc{KRecv})}_{\mathsf{K}} \xrightarrow{\textsc{LetVal}} \xrightarrow{\textsc{Fork}} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_0 \mid P_a \mid \mathrm{F} \mid \text{psend } a\ m \mid E_0[S.r\ \mathit{self}\ w] \\
\xrightarrow{\textsc{Cache}} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send } \mathit{ch}\ m0 \mid E_0[S.r\ \mathit{self}\ w] \\
\xrightarrow{\textsc{Apply}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send } \mathit{ch}\ m0 \mid E_0[\text{let m0 = precv } \mathit{self} \text{ in let st,m = join m0 in [...]} \\
\xrightarrow{\textsc{Apply}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid \text{send } \mathit{ch}\ m0 \mid E_0[\text{let m0 = recv } \mathit{ch} \text{ in let st,m = join m0 in [...]} \\
\xrightarrow{\textsc{CommL}} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{let st,m = join } m0 \text{ in [...]} \\
\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{if st.prins.}r = \mathit{self} \text{ then verify\_}\emptyset\ \mathit{st}\ m\ w] \\
\xrightarrow{\textsc{Match}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{verify\_}\emptyset\ \mathit{st}\ m\ w] \\
\xrightarrow{\textsc{Apply}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{let path = visible\_}\emptyset\ m \text{ in match path with [...]} \\
\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{match } \mathit{path} \text{ with [...]} \\
\xrightarrow{\textsc{Mismatch}}{}^{*} \xrightarrow{\textsc{Match}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{let st,payload = chk\_}\emptyset\_\widetilde{g'}\ \mathit{st}\ m \text{ in let next = [...]} \\
\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*} \xrightarrow{\textsc{LetVal}} \\
K_0, \rho_0, P_1 \mid \mathrm{F} \mid E_0[\text{let next =} w.last(\widetilde{g'})\ \mathit{st}.\text{prins } \mathit{payload} \text{ in [...]}
\end{array}
\tag{3}
$$

The cache transition (3) always succeeds, since there is no bad input.

KRecvS: The high-level transition is of the form

$$K, \rho, P \mid E[s.p\ (w)] \xrightarrow{sg\ \widetilde{v}}_{\mathsf{K}} K', \rho, P \mid E[s.p'\ (w.g\ \widetilde{a}\ \widetilde{v})]$$

We write $K_0 = [\![K]\!]_T, \rho_0 = [\![\rho]\!]$ and $P_0 \mid \mathrm{F} \mid E_0[\text{recv\_}\widetilde{g}f\ \text{st}\ w] = [\![P \mid E[s.p\ (w)]]\!]_T$.
We also note F = forward () the forward process.

The translation of this transition is the following sequence of transitions:

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{recv\_}\widetilde{g}f\ st\ w]$
$\xrightarrow{\textsc{Apply}}\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}$

$K_0, \rho_0, P_0 \mid \text{let a,m = recv psend}^{\bullet}\text{ in [...]} \mid E_0[\text{let m = recv } ch \text{ in verify\_}\widetilde{g}f\ st\ \text{m}\ w]$
$\xrightarrow{\text{psend}^{\bullet}\ (a,m)(\textsc{KRecv})}_{\mathsf{K}}\xrightarrow{\textsc{LetVal}}\xrightarrow{\textsc{Fork}}\xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid \text{psend } a\ m \mid E_0[\text{let m = recv } ch \text{ in verify\_}\widetilde{g}f\ st\ \text{m}\ w]$
$\xrightarrow{\text{Cache}}\xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid \text{send } ch\ m \mid E_0[\text{let m = recv } ch \text{ in verify\_}\widetilde{g}f\ st\ \text{m}\ w]$
$\xrightarrow{\textsc{CommL}}\xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{verify\_}\widetilde{g}f\ st\ m\ w]$
$\xrightarrow{\textsc{Apply}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{let path = visible\_}\widetilde{g}f\ m \text{ in match path with [...]}]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{match } path \text{ with [...]}]$
$\xrightarrow{\textsc{Mismatch}}{}^{*}\xrightarrow{\textsc{Match}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{let st,payload = chk\_}\widetilde{g}f\_\widetilde{g'}\ st\ m \text{ in let next = [...]}]$
$\xrightarrow{\textsc{Apply},\textsc{LetVal}}{}^{*}\xrightarrow{\textsc{LetVal}}$

$K_0, \rho_0, P_0 \mid \mathrm{F} \mid E_0[\text{let next = w.}last(\widetilde{g'})\ st\text{.prins } payload \text{ in [...]}]$

KSTEP: High-level session communication steps are translated as a high-level send (without extending K) followed by a high-level receive. □

We are now ready to prove completeness for our implementation:

Restatement of Theorem 3. *Let W be a valid implementation of H with no bad inputs. For all transitions $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H'$ in* F+S *such that ψ contains neither any signing key of a safe principal nor any elements of $\mathcal{N}$, there exist a valid implementation W' of H' with no bad inputs and a direct translation φ of ψ such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in* F.

Proof. By induction on the number of high-level transitions, applying Lemma 2 for each transition. □

E.2 Notations for Theorem 2

Administrative transitions Among the transitions that are part of the implementation of a high-level session transition (made explicit in the proof of lemma 2), we designate as *administrative* the ones that do not concern communication (i.e. different from SEND or RECV or COMML). We use $\rightarrow_{\mathsf{KA}}$ as an abbreviation for those transitions.

Transition names In order to easily reason about the reordering of the low-level transitions, we give names to some selected sequences of low-level transitions that are part of the session implementation detailed in the proof of Lemma 2.

- The low-level implementation of INIT and KSENDS have only one non-administrative transition, labelled with $\overline{ch}\ m$, corresponding to the application of the (KSEND) rule. We name this transition ϵ_0 and write the sequence of transitions as follows:

$$\xrightarrow{\text{pre}_{\epsilon_0}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_0}_{\mathsf{K}} \xrightarrow{\text{post}_{\epsilon_0}}{}^{*}_{\mathsf{KA}}$$

- The low-level implementation of KRECVS and JOIN have three non-administrative transitions, which we name as follows:
  - $\epsilon_1 = \text{psend}^{\bullet}(a, m)(\text{KRECV})$
  - ϵ_2 is the silent transition corresponding to a (COMML) on a $cache_a$ channel
  - ϵ_3 is the silent transition corresponding to a (COMML) on a ch_a channel

  The sequence of transitions is then written

$$\xrightarrow{\text{pre}_{\epsilon_1}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_1}_{\mathsf{K}} \xrightarrow{\text{pre}_{\epsilon_2}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_2}_{\mathsf{K}} \xrightarrow{\text{pre}_{\epsilon_3}}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon_3}_{\mathsf{K}} \xrightarrow{\text{post}_{\epsilon_3}}{}^{*}_{\mathsf{KA}}$$

  In the proof of theorem 2, we refer more generically to the pre and (possibly empty) post transitions of a given non-administrative transition ϵ_n.

