A Protocol Compiler for Secure Sessions in ML

Ricardo Corin[1,2] and Pierre-Malo Deniélou[1]

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

Abstract. Distributed applications can be structured using *sessions* that specify flows of messages between roles. We design a small specific language to declare sessions. We then build a compiler, called `s2ml`, that transforms these declarations down to ML modules securely implementing the sessions. Every run of a well-typed program executing a session through its generated module is guaranteed to follow the session specification, despite any low-level attempt by coalitions of remote peers to deviate from their roles. We detail the inner workings of our compiler, along with our design choices, and illustrate the usage of `s2ml` with two examples: a simple remote procedure call session, and a complex session for a conference management system.

1 Sessions for distributed programming

Programming networked, independent systems is complex: when systems communicate through an untrusted network, and do not trust each other, enforcing security properties is hard. As a first step to simplify this task, programming languages and system libraries offer abstractions for common communication patterns (such as private channels or RPCs). Beyond simple abstractions for communications, distributed applications can often be structured as parties that exchange messages according to some fixed, prearranged patterns, called *sessions* (also named contracts, or workflows, or protocols). Sessions simplify distributed programming by specifying the behaviour of each network entity, or *role*: the parties can then resolve most of the programming complexity upfront.

Language-based support for sessions is the subject of active research [2,3,4,7,10,14,15]. Several of these works focus on developing type systems which statically ensure compliance to session specifications. There, type safety implies that user code that instantiates a session role always behaves as prescribed in the session. Thus, assuming that every distributed program participating in a session is well-typed, any run of the session follows its specification. There, being well-typed implies that every session participant is benign, and therefore complies with the session specification. Moreover, the network is also assumed to behave as expected, (e.g., delivering messages correctly).

However, in an adversarial setting, remote parties may not be trusted to play their role. Moreover, they may collude to attack compliant participants, and may also control the network, being able to eavesdrop, intercept, and modify en route messages. 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 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.

In order to keep sessions being useful and safe abstractions, we consider their secure implementation in terms of cryptographic communication protocols, by developing `s2ml`. To our knowledge, our compiler `s2ml` is the first to systematically compile session specifications to tailored cryptographic protocols, providing strong security guarantees beyond simple functional properties.

In ongoing work [5], we explore language-based support for sessions. We design a small language for specifying sessions, and identify a secure implementability condition. We present a formal language extending ML [11,12] with distributed communication and sessions, designed in a way so that type safety yields functional guarantees: any sent message is expected by its receiver, with matching payload types. Then, we develop the `s2ml` compiler that translates sessions to cryptographic communication protocols, and formally show, as main result, that programs are shielded from any low-level attempt by coalitions of remote peers to deviate from their roles. In that work, we are most concerned about establishing the correctness of the code generation, and illustrate the approach with a small, simple toy example.

In this paper, we turn to present the details of our implementation. We focus on presenting our compiler `s2ml`, along with its usage and inner workings. Furthermore, we investigate the applicability and scalability of our approach to more realistic and complex settings through the study of a RPC session and a conference management system (CMS) session example.

Architecture. The basis of our work is a language for sessions with a CCS-like syntax to describe the different roles in a session. The `s2ml` compiler reads the session declarations, and works as follows: First, it checks correctness and security conditions on every session declaration, using an internal graph-based, global representation of the message flow. Then, it generates an ML module (along with its interface) for each specified session. The interface provides the programmer with the functions and types needed to execute every session role.

We rely on the ML language for several reasons. First, we take advantage of ML's typechecking to ensure functional correctness (i.e., that user code follows the session as prescribed), as opposed to having a dedicated type system as in other session types approaches. Second, our generated session role functions have (usually mutually recursive) types which are driven by user code using a continuation passing style (CPS) which allows for compact session programming. Finally, our generated types and cryptographic protocols heavily use algebraic types and pattern matching to specify and check the different allowed session paths. Our generated code uses the Ocaml syntax[3] and can be run in both Ocaml and F# [13].

