Secure Implementations for Typed Session Abstractions

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

1 Session types for 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, implementations of these abstractions can sometimes entirely shield programmers from low-level attacks (such as message interception and rewriting) [1, 2]. However, this is seldom the case in practice, as security concerns force the programmer to understand low-level protocol 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 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 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 [7, 8, 9, 13, 16, 30, 32]. 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 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 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# [28] (a dialect of ML [23, 24]) with distributed communication and sessions, so that type safety yields functional guarantees: 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 our programs 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 theorem states that the safety guarantees implied by session types do not depend on the implementations of any remote peers: from the viewpoint of our distributed programs, any action that may occur may also occur in an abstract setting, using a centralized implementation that enforces all session types.

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.

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. On the other hand, it leaves the code that uses 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.)

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.

The security of automatically-generated cryptographic protocol implementations crucially relies 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 uses standard cryptographic algorithms and networking primitives; the produced code supports distributed execution. A second, symbolic implementation defines cryptography using algebraic datatypes, in Dolev-Yao style; the produced code supports concurrent execution, and is also our formal model. Thus, our security theorems apply directly to arbitrary user code calling our session-generated code calling our symbolic library code, within a formal model of a subset of F# (rather than an ad-hoc abstract model of the protocol loosely related to actual executable code).

Related work Session types have been explored first for process calculi [17, 20, 32], to describe interaction on single channels. Behavioral types [9, 21] support more expressive sessions, typed as CCS processes possibly involving multiple channels. Another type system [6] also combines session types and correspondence assertions [19]. Recent works consider applications of session types to concrete settings such as CORBA [29], a multi-threaded functional language [30], and a distributed object-oriented language [13]. In particular, the Singularity OS [16] explores the usage of typed contracts in operating system design and implementation. In all these works, type systems are used to ensure session compliance within fully trusted systems, excluding the presence of an (active, untyped) attacker.

Sessions for Web Services are 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. In recent, independent work, Carbone *et al.* [7] also present a language for describing Web interactions from a global viewpoint and describe their endpoint 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. More generally, distributed languages such as Acute and HashCaml [26, 12, 5] also rely on types to provide general functional guarantees for networked programs, in particular type-safe marshalling and dynamic rebinding to local resources.

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 [33], while preserving high-level, typed-based integrity and secrecy guarantees. In a similar vein, the Fairplay [22] 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 [15].

Protocol synthesis and transformation have been explored in other settings: for instance, the Automatic Protocol Generation (APG) tool [25] generates authentication protocols then verified using Athena [27] and, more recently, Cortier *et al.* [11] 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 gives the (fairly standard) 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 concludes.

The appendix provides additional details on our implementation, including listings for selected libraries. A companion paper also includes a detailed programming example and all proofs [10].

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 the 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 role implementation.

As a running example, we consider 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.

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.

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 cannot have the same label: 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

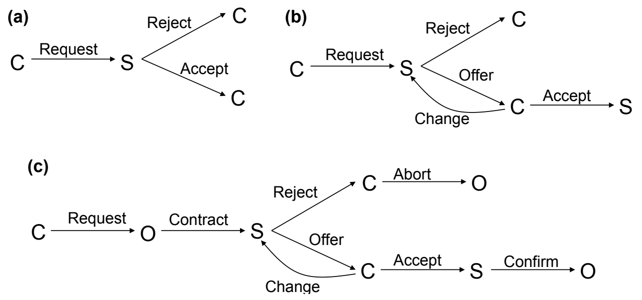

Figure 1. Graphs for (a) a basic session, (b) a session with a cycle, and (c) a three-party session.

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.

Figure 1 displays three increasingly complex session graphs for our running example:

(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.

(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.

Local session roles We now 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:

τ ::=	Payload types
int | string	base types
p ::=	Role processes
$!(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	send
$?(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	receive
$\mu\chi.p$	recursion declaration
χ	recursion
0	end
Σ ::=	Sessions
$(r_i : \widetilde{\tau_i} = p_i)_{i<n}$	initial role processes

Role processes can perform two communication operations: *send* (!) and *receive* (?). When sending, the process performs an internal choice between the labels f_i for $i = 0, \dots, k-1$ 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 the role for the session. On completion, a session role produces 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.