Thread and commutativity We designate as a *thread* an expression placed in evaluation context. Each session role implementation uses only one thread, so administrative transitions (which are purely local computations) can be reordered with respect to other threads' transitions.

Visibility We write V for the visibility function: if $\widetilde{g}$ is the sequence of labels on a given path from the initial node m_0 to a node m with role r, $V(\widetilde{g})$ is the corresponding visible sequence, that is, the result of erasing from $\widetilde{g}$ every label g (1) whose sending role is r; or (2) that is followed by a label whose sending role is either r or g's sending role.

E.3 An extended translation for Theorem 2

The proof of theorem 2 relies on the translation relation and other invariants that track the link between low and high-level configurations. However, there exist low-level intermediate states of our session implementation that do not have a direct high-level reflection. We thus extend the translation function so that it coincides with the translation given in Section 7 on their joint domain, and so that it also keeps track of the session implementation intermediate steps.

We first define an extended version of the state T. The only added elements are the inner and outer functions ($T.\mathit{inner}$ and $T.\mathit{outer}$). For every session record $s\ (a_i)_{i<n}\ \{\widetilde{r}\}$: S in ρ,

- $T.\mathit{nonce}(s)$ is a term N_s in K.
- $T.\mathit{path}(s)$ is an initial path $\widetilde{f}$ of S, decorated with strictly-increasing integers $\widetilde{j}$, ending by a label sent or received by a safe principal;

– $T.\mathit{stuck}(s)$ is the set of roles that have received a bad input so far—in that case, the roles have silently terminated.

For every principal a,

– $T.\mathit{cache}(a)$ is the content of the cache of principal a: a set of pairs of session identifiers and roles $(\widetilde{sid}, r)$.
– $T.\mathit{outer}(a)$ is the multiset of messages $\widetilde{m}$ that have been received on ch_abut have not been checked against the cache yet.
– $T.\mathit{inner}(a)$ is the multiset of messages $\widetilde{n}$ that have passed the cache test but have not received further treatment, with the following properties: (1) the multiset contains at most one joining message for each role of each running session, and (2) all joining message are also registered in the cache.

Finally, T provides an infinite (and co-infinite) set of fresh supplementary names, $\mathcal{N}$, which is formally used to supply a fresh nonce to the high-level environment whenever the low-level environment receives a fresh session nonce.

We define the translation $[\![K, \rho, P]\!]_T$ from F+S configurations to F configurations, as follows:
To translate K:

– we replace session records $s.p$ with the session nonces $T.\mathit{nonce}(s)$;
– we add the signatures exported by T;
– we remove the signing keys of all safe principals and the supplementary nonces $\mathcal{N}$.

To translate the store ρ:

– we replace session type definitions $\widetilde{S}$ with the types and function definitions of $M_{\widetilde{S}}$;
– we remove the session entries and the nonces of $\mathcal{N}$ not in the image of $T.\mathit{nonce}$.

To processes, we add the following ones:

– the forwarding process F = forward ();
– for each principal a, the process $P_a = \text{send } \mathit{cache}_a\ T.\mathit{cache}(a)$, where cache_a is the channel of $\rho_{L\widetilde{S}}$ that holds the state of the antireplay cache for a. (The forwarding code and cache management are defined in Appendix C.)
– for each principal a and for each message n of $T.\mathit{inner}(a)$, a sending process send $ch_a\ n$ (ch_a denotes the principal's communication channel).
– for each principal a and for each message m of $T.\mathit{outer}(a)$, a sending process psend $a\ m$.

To translate the process P:

– we translate all running session roles within P as follows:

$[\![s.p(w)]\!]_T = \mathbf{0}$	if p's role is in $T.\mathit{stuck}(s)$
$[\![s.p(w)]\!]_T = \text{S.recv\_}\widetilde{g}\ st\ [\![w]\!]_T$	else if p is an input or is an output of a message in $T.\mathit{inner}$ or $T.\mathit{outer}$
$[\![s.p(e)]\!]_T = \text{let } x_s = [\![e]\!]_T \text{ in S.send\_}\widetilde{g}\ st\ (x_s)$	else if p is an output
$[\![s.p(e)]\!]_T = [\![e]\!]_T$	else if p is $\mathbf{0}$

where $\widetilde{g}$ and st are computed from $T.\mathit{nonce}(s)$ and $T.\mathit{path}(s)$. Note that the translation knows for each process $s.p$ which session S it implements, the reason being that the translation acts on configurations which include the necessary information in environments ρ.
- Expressions of the form $S.r\ldots$ are unchanged but now interpreted as function calls, rather than primitive session entries.

The refined translation coincides with the original translation when both the inner and outer functions ($T.\mathit{inner}$ and $T.\mathit{outer}$) are empty (as in the proof of theorem 3).

E.4 Auxiliary path properties

Lemma 3. *Consider a session Σ and a node n with role r. Let $\widetilde{f}$ be a visible sequence ending in n. Let $\widetilde{g}$ be the longest suffix such that r is not active, of a pre-image of $\widetilde{f}$ from the visibility function V. Then the set of active roles in $\widetilde{f}$ is the set of active roles in $\widetilde{g}$.*

Proof. By definition of visibility. □

Lemma 4. *Consider a session Σ and an initial path $\widetilde{g}$ leading to a node n_2 with role r. Let $\widetilde{f}$ be the longest suffix of $\widetilde{g}$ for which r is not active. Let n_1 be the first node of $\widetilde{f}$.*

Then the set of active roles of all paths starting at n_1 where r is not active is included in the set of active roles of $\widetilde{f}$.