Programs using the generated session interfaces can be linked against networking and cryptographic libraries, obtaining executable code. We provide three alternative implementations for these libraries: two concrete implementations using either Ocaml/OpenSSL and F#/Microsoft .NET produce executable code supporting distributed runs; a third, symbolic library implements cryptography using algebraic datatypes and communication via a Pi calculus library, useful for correctness checks and debugging.

[3] Although we use Ocaml syntax, our work can easily be adapted to other ML dialects.

Contents. Section 2 presents the session language that serves as input to `s2ml`, and introduces the examples. Section 3 illustrates the usage of sessions used by programmers to develop secure distributed applications, by coding the roles the RPC and CMS examples. Section 4 presents our security property, called session integrity, along with several threats our implementation needs to guard against. Section 5 focuses on the compiler `s2ml`: first it describes its inner workings, then it illustrates generated output for examples, and finally, it presents some performance measurements. Section 6 concludes. Additional code excerpts of the examples' output and generated code is in Appendix A and B. The project website [6] contains additional information.

2 Specifying Sessions

A *session* is a static description of the valid message flows between a fixed set of roles. Every message is of the form $f(v)$, where f is the message descriptor, or label, and v is the payload. The label indicates the intent of the message and serves to disambiguate between messages within a session. Labels are also used as ML type constructors (and are thus expected to start with a capital letter).

We denote the roles of a session by $\mathcal{R} = \{r_0, \ldots, r_{n-1}\}$ where $n \geq 2$. 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.

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:

Global graph The graph describes the session as a whole and is convenient for discussing security properties and the secure implementability condition. Briefly, a session graph consists of nodes representing global states that are annotated with the corresponding active role (the role sending the next message), and edges between nodes labelled with message labels and the types of their payloads.

Local roles Local role processes are the basis of our implementation: they describe the session from each role's point of view. They thus provide a direct typed interface for programming roles, and constitute our language for sessions.

2.1 A Language for Sessions

Our language for sessions has a CCS-like grammar for expressing local roles processes:

$\tau ::=$	Payload types
unit | int | string	base types
$p ::=$	Role processes
$!(f_i\!:\!\tau_i\ ;\ p_i)_{i<k}$	send
$?(f_i\!:\!\tau_i\ ;\ p_i)_{i<k}$	receive
$\mu\chi.p$	recursion declaration

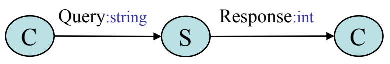
χ	recursion
0	end
Σ ::=	Sessions
$(r_i : T_i = p_i)_{i<n}$	initial role processes

Role processes can perform two communication operations: *send* (!) and *receive* (?). When sending, the process performs an internal choice between the labels f_i for $i = 0, \ldots, k-1$ and then sends a message $f_i(v)$ where the payload v is a value of types τ_i (for convenience, we consider only the basic unit, int and string types which simplify marshalling). 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 a value whose type T_i is specified in the process role $r_i : T_i = p_i$. For the return type T_i, we accept any ML type.

For convenience, we omit the trailing semicolon and 0 process at ending points. Also, our concrete syntax uses the keyword 'mu' for μ and keywords 'session' and 'role' in front of session and role definitions.

2.2 Example A: Remote Procedure Call

Figure 1 (top) shows a session graph for a simple RPC exchange, in which the client role, called **C**, sends the server role **S** a **Query** message (of payload type string), who answers with a **Response** message (of payload type int). The bottom part of the figure specifies the RPC session in terms of local role processes, using the above grammar. After naming the session as **Rpc**, the two roles are defined with a return type and their local message flows: the client sends a **Query**, then expects a **Response** and finally returns an int; the server waits for a **Query**, then sends a **Response** and finally returns unit. These three lines are the actual input of our compiler.



```
session Rpc =
  role client:int = !Query:string; ?Response:int
  role server:unit = ?Query:string; !Response:int
```

Fig. 1. Session graph and Local roles for an RPC (file `rpc.session`)

2.3 Example B: a Conference Management System

We now describe a session for a conference management system. Although this system is rather simplified from a real life implementation, we believe it's significantly large in comparison with other case studies attempted in the session types literature.