Given the role processes for a session, if the sends and receives are correctly matched, we can construct a corresponding session graph. Appendix A details this construction; the companion paper [10] also gives the reverse construction from session graphs to role processes.

We illustrate our local role syntax for the session graphs of Figure 1(a,b). 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).

```
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))
```

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:

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

Traces of the labelled semantics represent possible series of actions for these roles. For example, a complete trace for customer in session S1 is:

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

Distributed session runs At runtime, a session run involves processes 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 S begins as a principal P_0 initiates it, taking its initial role r_0, selecting other principals to play the other roles, and sending a first message. If P_0 picked the principal P_i to play role r_i, then P_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 exchange in a session is implemented as a single message exchange on the network.

As an example, a principal Alice may begin a run of the session S1 as a customer. Alice computes a unique session run identifier s, picks the principal Bob to play the role of 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 completes its part of the session. After receiving the response, Alice completes its role process and the session run s is terminated. This describes a session execution 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 all messages.

Session integrity We say that a distributed session implementation preserves session integrity if during every run, regardless of the behaviour of the malicious principals or 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 [31] (see also Appendix C).

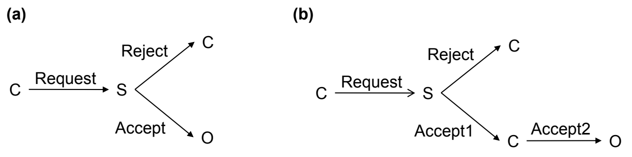

Figure 2. (a) A session graph with a vulnerable fork and (b) its safe counterpart.

However, 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 causing the compliant principals Alice and Charlie to have inconsistent session states. To avoid this attack, Charlie would typically require further cryptographic evidence from Bob.

Even if all principals are compliant, a network-based opponent could confuse them by mixing messages from different session runs, or by replaying old messages. If in our example session above, the customer Alice sends a Change to store Bob, that then sends a Reject to the officer Charlie, a network-based opponent may intercept the Reject and replay an old Offer 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 Dolev-Yao-style [14] attacks against (flawed variants of) cryptographic protocols; indeed, such protocols can be seen as particular sessions.

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 instance the session of Figure 2(a), where S may send either a Reject to C or an Accept to O. Unless C and O exchange some information, they cannot prevent a malicious S from sending both messages, thereby breaking the session specification.

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 messages are not seen by all roles, like the vulnerable session of Figure 2(a). There, the principals instantiating the roles reachable on the paths $\widetilde{f}_1$ and $\widetilde{f}_2$ may form a coalition (consisting of just S in this case) that attacks both r_1 and r_2 (C and O in the figure) by contacting them simultaneously. Nevertheless, such vulnerable session graphs can be transformed to equivalent ones that meet Property 3, at the cost of inserting additional messages. Figure 2(b) shows a safe counterpart of the vulnerable session of Figure 2(a), in which message Accept is split into two, Accept1 and Accept2, and S is obliged to contact C no matter which branch is taken. Appendix B shows the general transformation.

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 extend ML with typed sessions. We follow 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 sessions and channel-based communications. We compile programs in this language to a language without the session constructs (which we call F).

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

$E[\cdot] ::=$	Evaluation contexts
$[\cdot]$	top level
let $x = E[\cdot]$ in e_2	sequential evaluation
$s.p(E[\cdot])$	in-session evaluation (run-time only)
$P ::=$	Processes
e	running thread
$P \vert P$	parallel composition

The grammar defined by T, v, and e (except for **0** and the final three session-related constructs) generates a simple subset of ML; this is the language we call F. Remark that **0** corresponds to an expression that will not reduce anymore. 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 to the concrete F# language; our formal semantics is in fact untyped, using simply name instead of typed channels; 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.

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.

F+S embeds the session types Σ of Section 2, as follows. F+S code can define named session types $S = \Sigma$; it 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. Note that our syntax for session entry expressions is consistent with F# syntax for function application, where sessions S are implemented as modules containing functions r for each role, and message labels f are implemented as datatype constructors. In our semantics, session entry expressions reduce to active session 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. (In session entries $S.r^b$, the optional mark b will be set to $\bullet$ to mark that the session is entered by the opponent; this mark is used to specify security despite the compromise of some principals.)