Proof. The proof relies on the absence of "blind forks" in session graphs, excluded by property 3 of section 2. □

E.5 Proof of Theorem 2

Restatement of Theorem 2. *Let W be a valid implementation of H. For all transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, there exist a valid implementation W° of H° and a translation φ of ψ such that*

$$H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ \qquad W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{K}} W'' \qquad W' \rightarrow^*_{\mathsf{K}} W''$$

Proof. By induction on the number of non-administrative transitions in $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$. We start from a low-level configuration W that is the translation in a state T of a high-level configuration $H = K, \rho, U$, thus $W = (K_0, \rho_0, U_0) = (\llbracket K \rrbracket_T, \llbracket \rho \rrbracket_T, \llbracket U \rrbracket_T)$. We first explain our induction principle and the step reordering it uses, then conduct the main case analysis.

Outline In this proof, our initial hypothesis is that there are transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ that lead a low-level configuration W to a configuration W' and that have an observable trace φ. The invariant of our proof relates a high-level configuration H, a state T and a low-level configuration W by the translation function described above: $W = \llbracket H \rrbracket_T$. Our goal is then to propagate this invariant over the transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ by finding intermediary configurations W_i where we have high-level configuration H_i and states

T_i such that $W_i = [\![H_i]\!]_{T_i}$ and that the H_i are related by high-level transitions. However the intermediary low-level configurations W_i cannot always be found on the initial path from W to W' because of the interleaving of steps allowed by the session implementation. Thus, we first need to reorder (and complete) the original series of transitions before matching high-level and low-level transitions.

We present this reordering in an inductive manner. We start by examining the low-level transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ that we can decompose and complete in the following way:

$$W \xrightarrow{\text{extra} + \text{pre}_\epsilon}{}^{*}_{\mathsf{KA}} W_0 \xrightarrow{\epsilon}_{\mathsf{K}} W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$$

The sequence $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ starts with some administrative steps $W \xrightarrow{\text{extra+pre}_\epsilon}{}^{*}_{\mathsf{KA}} W_0$ followed by a session-related communication or a user-code transition $W_0 \xrightarrow{\epsilon}_{\mathsf{K}} W_1$. This transition may be silent or not: we use the metavariable ϵ to denote either the presence and content of a label or its absence. Among the administrative transitions that precede the ϵ transition, we distinguish the ones that causally precede ϵ: pre_ϵ; and the others: 'extra'. We write φ' for the sequence of labels that are on the remaining transitions $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$. We have thus $\varphi = \epsilon\ \varphi'$.

We then commute the 'extra' steps after the ϵ transition. We possibly need to add administrative post_ϵ steps to fully match in the low-level transitions the implementation of a high-level session transition. The result is the following series of transitions:

$$W \xrightarrow{\text{pre}_\epsilon}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^{*}_{\mathsf{KA}} W^\circ \xrightarrow{\text{extra}}{}^{*}_{\mathsf{KA}} W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$$

where the extra transitions are moved after the complete session step consisting in pre_ϵ, ϵ, post_ϵ. The transitions $W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$ consist of those in $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$ after having removed any transitions that is also in post_ϵ. If all of the post_ϵ transitions are included in $W_1 \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W'$, then we have $W' = W''$. Otherwise the missing ones can be applied to W' to reach W''.

We iterate the reordering from configuration W°. The remaining transitions

$$W^\circ \xrightarrow{\text{extra}}{}^{*}_{\mathsf{KA}} W_1' \overset{\varphi'}{\Rightarrow}_{\mathsf{K}} W''$$

contain fewer non-administrative steps than the original series of transitions.

The next part of the proof consists in a case analysis on the transitions $W \xrightarrow{\text{pre}_\epsilon}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^{*}_{\mathsf{KA}} W^\circ$ that correspond to specific high-level transitions .

Case analysis We now proceed with the inductive step of the proof: for each transitions $W \xrightarrow{\text{pre}_\epsilon}{}^{*}_{\mathsf{KA}} \xrightarrow{\epsilon}_{\mathsf{K}} \xrightarrow{\text{post}_\epsilon}{}^{*}_{\mathsf{KA}} W^\circ$, we exhibit a corresponding high-level transition that preserves the valid implementation relation.

APPLY, MATCH, MISMATCH, LETVAL, LETFUN, TYPE, FRESH (KSTEP): The different KSTEP transitions are reflexions of P-transitions and e-transitions. By hypothesis those silent transitions are not administrative steps, so the redex they reduce is also present in the original high-level U and is invariant under translation. The low level transitions, which are available at high level, may therefore be mirrored identically.

KAPPLY: Since we have excluded administrative steps, the applied function appears in ρ before and after the translation. The arguments applied to the function used in the low-level KAPPLY transition may consist of values built from cryptographic material (signatures, nonces) received by the low-level attacker as a biproduct of session communication. The matching between high and low-level adversarial knowledge provided by our invariant ensures that a high-level application of KAPPLY directly simulate the low-level one.

KSEND on a principal channel in chans$^\bullet$: In this case, the channel belongs to an unsafe principal b and the message is thus sent by the implementation of a running session role, for some principal a, at node n. As a consequence, the message is among the pending messages in $T.\mathit{inner}(b)$. The low-level process that initiates this message send is of the form send $ch_b\ m$; [...], itself derived from let x_s = $[\![e]\!]_T$ in S.send_$\widetilde{g}$ st (x_s).

We know that $T.\mathit{path}(s)$ is an initial path. If the message m is the first of the session, then $T.\mathit{path}(s)$ is empty and we set $T.\mathit{path}(s)$ to the label and time-stamp, thus maintaining the relation between $T.\mathit{path}(s)$ and the role processes. If s is already started, the implementation for the principal a imposes that it previously received a signed label that, by definition of a valid implementation, is in $T.\mathit{path}(s)$, and that there is only one safe participant a that is in a sending state. Thus $T.\mathit{path}(s)$ ends in the current node n. Appending the last label and time-stamp of m to $T.\mathit{path}(s)$ therefore preserves the invariant.

As part of m, the attacker receives a sequence of signatures that are added to K. These signatures correspond to the updated translation under the new $T.\mathit{path}(s)$. However, to have matching high and low-level adversarial knowledge as required by the valid implementation invariant, the high-level configuration need to do some KAPPLY transitions to build the corresponding signatures from the available keys.

When the attacker joins the session for the first time (i.e. has not already joined as a different role), the low-level message contains a nonce that we assume, without loss of generality, is already included in the supplementary infinite set of nonces $\mathcal{N}$ in the high-level K set. We therefore add N_s to $T.\mathit{nonce}(b)$ to maintain our invariant.

In all cases, the high-level transition is a KSENDS.