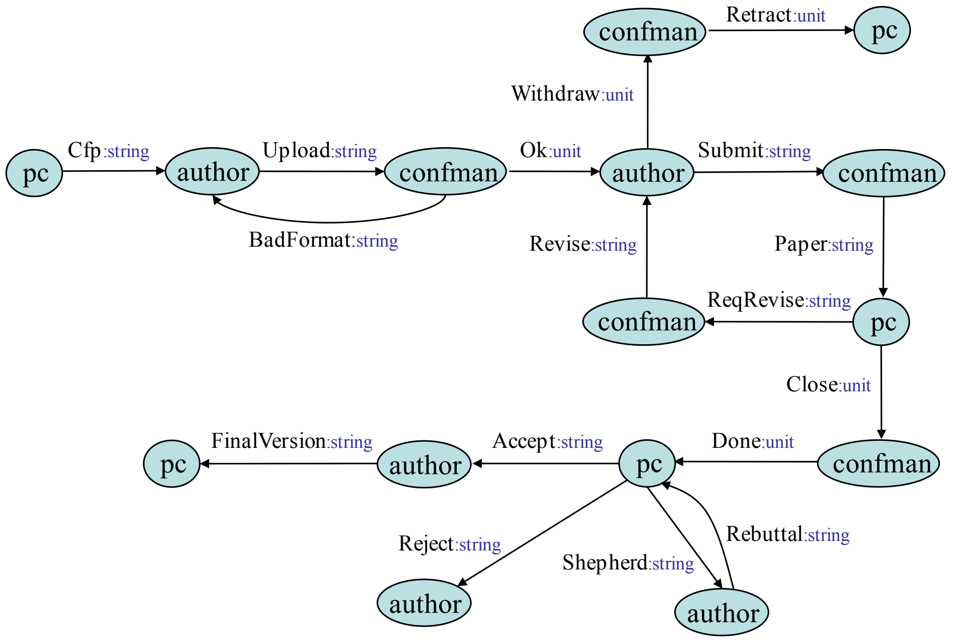


Fig. 2. A Conference Management System (CMS): Global graph

Global description. Figure 2 shows the graph of a CMS session. There are three roles: **pc** (the conference organizer), **author**, and **confman** (the submission manager). All messages carry as a payload either a string value (which is used for the call for papers, paper submissions, and so on), or a unit value, when no payload is necessary.

The session proceeds as follows. Initially, the program committee pc sends a call for papers message, **Cfp**, to the prospective author. The author then uploads a draft by sending an **Upload** message to the conference manager confman, who checks whether the draft meets the conference format (e.g., style format or compliance with the size). If the format is invalid, the confman replies to the author with a **BadFormat** message, with an explanation; at this point we have a loop in which the author can fix the draft and try again. Eventually the format is valid, and the confman replies with an **Ok** message. Now the author can submit a paper by sending a **Submit** message to the confman. Alternatively, it can choose to refrain from submitting a paper by sending a **Withdraw** message, which the confman communicates to the pc by sending a **Retract** message. If the author indeed submitted a paper, the confman forwards it to pc, who then will evaluate it. The pc can ask the author to revise the paper, by sending a **ReqRevise** message to the confman which will in turn send a **Revise** message to the author. This phase can loop until eventually the pc reaches a decision, and asks the confman to stop receiving revisions by sending a **Close** message. The confman answers with a **Done** message, and then the pc can notify the result to the author, enclosing possibly reviews for the paper. The notifications are either acceptance of the paper (sending an **Accept** message), or rejection (sending a **Reject** message), or a decision to exceptionally ‘shepherd’ the paper (sending a **Shepherd** message), in which the author can support her submission by sending a **Rebuttal**. This can again loop until the pc decides a final verdict, i.e. either accepting or rejecting the paper. In the case of acceptance, the author sends the pc a final version of the paper.