As an example, the following F+S code initiates session S1 of Section 2 as a customer:

```
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 with an explicit store ρ, that keeps track of generated names and active function and type definitions, plus, for programs with sessions, session type definitions and information about running sessions. Concretely, 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 $S = \Sigma$; and running sessions $s\ \widetilde{a}\ (\delta) : S$, where $\widetilde{a}$ 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 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. $\overline{sf}, sf$), and 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), we give a small-step semantics as follows:

(APPLY) $\rho, l\ v_0 \ldots v_k \rightarrow_{\mathsf{e}} \rho, e\{x_0 = v_0; \ldots; x_k = v_k\}$
when $(l\ x_0 \ldots x_k = e) \in \rho$

(MATCH) ρ, match v with $(|v_i \to e_i)_{i<k} \rightarrow_{\mathsf{e}} \rho, e_0\gamma$
when $v = v_0\gamma$ for some substitution γ

(MISMATCH) ρ, match v with $(|v_i \to e_i)_{i<k} \rightarrow_{\mathsf{e}}$
ρ, match v with $(|v_i \to e_i)_{0<i<k}$ otherwise

(LETVAL) ρ, let $x = v$ in $e \rightarrow_{\mathsf{e}} \rho, e\{x = v\}$

(LETFUN) ρ, let $(l_i\ x_0 \ldots x_{k_i} = e_i)_{i<k}$ in $e \rightarrow_{\mathsf{e}}$
$\rho \uplus \{(l_i\ x_0 \ldots x_{k_i} = e_i)_{i<k}\}, e$
up to renamings of l_i

(TYPE) ρ, type $(t_i = \lambda_i)_{i<k}$ in $e \rightarrow_{\mathsf{e}} \rho \uplus \{(t_i = \lambda_i)_{i<k}\}, e$
where $\lambda_i = (|f_{j_i} \text{ of } \widetilde{T}_{j_i})_{j_i<n_i}$
up to renamings of t_i, f_{j_i}

(FRESH) ρ, new $() \rightarrow_{\mathsf{e}} \rho \uplus \{n\}, n$

(SEND) ρ, send $c\ v \xrightarrow{\overline{c}\ v}_{\mathsf{e}} \rho, ()$ when $c \in \rho$

(RECV) ρ, recv $c \xrightarrow{c\ v}_{\mathsf{e}} \rho, v$ when $c \in \rho$

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

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

$$(\textsc{Eval})\ \frac{\rho, e \xrightarrow{\alpha}_{\mathsf{e}} \rho', e'}{\rho, E[e] \xrightarrow{\alpha}_{\mathsf{P}} \rho', E[e']}$$

$$(\textsc{Fork})\ \rho, E[\text{fork } l] \rightarrow_{\mathsf{P}} \rho, E[()] \,|\, l\ ()$$

$$(\textsc{CommR})\ \frac{\rho, P \xrightarrow{\overline{z}\ v}_{\mathsf{P}} \rho', P' \quad \rho', Q \xrightarrow{z\ v}_{\mathsf{P}} \rho'', Q'}{\rho, P \,|\, Q \rightarrow_{\mathsf{P}} \rho'', P' \,|\, Q'}$$

$$(\textsc{CommL})\ \frac{\rho, Q \xrightarrow{\overline{z}\ v}_{\mathsf{P}} \rho', Q' \quad \rho', P \xrightarrow{z\ v}_{\mathsf{P}} \rho'', P'}{\rho, P \,|\, Q \rightarrow_{\mathsf{P}} \rho'', P' \,|\, Q'}$$

$$(\textsc{ParR})\ \frac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q}{\rho, R \,|\, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', R \,|\, Q}$$

$$(\textsc{ParL})\ \frac{\rho, P \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q}{\rho, P \,|\, R \xrightarrow{\alpha}_{\mathsf{P}} \rho', Q \,|\, R}$$

The communication rules (COMMR) and (COMML) combine a send and a receive action, which in turn may involve session transitions that modify ρ (as shown below).

For sessions, we let σ range over $S.r^b\ \widetilde{a}$ and $s.p$, that is, session entries parametrized by principals as well as running sessions.

We first define auxiliary transitions $\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p$, from role transitions $p \xrightarrow{\eta}_{\mathsf{r}} p'$ and with the same labels, in order to keep track of running sessions in the store:

$$(\textsc{Init})\ \frac{p_0 \xrightarrow{\overline{g}}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \quad s\ \textit{fresh}}{\rho, S.r_0^b\ (a_i)_{i<n} \xrightarrow{\overline{g}}_{\mathsf{s}} \rho \uplus \{s\ (a_i)_{i<n}\ \{r_0\} : S\}, s.p'}$$

$$(\textsc{Step})\ \frac{p \xrightarrow{\eta}_{\mathsf{r}} p'}{\rho, s.p \xrightarrow{\eta}_{\mathsf{s}} \rho, s.p'}$$

$$(\textsc{Join})\ \frac{\begin{array}{c} p_j \xrightarrow{f}_{\mathsf{r}} p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \\ \rho' = \rho \uplus \{s\ (a_i)_{i<n}\ \delta : S\} \end{array}}{\rho', S.r_j^b\ a_j \xrightarrow{f}_{\mathsf{s}} \rho \uplus \{s\ (a_i)_{i<n}\ (\delta \uplus \{r_j\}) : S\}, s.p'}$$

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:

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

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

$$\text{(R{\scriptsize ECV}S)}\ \frac{\rho, \sigma \xrightarrow{g}_{\mathsf{s}} \rho', s.p \quad s\ \widetilde{a}\ \delta : S \in \rho \quad \text{safe } \sigma}{\rho, \sigma\ (w) \xrightarrow{sg\ \widetilde{v}}_{\mathsf{e}} \rho', s.p\ (w.g\ \widetilde{a}\ \widetilde{v})}$$

$$\text{(E{\scriptsize ND}S)}\ \rho, s.0\ (v) \rightarrow_{\mathsf{e}} \rho, v$$

where the predicate safe σ, defined later 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

We now describe the design and interfaces of our libraries for cryptography and principals, coded as F# modules. (Formally, a F# module M is just an expression context that binds types, session types, values, and functions; we write $M\ M'$ as syntactic sugar for $M[M'[\_]]$.) 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# (see Section 2), is an important part of our security model. Its code is listed in Appendix D.

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