KRECV on the psend$^\bullet$ channel: This a communication from the adversary to a safe principal a using the psend$^\bullet$ channel and therefore handled by the forward process, defined as let a,m = recv psend$^\bullet$ in [...]. The reception of the message yields the process

$$\text{let a,m} = a, m \text{ in fork forward ; psend a m}$$

which performs administrative transitions then yields process psend $a\ m$. To conclude, we update T by appending the message m to the multiset of pending messages in $T.\mathit{outer}(a)$, and leave the high-level configuration unchanged.

COMML or COMMR on a safe principal channel from chans: By definition of the translation, this transition is enabled only for a receiver in the implementation of a running role r instantiated by a safe principal a. (Reception for unsafe principals is handled in case KRECV). Then we know that the received message m has passed the cache test and is among the pending messages in $T.\mathit{inner}(a)$. The sending pro-

cess is then of the form: send $ch\ m$ while the receiving one is:

$$\text{let m = recv } ch \text{ in verify\_}\widetilde{g}g \text{ st m } [\![w]\!]_T$$

Recall that $\widetilde{g}$ ends at the node from which r previously sent g ; let n_0 be the node receiving g. Let f be the label corresponding to the m message and n_2 be the node in which r receives f.
The message m contains a session id, and a list of signed labels and time-stamps. Checking that the session id corresponds to a running session (or a new one) gives us the correct high-level s. The verify_$\widetilde{g}g$ function also checks that the signatures are correct, the time-stamps are consecutive and the list of labels corresponds to a visible sequence $\widetilde{f}f$.
By our invariant, the labels in $\widetilde{f}$ that are signed by safe principals are in $T.path(s)$. Since we know that $T.path(s)$ is an initial path, we can deduce that $T.path(s)$ contains in particular the last label that is signed by a safe principal in $\widetilde{f}$. We call n_1 the receiving node of this label.
We next prove that $T.path(s)$ ends in n_1. Suppose for contradiction that $T.path(s)$ goes further. Then there is a path starting in n_1 which does not meet r and which leads a node n_3 where role r', different from r, is active and is instantiated by a safe principal. By Lemma 4, this role r' is also active in all paths from n_1 to n_2. By Lemma 3, this role has signed a label among the visible sequence $\widetilde{f}$ received by r in n_2 and thus $T.path(s)$ contains this label between n_1 and n_2, a contradiction. Therefore $T.path(s)$ ends in n_1 and there is no label signed by a safe principal between n_1 and n_2 in $\widetilde{f}$.
If a compliant principal is the sender of the message m, then we can simulate the f message exchange at high-level by a CommL or CommR. If the message m is sent by an unsafe principal, then there is (possibly) a multi-step gap between n_1 and n_2. The signed visible labels $\widetilde{f}$ received by r in n_2 give then a skeleton of internal adversary communication. By definition of visibility, we know that we can complete it to a contiguous path ending by f, going from n_1 to n_2. We therefore append this path to $T.path(s)$ to maintain the invariant. These internal communication steps are simulated at high-level through the use of (multiple) OComm steps inside the global KRecvS transition.
Finally, the message m is removed from $T.inner(a)$.

CommL or CommR on a cache channel: In this case, the receiving process is produced by a call psend $a\ m$ that either is itself called by the implementation code

$$\text{let } x_s = [\![e]\!]_T \text{ in S.send\_}\widetilde{g}\ st\ (x_s)$$

or that comes from the presence of m in $T.outer(a)$. The object of this transition is the communications of the current version of the cache of a and then administrative transitions decide if m can be forwarded to a. If m is a joining message and the session id s and the role r played by a in s are already recorded in the cache, then a new cache process is forked and r is added to $T.stuck(s)$ because m is a replay attack. Otherwise if m passes the cache test, we add it to $T.inner(a)$ and remove it from $T.outer(a)$. If m is a joining message we add the corresponding role and session id to $T.cache(a)$. We make no transition at high-level. The new low-level

configuration is the image of the high-level configuration under the updated state T, modulo the identification of stuck processes with $\mathbf{0}$.

KSEND, KRECV, COMML or COMMR on any other channel: Since in this case the channels are not part of the session, these transitions correspond to redexes that are identically present in the original high-level U. We therefore reflect these transitions identically at high-level. Note that, in the KRECV case, a value v from the low-level $\mathsf{Val}(K, \rho)$ is sent from the attacker: our invariant that matches high and low-level adversarial knowledge ensures that such a value can be built and sent at high-level. □

E.6 Proof of Lemma 1

Restatement of Lemma 1. *We have transitions $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ for some fresh names $\widetilde{n}$ where $\omega \notin fn(\psi)$ if and only if $\rho, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ for some process O that does not contain ω, does not match on constructors in ρ, that calls only pure functions of ρ, and whose values defined in ρ are all included in $\mathsf{Val}(K, \rho)$.*

Proof. ($\Rightarrow$) First, for every K, ρ, P, and ψ where $\overline{\omega}() \notin fn(\psi)$ we exhibit a process $O = e_O \mid \prod_{\sigma \in K} e_\sigma$, such that
(1) each e_σ represents an active session role controlled by the attacker and is of the form $(\text{let x}_i = e_i \text{ in})_{i \in [1..n]}\ \sigma\ v$, where $n \geq 0$;
(2) the processes in O do not contain ω;
(3) the processes in O do not match on constructors and call only pure functions of ρ;
(4) the values in O are all contained in $\mathsf{Val}(K \uplus \{c\}, \rho \uplus \{c\})[\omega := \omega']$;
(5) if $K, \rho, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ then $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$.
Here, $c \notin \rho$ is a distinguished channel used for communicating values between the different subprocesses of O; $\omega' \notin \rho$ is also a distinguished name used whererver K uses ω instead. We proceed by induction on the length of $\overset{\psi}{\Rightarrow}_{\mathsf{K}}$.

Base Case: $\overset{\psi}{\Rightarrow}_{\mathsf{K}}$ is empty.
Hence, P sends on ω (KSEND). Let O be an arbitrary process satisfying the five conditions above; say $O = \mathbf{0} \mid \prod_{\sigma \in K} \sigma()$. Then $\rho \uplus \{c, \omega'\}, P \mid O \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$ (SEND,PARL).
Inductive Case: $K, \rho, P \xrightarrow{\alpha}_{\mathsf{K}} K', \rho', P' \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$, where $\alpha \neq \overline{\omega}()$. By the inductive hypothesis, there exists a process $O' = e_{O'} \mid \prod_{\sigma \in K'} e'_\sigma$ such that the five conditions above hold. By case analysis on the first transition, we construct a process $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ that also satisfies the conditions:

KSTEP The transition is wholly in P. Let $O = O'$; P performs the same transition in $\rho \uplus \{c, \omega'\}, P \mid O$.