Local processes. Figure 3 presents the counterpart of the CMS graph from Figure 2 in terms of local roles. We illustrate local roles by describing in detail the behaviour of the author role. From the author’s point of view, the session starts by receiving a **Cfp** message. A recursion point called **reformat** is created, and then the author checks the paper by sending an **Upload** message. If a **BadFormat** message is received, execution jumps back to the **reformat** point. If an **Ok** message is received, the author sets a recursion point called **submission** and then choose to either send a **Submit** or a **Withdrawal** message. For the latter, execution ends. For the former, another recursion step **discuss** is set, and several messages can be expected: either an **Accept**, in which the author ends by sending a **FinalVersion**, or a **Reject** which also ends execution, or a **Shepherd** message to which the author replies with a **Rebuttal** and then jumps back to **discuss**; finally, a **Revise** message may also be received, in which the author jumps back to **submission**.

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

Fig. 3. A Conference Management System: Local role processes (file `cms.session`)

3 Programming with sessions

Once we know how to specify sessions, we are now ready to use them by instantiating the different session roles by actual principals. We start by describing how principals are defined. Then we program the simple RPC session from the previous section, and finally we consider the more challenging case of the CMS session.

Principals. Principals are the network entities that instantiate session roles and specify networking information (i.e., IP address and port) and cryptographic credentials (X.509 certificates and private keys), for message delivery and security. Hence, when a programmer wants to initiate or join a session, she must register the principals in a local store used by our implementation. To this end, we provide a library for managing principals called **Prins**, with a **register** function, which when invoked as **register id filename inet port** registers a principal called **id**, whose credentials are in the file **filename**, IP address is **inet** and port is **port**.

For example, the programmer's source code for the CMS example that involves three participants includes the following calls:

```
let _ = Prins.register "alice" "alice.cer" "193.55.250.70" 8765
let _ = Prins.register "bob" "bob.cer" "193.55.250.71" 8765
let _ = Prins.register "charlie" "charlie.cer" "193.55.250.72" 8765
```

Files containing cryptographic credentials have to include an X.509 certificate, plus optionally the corresponding private key. Thus, user code can register both the running principal of a session role by including both keys (which the generated protocol will use to sign and verify messages) and other principals running the session, by registering only their certificates (which are used to verify other principals'signatures).

3.1 Programming an RPC session

Initially we invoke our compiler with file `rpc.session` from Figure 1. Two files, called `Rpc.ml` and `Rpc.mli`, are created by `s2ml`. The former is the generated module implementing the RPC session, while the latter is its interface:

```
type principal = string
type principals = {client:principal; server:principal}

type result_client = int
type msg0 = Query of (string∗msg1) and msg1 = {hResponse:(principals→ int→ result_client)}
val client : principals→ msg0→ result_client

type result_server = unit
type msg3 = {hQuery: (principals→ string→ msg4)} and msg4 = Response of (int∗result_server)
val server : principal→ msg3→ result_server
```

The record type principals is used to instantiate roles with principals at runtime. Function **client** runs the session as the client role; when invoked, user code needs to provide:

1. a principals record populating the roles (since the client role is the session initiator, it can choose the session participants); and

2. a continuation (of type msg0) which drives the client role (our programming discipline relies on a CPS style, see below Section 3.2); here, it sends a **Query** message consisting of a payload to be sent (of type string) and a continuation message handler (of type msg1), which processes the answer **Response** message.

The server is symmetric, except that as responder it only needs to choose its identity.

We can easily program this RPC session; here's the code for a client that runs as `alice`, contacts `bob`, as the server, with a **Query** "Number?", and prints the response (we assume the principals registered as described above):

```
open Rpc
... (* register principals *)
let prins = {client = "alice"; server = "bob"}
let answer = client prins (Query("Number?",{hResponse = fun _ i → i}))
let _ = Printf.printf "Answer is %i\n" answer
```

A programmer runs a session (as role client) by calling function **client** providing a record instantiating roles to principals, and a continuation that sends and processes incoming messages. The first message (of type msg0) has to be sent by client, modelled by constructor **Query** which awaits for a payload and a continuation. Since the client then waits for a reply, the programmer has to provide a function handler for each of the possible incoming messages, those functions acting as continuations: here only one continuation is required (since only a **Response** may arrive) and the record has thus only one field labelled **hResponse**. The continuation has to be a function of two arguments: the first is the vector of principals involved in the session and the second is the payload of the corresponding message.