```
type bytes
type keybytes
val nonce: name → bytes
val hash: bytes → bytes
val genskey: name → keybytes
val genvkey: keybytes → keybytes
val sign: bytes → keybytes → 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 key, and verify checks a signature.

The concrete implementation of this library uses standard cryptographic algorithms. For example. the datatype bytes is implemented as a byte array, and sign is implemented as an asymmetric signing function (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. 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.)

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 protocol with responder roles, our protocol relies on dynamic anti-replay protection 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. In the former case, the message is transmitted. In the latter case, if the 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 is listed in Appendix D. This simple mechanism is verified within our formal model. It can be refined using any standard, timestamp-based technique to bound the size of the cache while preserving its correctness.

5 Protocol outline

We now outline the security protocol used to enforce session compliance. (Section 6 describes its compiled implementation.) We present this protocol (called Third Protocol below) as a refinement of simpler, intuitively secure protocols (First and Second protocols below), which are presented just as explanatory steps. Exploiting the session structure and the implementability condition of Section 2, our final, optimized protocol has compact messages and requires minimal message processing.

These protocols 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 each 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 $D = \text{hash}(\Sigma)$, a digest of the whole session type definition; $\widetilde{a}$, the principals assigned to the session roles; and N, a nonce freshly generated by the initiator. Every initial message also includes $\widetilde{a}$ and N.

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 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 recorded path 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 since 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 $ts = \text{sign}(s\ f\ t, \text{skey}(a))$ 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; 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 Next, we show how to avoid any dynamic graph computation. We rely on the following notion of *visibility*.

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 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 compute (at compile-time) 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 previously-received from other roles for $\widetilde{g}$, plus a new signature for f computed by the sender. Conversely, to verify a bytes 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 sizes 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.

Generated session-type interface We first generate type declarations, including a record type principals that maps roles to principals and, for each role, mutually recursive 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 the customer and store roles of session S2 in Section 2.

```
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 msgi_r types 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, the nonce (used in the session identifier), the 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.

```
type tsig = { tstime: int; tsval: bytes }
type tsigs = { [ r: tsig; ] for each (r : τ̃ = p) ∈ Σ }
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_g̃_f: state ∗ payload(f) → bytes
val gensig_p_f: state → bytes
```

The function gensig_p_f computes a signature ts for label f with stored timestamp 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_g̃f_g̃' : state ∗ bytes → state ∗ payload(g̃')
```

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.