KAPPLY $K, \rho, P \rightarrow_{\mathsf{K}} K \cup \{w\}, \rho, P$, where $\rho, l_i\ v_0 \ldots v_k \longrightarrow_{\mathsf{e}}{}^{*} \rho, w$ and $l_i, v_0, \ldots, v_k \in \mathsf{Val}(K, \rho)$ We spell out this case in detail, the construction of O in the other cases is similar. Here, $K' = K \cup \{w\}$, $\rho' = \rho$, and $P' = P$. First, we rename any occurrences of ω in $v_0, \ldots, v_k$ to ω', obtaining $v'_0, \ldots, v'_k$ By the inductive hypothesis, O' may contain $w[\omega = \omega']$ and so must be of the form:
$e_{O'}[x := w][\omega = \omega'] \mid \prod_{\sigma \in K} e'_\sigma[x := w][\omega = \omega']$, where x is some fresh variable.
Let e_O be $\text{let x} = l_i\ v'_0 \ldots v'_k \text{ in } (\text{let } \_ = \text{send } c \text{ x in})_{\sigma \in K}\ e_{O'}$ and for each $\sigma \in K$, let $e_\sigma = \text{let x} = \text{recv } c \text{ in } e'_\sigma$. Here, e_O computes the function result, broadcasts it to all

the active session processes in O, and continues with $e_{O'}$, while each e_σ receives this result and continues with e'_σ. (This is a common pattern that we use to distribute any new values that are generated in K among the subprocesses of O.) Then $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ satisfies the induction hypothesis; it extends O' with the application of a pure function l_i, with some new values $v'_0, \ldots, v'_k \in \mathsf{Val}(K, \rho)[\omega := \omega']$; it introduces some message sends and receives on the channel c but none on the channel ω; moreover, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \rho \uplus \{c, \omega'\}, P \mid O'$ (APPLY,COMMR); hence $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KSEND $K, \rho, P \xrightarrow{\overline{ch}\ w}_{\mathsf{K}} K \cup \{w\}, \rho, P'$, where P sends a message w on a channel ch in K. Here, $\omega \notin fn(ch, w)$; hence, $ch, w \in \mathsf{Val}(K \uplus \{w\}, \rho)[\omega := \omega']$, and $O' = e_{O'}[x := w] \mid \prod_{\sigma \in K} e'_\sigma[x := w]$.
Let e_O = let x = recv ch in (let _ = send c x in)$_{\sigma \in K}$ $e_{O'}$, and for each $\sigma \in K$, let e_σ = let x = recv c in e'_σ. Then $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ satisfies the induction hypothesis; in particular, using COMMR, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \rho \uplus \{c, \omega'\}, P' \mid O' \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KRECV $K, \rho, P \xrightarrow{ch\ w}_{\mathsf{K}} K, \rho, P'$, where $ch, w \in \mathsf{Val}(K, \rho)$ and $\omega \notin fn(ch, w)$; hence $ch, w \in \mathsf{Val}(K, \rho)[\omega = \omega']$, and $O' = e_{O'} \mid \prod_{\sigma \in K} e'_\sigma$.
Let e_O be let _ = send ch w in $e_{O'}$. Then $O = e_O \mid \prod_{\sigma \in K} e'_\sigma$ satisfies the induction hypothesis; it extends O' with a message w sent on ch, where $w \in \mathsf{Val}(K, \rho)$; it does not introduce a message send on ω, since by the inductive hypothesis $ch \neq \omega$; using COMML, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KSENDS $K[\sigma_r], \rho, P \xrightarrow{s\overline{g}\ w}_{\mathsf{K}} K[s.p] \cup \{w\}, \rho'', P'$, where $\omega \notin fn(w)$, and $\rho, P \xrightarrow{s\overline{g}\ w}_{\mathsf{P}} \rho', P'$, and $\rho', \sigma_r \xrightarrow{sg}_{\mathsf{s}} \rho'', s.p$. Hence $O' = e_{O'}[x := w] \mid \prod_{\sigma \in K[s.p]} e'_\sigma[x := w]$; in particular $e_{s.p}$ corresponds to the session process $s.p$.
Let $e_{\sigma_r} = \sigma_r(v)$ where $v.g\ \widetilde{a}\ x$ = (let _ = send c x in)$_{\sigma \in K[s.p]} e_{s.p}$; and for each $\sigma \in K[\sigma_r]$, if $\sigma \neq \sigma_r$ then let e_σ = let x = recv c in e'_σ. Let e_O be let x = recv c in $e_{O'}$. Then $O = e_O \mid e_\sigma \prod_{\sigma \in K[\sigma]} e'_\sigma$ satisfies the induction hypothesis; using RECVS, $\rho \uplus \{c, \omega'\}, P \mid O \rightarrow_{\mathsf{P}}{}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

KRECVS $K[\widetilde{\sigma}], \rho, P \xrightarrow{sg\ w}_{\mathsf{K}} K[\widetilde{s.p}, s'.p'], \rho''', P'$, where $\omega \notin fn(w)$, $K[\widetilde{\sigma}], \rho \xrightarrow{s\overline{g_1}}_{\mathsf{o}} \xrightarrow{sg_1}_{\mathsf{o}} \cdots \xrightarrow{s\overline{g_m}}_{\mathsf{o}} \xrightarrow{sg_m}_{\mathsf{o}} K[\widetilde{s.p}, \sigma_s], \rho'$, and $\rho', \sigma_s \xrightarrow{s\overline{g}}_{\mathsf{s}} s'.p', \rho''$, and $\rho'', P \xrightarrow{sg\ w}_{\mathsf{P}} \rho''', P'$. That is, the session role processes $\widetilde{\sigma}$ in K perform session communications with each other, resulting in the processes $\widetilde{s.p}$ and a process σ_s that then performs a session send and interacts with a session receive in P. We construct a process O that simulates this sequence of steps.
First we construct the process O_m corresponding to $K[\widetilde{s.p}, \sigma_s]$,$\rho'$, just before the last session send. By the inductive hypothesis, $O' = e_{O'} \mid \prod_{\sigma \in K[\widetilde{s.p}, s'.p']} e'_\sigma$, where $e_{s'.p'}$ must be of the form (let $\mathrm{x}_i = e_i$ in)$_{i \in [1..n]}$ $s'.p'(v)$, where v is a record of handlers $h_1, \ldots, h_k$. We let $e_{\sigma_s} = \sigma_s(g(w), v')$, where v' has the same message handlers $h_1, \ldots, h_k$ as v, and for each h_i, we let $v'.h_i\ \widetilde{a}\ x$ =let (x_1, ...,x_n) = recv c in $e_{s.p}$; in all other cases $e_\sigma = e'_\sigma$. We let $e_{O_m} = e_{O'}$ |(let $\mathrm{x}_i = e_i$ in)$_{i \in [1..n]}$ send c (x_1, ...,x_n).
Hence, we split the process for σ_s into two threads, passing control through a standard process calculus maneuver: the first part e_{σ_s} performs the session send and waits with a message handler for the next session receive; in the meanwhile, the second subprocess of e_{O_m} performs some intermediate computations and sends the re-