Here's the code for a server ignoring the query content and responding with '42':

```
open Rpc
... (* register principals *)
let _ = server "bob" {hQuery = fun _ _ → Response(42,())}
```

From the session programmer's point of view, sending a message is as simple as returning a constructed type with the right payload and continuation: **Response**(42,()). Here the continuation is simply unit as the session ends and any value of type **result_server** (which is above defined as unit) will do. All the rest is taken care by the module **Rpc** generated by `s2ml`, like message formatting, cryptographic signing, and routing.

Finally, in order to obtain an executable, we compile this user code with `Rpc.ml` and libraries for implementing cryptographic operations (like hashing and signing) and networking. Appendix A shows the output of the client for an execution.

3.2 Session programming and CMS example

We run `s2ml` with the CMS example of Figure 3 for the **Conf** session, on file `cms.session`. This produces files `Conf.ml` and `Conf.mli`. As in the RPC example, the interface `Conf.mli` contains a specialized principals record plus generated types and functions for each role (here we show only the ones for the author role):

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

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

4 Session Security

At run time, a session is executed by processes running on hosts connected through an untrusted network. Each process runs on behalf of a principal. In order to state our security property, called session integrity, we first describe the threat model, and then informally discuss session integrity and possible threats to it.

Threat model. We consider a variant of the standard Dolev-Yao threat model [8]: the attacker can control corrupted principals (that may instantiate any of the roles in a session, and do not necessarily run as specified by the session declaration nor use our compiler), and perform network-based attacks: intercept, modify, and send messages on public channels, and perform cryptographic computations. Moreover, the corrupted principals may collude between themselves and the network during an attack. However, the attacker cannot break cryptography, guess secrets belonging to compliant principals, or tamper with communications on private channels.

Session Integrity. We say that a distributed session implementation preserves session integrity if during every run, regardless of the behaviour of the attacker, the process states at compliant principals (which use the generated cryptographic protocols as detailed in the next section) are consistent with a run where all principals seem to comply with all sessions. (This informal notion is made precise in [5]; see also below.)

Session integrity requires that all message sequences exchanged by compliant principals are consistent and comply with the session graph, that is, every time a compliant principal sends or accepts a message in a session run, such a message 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 is silently dropped, or reliably detected as anomalous.

In order for our compiler `s2ml` to enforce session integrity, it must generate a cryptographic protocol for each compliant principal that can guard against several possible attacks. We illustrate next some of these attempts to break integrity, and how the generated cryptographic protocol prevents them.

Session identifier confusions Each session instance needs to have a unique session identifier, as otherwise there could be confusions between different running sessions. The generated protocols compute a unique session identifier as $s = \textbf{hash}(D\widetilde{a}N)$, where $D\,\widetilde{a}\,N$ is the tagged concatenation of $D = \textbf{hash}(\Sigma)$, a digest of the whole session declaration, $\widetilde{a}$, the principals assigned to the session roles; and N, a nonce freshly generated by the initiator. Including D prevents confusions about the specification of the session being executed; including $\widetilde{a}$ prevents confusions about which principal is executing which role; and including N prevents confusion with other running session instances of the same declaration Σ and principal assignment $\widetilde{a}$. Messages sent by our generated cryptographic protocols always include as header the session identifier s, plus, in initial messages, $\widetilde{a}$ and N to allow receivers to recompute s (we assume D is expected and known by receivers). For example, for our CMS example, the generated protocol computes D as the hash of the session declaration from Figure 3, $\widetilde{a}$ = charlie alice bob (indicating that `charlie` plays the **pc** role, `alice` the **author** and `bob` the **confman**), and N is a random nonce.