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 the 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 (the text in italics specifies how the compiler produces the code):

```
*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* $r, r', \mathcal{F}$ *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* $\widetilde{g}'$ *s.t.* $\widetilde{g}f + \widetilde{g}'$ *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 st and msg = f(v,w) as parameters, 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 the function verify_$\widetilde{g}f$ st on the message it has received. This function 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, with a **0** expression.

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 that initialize the state when a role initiates or joins the session:

```
val init: principals $\rightarrow$ unit state
*for the initiator role* $r_0$:
[let $r_0$ prins msg =
   let st = init prins in send_$\emptyset$ st msg ]
val join: bytes $\rightarrow$ 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_$\emptyset$ 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. (The labelled semantics is also used to structure the security proofs.) So, we relate the behaviour of high-level processes, of the form $L\ \widetilde{S}\ U\ O$, to their implementations $L\ M_{\widetilde{S}}\ U\ O'$, where

- L consists of the symbolic libraries of Section 4;
- $\widetilde{S}$ is a series of session declarations; $M_{\widetilde{S}}$ is their implementation, as in Section 6;
- U represents code that use sessions but does not access Prins;
- O represents an F+S opponent that can access the “opponent” interface of Prins, including recv$^\bullet$ and skey$^\bullet$, but not recv and skey and may use $S.r^\bullet$ session entries;
- O' is similar to O at the F level, with similar assumptions; in the implementation, we assume that O' does not access $M_{\widetilde{S}}$—this entails no loss of generality, since O' may in particular include its own copy of our implementation.

The conditions on U, O, and O' can be checked easily (e.g. by typing). Their code can freely 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.

Our first security theorem is stated in terms of may testing. As 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, can not deviate from high-level F+S configurations, where sessions are ideally followed as prescribed by the session semantics of Section 3.

Conversely, we can check that the theorem does not hold if we relax our secure implementability condition. For instance, consider the session of Figure 2(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 discussed 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 both of them 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. Appendix E lists the concrete user code for conducting this test and reporting the attack.

Our second security theorem is more precise; it provides an explicit 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.

Labelled transitions for modelling the adversary In the transitions of Section 3, we do not maintain scope for the sessions and values available to the opponent, and maintain a global store ρ instead, using interfaces to ensure that the opponent code cannot access some values. Now, we introduce labelled transitions with an abstract environment, and keep track of the values and sessions available to that environment. We let K represent this knowledge and capabilities. 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, as well as any value exported by U. 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.

For all the active sessions recorded in ρ, high-level Ks also record the state of all 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 active session s of ρ, or when $K = K'$ and $\sigma = S.r_i^\bullet\,\widetilde{a}$ with safe $a_i = \text{false}$.

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

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

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

$$\text{(KRecv)}\ \frac{\rho, P \xrightarrow{c\ v}_{\mathsf{P}} \rho, P' \quad c, v \in \mathsf{Val}(K,\rho)}{K, \rho, P \xrightarrow{c\ v}_{\mathsf{K}} K, \rho, P'}$$

$$\text{(KStep)}\ \frac{\rho, P \rightarrow_{\mathsf{P}} \rho', P'}{K, \rho, P \rightarrow_{\mathsf{K}} K, \rho', P'}$$

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

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

$$\text{(OStep)}\ \frac{\rho, \sigma \xrightarrow{\eta}_{\mathsf{s}} \rho', s.p}{K[\sigma], \rho \xrightarrow{s\eta}_{\mathsf{o}} K[s.p], \rho'}$$

$$\text{(OComm)}\ \frac{K, \rho \xrightarrow{s\overline{g}}_{\mathsf{o}}\xrightarrow{sg}_{\mathsf{o}}\xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}{K, \rho \xrightarrow{s\overline{f}}_{\mathsf{o}} K', \rho'}$$

Rule (KApply) lets the environment apply functions; its hypothesis requires that the function be pure, and that both the function and its arguments are known to the environment. (In addition, function with side effects, or calls from P to the environment, may be modelled using channel-based communications.) Rules (KSend) and (KRecv) similarly represent channel-based communications with the environment. Rule (KStep) enables P to make 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.