sults to the message handler in e_{σ_s} over the channel c. Then $O_m = e_{O_m} \mid \prod_{\sigma \in K[\widetilde{s.p}, \sigma_s]} e_\sigma$ satisfies the induction hypothesis; using RECVS and SENDS, $\rho \uplus \{c, \omega'\}, P \mid O_m \rightarrow_{\mathsf{P}}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$. By using similar techniques, we extend O_m to the process O corresponding to $K[\widetilde{\sigma}]$, ρ by adding the corresponding internal session communications.

Using the process $O = e_O \mid \prod_{\sigma \in K} e_\sigma$ as constructed above, we can establish the ($\Rightarrow$) direction of the lemma. For every $K, \widetilde{n}, \rho, P, \psi$, if $K \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$ and $\omega \notin fn(\psi)$, then the process $O_{K,c,\widetilde{n},\rho}$ defined as follows satisfies the lemma:
let c = new() in let ω' = new() in (let n_i = new () in)$_{n_i \in \widetilde{n}}$ (let _ = fork e_σ in)$_{\sigma \in K}$ e_O.
It uses only pure functions in ρ, does not match on constructors, and does not contain ω; all its values are in $\mathsf{Val}(K_{\rho,O})[\omega = \omega']$ and hence the only values that are not in $\mathsf{Val}(K_{\rho,O})$ are undefined in ρ; finally, using FRESH and FORK, we have:
$\rho, P \mid O_{K,c,\widetilde{n},\rho} \rightarrow_{\mathsf{P}}^* \rho \uplus \{c, \omega', \widetilde{n}\}, P \mid O \rightarrow_{\mathsf{P}}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$

($\Leftarrow$) For every ρ, O, such that O only uses pure functions in ρ, does not match on constructors, does not contain ω, and whose values defined in ρ are contained in $\mathsf{Val}(K, \rho)$ we exhibit a $K, \widetilde{n}, \psi$ such that the lemma holds. We define reduction contexts: $R[\cdot] ::= E[\cdot] \mid\mid R[\cdot] \mid P \mid\mid P \mid R[\cdot]$.

For a given ρ, O we define $K_{\rho,O}$ as the smallest set that satisfies the following:

- $\omega \in K_{\rho,O}$,
- for every pure expression e such that $O = R[e]$ and $\rho, e \longrightarrow_{\mathsf{e}}^* \rho, v$, $v \in K_{\rho,O}$, and
- for every session state σ in O: for all ρ_1, σ_1 such that $\rho_1, \sigma_1 \xrightarrow{\widetilde{\eta}}_{\mathsf{s}}^* \rho, \sigma, \sigma_1 \in K_{\rho,O}$; and for all ρ_2, σ_2 such that $\rho, \sigma \xrightarrow{\widetilde{\eta}}_{\mathsf{s}}^* \rho_2, \sigma_2, \sigma_2 \in K_{\rho,O}$.

We show that if $\rho \uplus \rho_l, P \mid O \rightarrow_{\mathsf{P}}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$, where ρ_l consists of local type, session, and function definitions used only in O (and not in P), then there exists $\widetilde{n}, \psi$, such that $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$. We proceed by induction on the number of reductions in $\rightarrow_{\mathsf{P}}^*$.
Base Case: $\rightarrow_{\mathsf{P}}^*$ is empty.
Hence, P sends on ω; and $K_{\rho,O}, \rho, P \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$, using KRECV.
Inductive Case: $\rho \uplus \rho_l, P \mid O \rightarrow_{\mathsf{P}} \rho' \uplus \rho_l', P' \mid O' \rightarrow_{\mathsf{P}}^* \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$. By the induction hypothesis, there exists $\widetilde{n}', \psi'$, such that $K_{\rho',O'} \uplus \{\widetilde{n}'\}, \rho' \uplus \{\widetilde{n}'\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$. We exhibit $\widetilde{n}, \psi$ by case analysis on the first reduction $\rho, P \mid O \rightarrow_{\mathsf{P}} \rho', P' \mid O'$:

PARL The reduction step occurs within P: $\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', P'$ and $O = O'$. Then $K_{\rho,O} = K_{\rho',O'}$, $\widetilde{n} = \widetilde{n'}$, $\psi = \psi'$, and, using KSTEP, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \Rightarrow_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.

PARR The reduction step occurs within O. By further case analysis:

MATCH,MISMATCH,LETVAL,FORK, LETFUN, TYPE, SESSION In these cases, $\rho = \rho'$, $O = O'$, $K_{\rho,O} = K_{\rho',O'}$, and no labeled transitions are needed.

APPLY $O = R[l v_o \ldots v_k]$, $O' = R[e\{x_0 = v_0; \ldots; x_k = v_k\}]$, and $\rho = \rho'$, where l must be a pure function, and hence, e is a pure expression.

APPLY $O = R[l v_0 \ldots v_k]$, $O' = R[e\{x_0 = v_0; \ldots; x_k = v_k\}]$, and $\rho = \rho'$, where l must be a pure function, and hence, e is a pure expression. $K_{\rho',O'}$ already includes the value v computed from the function application ($\rho, e\{x_0 = v_0; \ldots; x_k = v_k\} \longrightarrow_{\mathsf{e}}^* \rho, v$). $K_{\rho,O}$ contains $l, v_0, \ldots, v_k$, hence by KAPPLY, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \rightarrow_{\mathsf{K}} K_{\rho',O'} \uplus \{\widetilde{n}\}, \rho' \uplus \{\widetilde{n}\}, P$.