Message integrity attacks Whenever a principal playing a role in a session receives a message corresponding to a path executed in the session graph, it needs to ensure every label in the path has been sent by the presumed principal. Otherwise, an attack is possible, where some principal is impersonated by the attacker: for example in Figure 2, a malicious author could send the confman an **Upload** message even though the pc never sent a **Cfp**; if the confman does not check the presence of the pc, session integrity is violated. In order to prevent these attacks, the generated protocols include in messages a series of cryptographic signatures[4]: 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).

For our CMS example, consider the first time that the confman role gets contacted with an **Upload** message in Figure 2. At that point, the generated protocol needs to check signatures from the principals playing the roles author and pc; for our running session with session identifier as above, an incoming message is accepted by `bob` as **confman** only if it includes a signature from `charlie` (as role **pc**) of a **Cfp** message, and another signature from `alice` (as author) of an **Upload** message. On the other hand, if `bob` as **confman** is at the same node contacted again (e.g., because `bob` sent a **BadFormat** message and entered a loop), in the next incoming message `bob` needs to only check a (new) **Upload** message from `alice`, and the **Cfp** message needs not be forwarded again, as `bob` already checked it. The

[4] Cryptographic (or *digital*) signatures ensure the sender authenticity, as the signing private key of a compliant principal is kept secret.

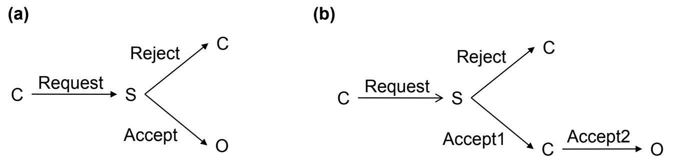


Fig. 4. (a) A session graph with a 'blind fork' and (b) its safe counterpart.

compiler accounts for both situations, and outputs accordingly specifically tailored functions for message generation and verification.

Intra- and Inter-session replays Message replays can also thwart session integrity. Three situations can happen: (1) a message from one running session can be injected into another running session; (2) an initial message involving a principal can be replayed, trying to re-involve the same principal twice; and (3) a message from one running session can be replayed in the same running session (e.g., messages inside loops, which are particularly vulnerable).

Whilst (1) is directly prevented using a unique session identifier as detailed above, (2) and (3) need special treatment. For the former, like any protocol with responder roles, our generated 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. For the latter, our generated protocol includes a logical timestamp for messages inside loops, that is incremented at each loop iteration; it thus disambiguates messages occurring in cycles (messages not occurring in loops are not vulnerable, as message labels are assumed to be unique, see below).

Valid Sessions Not every session encodable using the language of Section 2 makes sense: for example, a role sending a message that is never received is clearly undesirable. Our compiler checks this and other syntactic conditions that a session has to satisfy in order to be implementable (see Section 5). In particular, the compiler checks the absence of 'blind forks', which are in fact a security threat to session integrity. Consider for instance the session of Figure 4(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. (In fact, any graph containing the one in Figure 4(a) as subgraph is vulnerable!)

Nevertheless, such vulnerable session graphs can be transformed to equivalent ones without forks, at the cost of inserting additional messages. Figure 4(b) shows a safe counterpart of the vulnerable session of Figure 4(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. (The general transformation is not difficult to build [5].)

Proving Session Integrity 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.* [1], 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 (we have both Ocaml/OpenSSL and F#/Microsoft .NET implementations). 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.

In order to formally state and prove session integrity, we develop a high-level semantics that enforces sessions following their specification [5]. Our compiler, in turn, transforms session declarations to modules implementing them. Thus, we have two possible semantics in which user code runs: either a high-level configuration (where sessions execute as prescribed by definition) and a low-level configuration, in which user code executes calling the session-implementation modules. Our main security result (Theorem 1 in [5]), stated in terms of may testing, expresses that any behaviour of a low-level configuration can be simulated by a corresponding high-level configuration. Hence, the cryptographic protocol implementing the session is not letting an adversary gain anything, as *any* possible behaviour of session implementations using our compiler interacting with an adversary (comprising of corrupted principals colluding with the network) can be also reproduced by an adversary that does not interact with session implementations, and is subject to semantics where sessions run as prescribed.