We let $\rightarrow_{\mathsf{KD}}$ denote a transition among the subset of the $\rightarrow_{\mathsf{K}}$ transitions that are silent. We write $\overset{\varphi}{\Rightarrow}_{\mathsf{K}}$ for a series of transitions where the observable transitions (with series of labels φ) are interleaved with any number of silent ones.

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.

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 $\widetilde{n}$ 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, calls only pure functions of ρ, and whose values defined in ρ are all included in K.*

Relating abstract and compiled sessions at runtime The state of a role implementation in F is not entirely determined by the role process in F+S; in addition to S and s recorded in ρ, and $s.p$ 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 information: T is initially empty; for every $s\ (a_i)_{i<n}\ \{\widetilde{r}\} : S$

in ρ, $T(s)$ provides a term N_s in K and a path in the session graph of S, decorated with strictly-increasing integers, such that, for all roles r_i attributed to safe principals safe a_i, there is a running session role process $s.p$ if and only if the role has received a message in the path, and p is the last role process for that role of the path. Also, T records, for each safe principal, the state of their cache. Finally, T records, for each receiving role, whether a bad input has been received so far—in that case, our session implementation for the role has silently terminated.

We define a translation $[\![\cdot]\!]_T$ from F+S session expressions to implementation states (F expressions and context). In stores ρ, we replace every session type definition S with the types and function definitions of M_S, remove every session entry s, and, if s is initiated by a safe principal, add a fresh nonce N_s. In processes and expressions, we translate only active session roles, as follows:

$$\begin{array}{rcl}
[\![s.p(e)]\!]_T & = & \text{let}\, x_s = [\![e]\!]_T \text{ in S.send\_}\widetilde{g}\ st\ (x_s) \\
 & & \text{when } p \text{ is an output} \\
[\![s.p(w)]\!]_T & = & \text{S.recv\_}\widetilde{g}\ st\ w \quad \text{when } p \text{ is an input} \\
[\![s.p(w)]\!]_T & = & \mathbf{0} \\
 & & \text{when } p \text{ is an input, after receiving a bad input} \\
[\![s.p(e)]\!]_T & = & [\![e]\!]_T \quad \text{when } p \text{ is } \mathbf{0}
\end{array}$$

where $s.p$ must be the last node for the role of p, denoted $r(p)$, in the trace of s in T, $\widetilde{g}$ is the sequence of labels visible from $s.p$ for this trace, and st is computed from N_s and $T(s)$ (including valid $tsig$s for $\widetilde{g}$). Expressions of the form $S.r\ldots$ are unchanged, but now interpreted as function calls, rather than primitive session entries. The translation also adds to the original processes the forward process used by the adversary to communicate and the processes involved in the cache management for all safe principals (as described in Appendix D). In K, we replace high-level session records for s with the session nonces recorded in T, and we add all the signatures from safe principals built from $T(s)$, as defined in function gensig_p_f, that are visible from any role on the path that is instantiated to an unsafe principal.

Implementation soundness for transitions We let $\rho_{L\widetilde{S}}$ be defined by the deterministic reductions $\emptyset, L\ \widetilde{S}\ [\_] \rightarrow_{\mathsf{P}}{}^{*}$ $\rho_{L\widetilde{S}}, [\_]$. An F+S configuration $H = K, \rho, P$ is *valid* when ρ includes $\rho_{L\widetilde{S}}$; the names of P are defined in ρ and do not include Prins names; the values in K defined in $\rho_{L\widetilde{S}}$ are built from the library interfaces of Section 4; the session types in ρ are valid; and the sessions in ρ have a session role in P for each safe principal and a session role in K for each unsafe principal, such that these roles are reached on a path in the graph of their session types, with a last label with (at least) a safe sender or a safe receiver.

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 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$^{\bullet}$.

The following theorem states that all low-level events on the network can be explained by the high-level semantics, thereby ensuring that attackers do not get anything from trying to break the sessions at the low-level.

Theorem 2 *Let W be a valid implementation of H. For all transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, where φ represents the observable actions of these transitions, there exists W° valid implementation of H° such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^{\circ} \rightarrow^{*}_{\mathsf{KD}} W''$, $W' \rightarrow^{*}_{\mathsf{KD}} W''$, and $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^{\circ}$ with φ a translation of ψ.*

8 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 are a rich area of study [7, 8, 9, 13, 16, 30, 32], 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 [18].