FRESH $\rho' = \rho \uplus \{n\}$, $O = R[new()]$, and $O' = R[n]$. Let $\widetilde{n} = \widetilde{n}' \uplus \{n\}$, then $K_{\rho,O} \uplus \{\widetilde{n}', n\} = K_{\rho,O'} \uplus \{\widetilde{n}'\}, \rho \uplus \{\widetilde{n}', n\}, P \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$

COMMR,COMML In the non-session communication case, $\rho = \rho'$ and $K_{\rho,O} = K_{\rho,O'}$; no labeled transitions are needed. If it is a session communication on a session defined in ρ_l, again $K_{\rho,O} = K_{\rho',O'}$ and no transitions are needed. The remaining cases are when $O = R[\sigma_1 e_1] \mid R'[\sigma_2 e_2]$ and $O' = R[\sigma_1' e_1'] \mid R'[\sigma_2' e_2']$ Then σ_1, σ_2, σ_1', and σ_2' are all included in both $K_{\rho,O}$ and $K_{\rho',O'}$ and again no transitions are needed.

COMML O sends a message to P. If it is a channel communication $\overline{c}\ v$, then both c and v must be in $K_{\rho,O}$; and using KRECV, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{c\ v}{\Rightarrow}_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.
If it is a session communication $s\overline{g}\ v$, then $O = R[\sigma(g(v), w)]$ and $\rho, \sigma \xrightarrow{\widetilde{s\overline{g}\ v}}_{\mathsf{s}} \rho', \sigma'$; using KRECVS, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \xrightarrow{sg\ v}_{\mathsf{K}} K', \rho'', P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.

COMMR P sends a message to O. If it is a channel communication cv, then c must be in $K_{\rho,O}$; and using KSEND, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \overset{\overline{c}\ v}{\Rightarrow}_{\mathsf{K}} K_{\rho,O} \uplus \{\widetilde{n}, v\}, \rho \uplus \{\widetilde{n}\}, P' \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$.
If it is a session communication $s\overline{g}\ v$, then $O = R[\sigma(w)]$ and $\rho, \sigma \xrightarrow{\widetilde{s\overline{g}\ v}}_{\mathsf{s}} \rho', \sigma'$; using KSENDS, $K_{\rho,O} \uplus \{\widetilde{n}\}, \rho \uplus \{\widetilde{n}\}, P \xrightarrow{s\overline{g}\ v}_{\mathsf{K}} K', \rho'', P' \overset{\psi'}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$. □

E.7 Proof of Theorem 1

Auxiliary Lemma In the configuration $L\ \widetilde{S}\ UO$ we have that the libraries L and the session declarations $\widetilde{S}$ reduce deterministically. Hence, for any U and O, there exists $\rho_{L\widetilde{S}}$ (defining functions, sessions and values), substitutions σ and $\sigma^\bullet$ for values, the forwarder process F and the cache processes C, s.t. $\emptyset, L\ \widetilde{S}\ UO \rightarrow_{\mathsf{P}}^* \rho_{L\widetilde{S}}, U\sigma \mid \mathrm{F} \mid \mathrm{C} \mid O\sigma^\bullet$. Here $\rho_{L\widetilde{S}}$ records the declared sessions, plus the functions and types declared in the Crypto and Prins libraries as given in Appendix C, while σ records principals constants prins and $\sigma^\bullet$ records opponent accessible information: prins, safe, vkey, psend$^\bullet$, chans$^\bullet$, and skeys$^\bullet$, plus any information bound by U. Also, F = forward () is the forwarder process that receives messages from opponent code to be sent to compliant principals and C represents the cache processes send $cache_a$ $content_a$ (one per principal).

Similarly, for the implemented sessions $M_{\widetilde{S}}$, we have the deterministic reductions $\emptyset, L\ M_{\widetilde{S}}\ UO' \rightarrow_{\mathsf{P}}^*\ \rho_{LM_{\widetilde{S}}}, U\sigma \mid \mathrm{F} \mid \mathrm{C} \mid O'\sigma^\bullet$. Here, in contrast to the session declarations in $\rho_{L\widetilde{S}}$, the environment $\rho_{LM_{\widetilde{S}}}$ contains role functions and types defined by the compiler, satisfying the relation: $[\![\rho_{L\widetilde{S}}]\!]_\emptyset = \rho_{LM_{\widetilde{S}}}$ which holds at the beginning, when no session has yet started, for the state empty $T = \emptyset$, that is, with empty *cache, nonce, path*, and *stuck* (see Section 7).

We also know that initial user code is independent of the compilation: $[\![U\sigma]\!]_\emptyset = U\sigma$

In addition, we let $K_0 \uplus K_h$ denote the initial values K_0 exported by L to opponent code, i.e. prins, safe, vkey, psend$^\bullet$, chans$^\bullet$, and skeys$^\bullet$, plus K_h that contains additional nonces $\mathcal{N}$ and signing keys from safe principals.

We then obtain that for all fresh $\widetilde{n}$: $K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}}, U$ is a valid configuration. Since the values $K_0 \uplus K_h$ exported by the libraries are the same in both high and low levels, modulo K_h, we get $[\![K_0 \uplus K_h \uplus \{\widetilde{n}\}]\!]_\emptyset = [\![K_0 \uplus K_h]\!]_\emptyset \uplus \{\widetilde{n}\} = K_0 \uplus \{\widetilde{n}\}$.

Gathering all of the above auxiliary results, we obtain the following lemma:

Lemma 5. *For the initial configuration $L\ \widetilde{S}\ UO$ it holds:*

$$\emptyset, L\ \widetilde{S}\ UO \rightarrow_{\mathsf{P}}{}^{*}\ \rho_{L\widetilde{S}}, U\sigma \,|\, F \,|\, C \,|\, O\sigma^{\bullet} \tag{4}$$

$$\emptyset, L\ M_{\widetilde{S}}\ UO' \rightarrow_{\mathsf{P}}{}^{*}\ \rho_{LM_{\widetilde{S}}}, U\sigma \,|\, F \,|\, C \,|\, O'\sigma^{\bullet} \tag{5}$$

$$[\![\rho_{L\widetilde{S}}]\!]_{\emptyset} = \rho_{LM_{\widetilde{S}}} \tag{6}$$

$$[\![U\sigma]\!]_{\emptyset} = U\sigma \tag{7}$$

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}}, U \quad \textit{is a valid configuration} \tag{8}$$

$$[\![K_0 \uplus K_h \uplus \{\widetilde{n}\}]\!]_{\emptyset} = [\![K_0 \uplus K_h]\!]_{\emptyset} \uplus \{\widetilde{n}\} = K_0 \uplus \{\widetilde{n}\} \tag{9}$$