5 Compiling Sessions to Modules

In Section 3 we present the interface generated by `s2ml`, so that programmers can use sessions. In this section, in turn, we discuss the inner workings of the compiler, i.e., how `s2ml` generates a cryptographic protocol securely implementing the session, and preventing possible threats to session integrity as detailed in the previous section. Our compiler `s2ml` works as follows:

1. For each session definition using local roles, it transforms it to a global graph and checks several well-formed and implementability conditions on it. From this graph, it also generates visible sequence messages which are used by the code generation phase.
2. Then, the compiler generates for each session its corresponding cryptographic protocol, and emits both its interface and its code as an ML module.

Checking validity conditions and generating visible sequences. As the session specifications are written in term of local role processes, and since a global view is required, the compiler first tries to generate the graph version of the session. Following the flow of the session (starting from the first role and messages), `s2ml` verifies that all the sent messages are expected by someone (i.e., are among the messages declared to be possibly received by a different role). Each node of the graph thus corresponds to a given active role and the edges are the messages sent to a different role which, after reception of the message, becomes active.

This conversion checks the correctness and coherence of the session declaration. In particular, we rule out invalid sessions in which messages are sent but not expected, and self-sent messages. We also require that labels are unique: two different edges cannot have the same label. This ensures the intent of each message label is unambiguous: the label uniquely identifies the source and target session states.

As explained in the previous section, branching in itself can lead to a security risk. The minimal condition to avoid this kind of attacks can be formulated in the following way (see [5] for details): For any two paths in the graph starting at the same node and ending with roles r_1 and r_2, we require that if neither r_1 nor r_2 are in the active roles of the two paths (i.e., they don't send any of the messages), then $r_1 = r_2$. Checking this property is done in the `s2ml` implementation by a careful look at branching nodes: lists of active roles are recorded on every path starting at these nodes, followed by a comparison that ensures that the roles in different branches are related.

As an additional output, from this global graph `s2ml` generates the DOT [9] graph of the session graph, which can be used to view the specified session.

Visibility. After checking that the graph is valid and safe, `s2ml` generates the *visible* sequences, an essential part of the generation of the cryptographic protocol. Briefly, a sequence of labels is visible at a given node in the session graph if it contains only the last label sent by every other role. This notion is used in minimizing the number of signatures checks at runtime in the generated implementation: it relies on the fact that only the latest labels sent by every other role have to be checked to ensure session integrity. We compute the visible sequences at compile-time to avoid any graph computation at runtime: the runtime signature checks which rely on visible sequences can thus be efficiently performed. For example, in the CMS example of Figure 2, the node in which the confman role is first contacted by an **Upload** message has two visible sequences, **Cfp**-**Upload** (along the initial path) and just **Upload** (through the cycle).

Generating the session interface and implementation. The main difficulty in the interface generation is to produce the set of recursive types that specify the alternation of constructed messages and continuations required from the user.

The generation of these types is based on four principles: first, an internal choice is translated into an algebraic sum type where message labels are used as constructors and where the constructor expects a correct payload and a continuation corresponding to the role's next expected message; second, an external choice generates a record whose labels are derived from message labels and whose data are functions handlers for the incoming messages (those functions take as arguments the record of principals and the payload of the message); third, mutual recursion reflects a recursive point in the local role description; forth, when ending, the result type is used.

More formally, our algorithm first associate type names to each of the sub-processes of a given role process: the names are of the form **msg**n (below we call this function *name*). The **0** sub-process is a particular case and its associated type name is of the form **result**_*rolename*.