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, F+S already enables this approach, as principals may run other protocols on communication channels, but does not offer linguistic support for them.) However, such extensions would also unavoidably complicate our security model for session programmers.

Acknowledgments This work benefited from discussions with Cosimo Laneve and Jean-Jacques Lévy.

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. In *ACM Workshop on Secure Web Services (SWS)*, pages 11–22, Oct. 2004.

[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*, Sept. 2006.

[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. 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, P.-M. Dénielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. At `http://www.msr-inria.inria.fr/projects/sec/sessions/`, 2007.

[11] V. Cortier, B. Warinschi, and E. Zalinescu. How to protect security protocols against active attackers. Unpublished draft.

[12] P.-M. Deniélou and J. J. Leifer. Abstraction preservation and subtyping in distributed languages. In *11th International Conference on Functional Programming (ICFP)*, 2006.

[13] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *20th European Conference for Object-Oriented Languages*, July 2006.

[14] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[15] D. Duggan. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop (CSFW)*, 2002.

[16] 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.

[17] 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.

[18] 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.

[19] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[20] 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.

[21] 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.

[22] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[24] Objective Caml. At `http://caml.inria.fr`.

[25] 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.

[26] 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)*, Sept. 2005.

[27] 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.

[28] D. Syme. *F#*, 2005. At `http://research.microsoft.com/fsharp/`.

[29] 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.

[30] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.

[31] 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.

[32] 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.

[33] 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.

A From role processes to session graphs

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.

B Transforming 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:

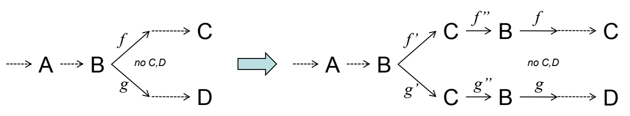

Figure 3. Eliminating blind forks

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

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 3 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.

C Session integrity and correspondences

Session graphs enforce causal relationships between message events. In cryptographic protocol analyses, such relationships are typically written as injective correspondence properties [31]. Indeed, from a session graph, one can read a series of injective correspondences that hold in any session run. Our session integrity theorem (Theorem 1) guarantees that every correspondence read from a session graph holds for compliant principals.

Consider, for example, session (c) in Figure 1. The following are some of the injective correspondence properties that hold in a session run where the principals P_c, P_s, and P_o play the roles C, S, and O, respectively:

- If P_c and P_s are compliant, for each Offer message accepted by P_c, P_s must have sent one.
- If P_s and P_o are compliant, for each Abort message accepted by P_o, P_s must have sent a Reject message to P_c.
- If P_o is compliant, then it never accepts both an Abort and a Confirm message.

These correspondences correlate only two message events occurring on a path. More generally, one can write nested correspondences for sequences of messages on a path in the graph, also enforced by session integrity. On the other hand, some other session-integrity properties (with mutually-exclusive events, or loops) are not expressible as correspondence properties.

D Symbolic code for the libraries

In this appendix we provide the symbolic implementations for the libraries described in Section 4. Our code relies on syntactic sugar: if ... then ... else is a shortcut for standard pattern-matching on the result of the test; the semicolon, 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$^\bullet$ 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$^\bullet$, chans$^\bullet$, and skeys$^\bullet$. Our generated protocol implementations access safe, skey, vkey, psend, and precv. User code is given access only to principal constants.

E Counter-example for Theorem 1 without enforcing Property 3

We list the concrete code for the two compliant roles for the counter-example described below Theorem 1 for the session S given in Figure 2(a), which violates Property 3. Let U be the process:

```
let pr =
  { client = "Alice"; server = "Eve"; officer = "Bob"; }
let x = new()
let acceptbranch _ _ = send x "OK"
let rejectbranch pr′ _ =
  if pr = pr′ then let _ = recv x in send ω ()

let office () = S.officer "Bob" {hReject=rejectbranch} in
fork office;
S.client pr (Request (42,{hAccept=acceptbranch}))
```

In this code, Alice plays the client role and Bob plays the officer role for at most one run of the session. These compliant principals synchronize using the side communication channel x only when they both receive Accept and Reject messages. In that case, Bob fails with ω. This behaviour would be enabled in the F implementation but not in F+S.