Restatement of Theorem 1. *If $L\ M_{\widetilde{S}}\ U\ O'$ may fail in* F *for some O' where ω does not occur, then $L\ \widetilde{S}\ U\ O$ may fail in* F+S *for some O where ω does not occur.*

Proof. We prove the theorem using the above fact. Since $L\ M_{\widetilde{S}}\ U\ O'$ fails, using Lemma 5(5) we have that:

$$\emptyset, L\ M_{\widetilde{S}}\ U\ O' \rightarrow_{\mathsf{P}}{}^{*}\ \rho_{LM_{\widetilde{S}}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O'\sigma^{\bullet} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$$

Let W be

$$K_0 \uplus \{\widetilde{n}\}, \rho_{LM_{\widetilde{S}}} \uplus \{\widetilde{n}\}, (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$$

Since $O'\sigma^{\bullet}$ does not contain ω, by Lemma 1($\Leftarrow$) applied on $\rho := \rho_{LM_{\widetilde{S}}}$, $P := (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$, $O := O'\sigma^{\bullet}$, there are fresh names $\widetilde{n}$ and φ s.t. $\omega \notin fn(\varphi)$ and $W \overset{\varphi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}}$.

Let H be

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}} \uplus \{\widetilde{n}\}, (U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C})$$

We have that W is a valid implementation of H, that is, $[\![H]\!]_{\emptyset} = W$, by Lemma 5(7), Lemma 5(8), and Lemma 5(9). By Theorem 2, there is $W^{\circ}, H^{\circ}, W''$ s.t. $W \overset{\varphi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}}$ $W^{\circ} \rightarrow_{\mathsf{K}}^{*} W''$, $W' \rightarrow_{\mathsf{K}}^{*} W''$ and $H \overset{\psi\cdot\overline{\omega}()}{\Rightarrow}_{\mathsf{K}} H^{\circ}$, with φ a translation of ψ. Hence, $\omega \notin fn(\psi)$ neither. Expanding H and ignoring H°, we have:

$$K_0 \uplus K_h \uplus \{\widetilde{n}\}, \rho_{L\widetilde{S}} \uplus \{\widetilde{n}\}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \overset{\psi}{\Rightarrow}_{\mathsf{K}} \xrightarrow{\overline{\omega}()}_{\mathsf{K}}$$

Now, by Lemma 1($\Rightarrow$), since $\omega \notin fn(\psi)$, there is O s.t. $\rho_{L\widetilde{S}}, U\sigma \,|\, \mathrm{F} \,|\, \mathrm{C} \,|\, O\sigma^{\bullet} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$; by Lemma 5(4), then:

$$\emptyset, L\ \widetilde{S}\ U\ O \rightarrow_{\mathsf{P}}{}^{*} \rightarrow_{\mathsf{P}}{}^{*} \xrightarrow{\overline{\omega}()}_{\mathsf{P}}$$

i.e., $L\ \widetilde{S}\ U\ O$ fails, establishing the theorem. □

References

1. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, Apr. 2002.
2. P. Adão and C. Fournet. Cryptographically sound implementations for communicating processes (extended abstract). In M. B. et al., editor, *33rd International Colloquium on Automata, Languages and Programming (ICALP), Part II*, volume 4052 of *LNCS*, pages 83–94. Springer, July 2006.

3. K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10(2):8, 2007.
4. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop, (CSFW)*, pages 139–152, July 2006.
5. J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-Safe Distributed Programming for OCaml. In *ACM SIGPLAN Workshop on ML*, pages 20–31, New York, NY, USA, Sept. 2006. ACM.
6. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. In A. Brogi, editor, *1st International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA)*, volume 97 of *ENTCS*, pages 175–195. Elsevier Science, 2004.
7. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In C. Hankin, editor, *Programming Languages and Systems, 16th European Symposium on Programming (ESOP)*, LNCS, pages 2–17. Springer, 2007.
8. S. Carpineti and C. Laneve. A basic contract language for web services. In P. Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 197–213. Springer, 2006.
9. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 45–57, Jan. 2002.
10. R. Corin and P. Denielou. A protocol compiler for secure sessions in ml. In *3rd Symposium on Trustworthy Global Computing (TGC'07)*, Sophia, France, Nov. 2007. LNCS. To appear.
11. R. Corin, P. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 170–186, Venice, Italy, July 2007. IEEE.
12. V. Cortier, B. Warinschi, and E. Zalinescu. How to protect security protocols against active attackers. Unpublished draft.
13. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
14. P.-M. Deniélou and J. J. Leifer. Abstraction preservation and subtyping in distributed languages. In *11th International Conference on Functional Programming (ICFP)*, pages 286–297, New York, NY, USA, 2006. ACM.
15. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *20th European Conference for Object-Oriented Languages*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, July 2006.
16. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
17. Dot. At `http://www.graphviz.org/`.
18. D. Duggan. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop (CSFW)*, page 238, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
19. M. Fahndrich, M. Aiken, C. Hawblitzel, G. H. Orion Hodson, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EUROSYS*, 2006.
20. S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, pages 74–90, 1999.
21. S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17(2):281–308, 1988.
22. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

23. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Programming Languages and Systems, 7th European Symposium on Programming (ESOP)*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
24. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, 2008. To appear.
25. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 128–141, 2001.
26. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.
27. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
28. J. H. Morris, Jr. *Lambda-Calculus Models of Programming Languages*. Ph. D. dissertation, MIT, Dec. 1968. Report No. MAC–TR–57.
29. Objective Caml. At `http://caml.inria.fr`.
30. A. Perrig and D. Song. Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 64–76, 2000.
31. P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *10th International Conference on Functional Programming (ICFP)*, pages 15–26, New York, NY, USA, Sept. 2005. ACM.
32. D. X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.
33. D. Syme. *F#*, 2005. At `http://research.microsoft.com/fsharp/`.
34. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA)*, volume 68 of *ENTCS*. Elsevier Science, 2003.
35. V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.
36. T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy (S&P)*, page 178, 1993.
37. N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *1st International Workshop on Security and Rewriting Techniques*, ENTCS, 2006.
38. L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 236–250, 2003.