Then we have the following generating function that is applied successively to all sub-processes:

$$[\![!(f_i : \tau_i \; ; \; p_i)_{i<k}]\!] = \text{and } \mathit{name}(p) = \{ \mid f_i \text{ of } (\tau_i {*} \mathit{name}(p_i))\}_{i<k}$$
$$[\![?(f_i : \tau_i \; ; \; p_i)_{i<k}]\!] = \text{and } \mathit{name}(p) = \{\{\mathbf{h} f_i \text{: principals} \rightarrow \tau_i \rightarrow \mathit{name}(p_i);\}_{i<k}\}$$

This generates a collection of potentially mutually recursive types, which explains the default use of the and keyword. A pretty-printing phase then completes the interface generation. As shown in section 3, the result types have the following shape:

```
[...] and msg11 = {
        hBadFormat : (principals → unit → msg10) ; hOk : (principals → unit → msg12)}
    and msg12 =
    | Submit of (string ∗ msg13) | Withdraw of (unit ∗ result_author)
[...]
```

Wired types and messages generation. The low-level handling of messages in the generated protocols is done by a series of specialized types and functions. These functions have also the task of maintaining a local store containing the necessary cryptographic material for the session. Concretely, `s2ml` generates a family of **sendWired***label* functions (one generated function for each message tagged with *label* of the session) that perform the following operations:

1. build the session id (a digest of the session declaration, principals, and a nonce);
2. build the header (the session id plus the sender and receiver's identities);
3. marshall the payload;
4. create a new signature of the label and logical time;
5. update the local signature store and logical clock;
6. build the message from the header, the label, the payload and the transmitted signatures (whose list is known from the previously computed visibility);
7. send the message on the network

Symmetrically, the receiving sequence of actions done by the family of **receiveWired**n functions (one function for each node n in the graph) is the following:

1. receive the message from the network;
2. unmarshall and decompose into parts (header, label, payload, signatures);
3. check the session id;
4. match the message label against possible incoming messages;
5. check the signatures' correctness (using visibility) and logical time-stamps;
6. update the local signature store and logical clock;
7. check the message against the cache (if it is the first message of a run of the session)

Any check failure will either silently restart the function (to continue listening) or throw an exception. Since initial messages require special treatment (e.g., cache checking), `s2ml` creates specific versions of the low-level functions (named with the **init** suffix).

The types of the **sendWired***label* and **receiveWired**n are of the form:

```
val sendWiredlabel : wiredn → state
val receiveWiredn : state → unit → wiredn
```

where the state corresponds to the local cryptographic store, and the **wired**n types are the sum types corresponding to messages that can be received in the state n of the role's process. The internals of the proxy, in charge of enforcing the session flow and user interaction, critically relies on these types.

Proxy functions The last part of the generated protocol implementation consists on the proxy functions that the user can call from the interface. Their purpose is to follow the flow of sent and received messages as specified by the session and to call back a user-defined continuation at the correct moment.

Concretely, these functions have to be able to handle the users' choices of messages to send and call the appropriate low-level **sendWired***label* function. Then they have to listen to incoming messages using the **receiveWired***n* functions and, when a message is received, to call back the appropriate field of the user-specified record of continuations.

We illustrate these proxy functions by the **author** function from the CMS example:

```
let author (prin: principal) (user_input : msg9) =
...
and author_msg10 (st:state) : msg11 → result_author = function
   | Upload(x, next) → let newSt = sendWiredUpload host dest (WiredUpload(st, x)) in
      author_msg11 newSt next
and author_msg9_init : msg9 → result_author =
 function handlers →
  let (newSt, r) = receiveWired0_init host prin () in
      match r with
      | WiredCfp (newSt, x) → let next = handlers.hCfp newSt.prins x in
      author_msg10 newSt next
 in
 Printf.printf "Executing role author with principal %s...\n" prin;
 author_msg9_init user_input
```

Initially it calls the function **author_msg9_init** which uses **receiveWired0_init** to receive a first message. It is checked to be a **Cfp** message, and if so, the payload **x** is applied to the user code continuation (**handlers**.**hCfp**), and then the function **author_msg10** is invoked, which continues the session by sending a **Upload** message.

5.1 Concrete implementation and benchmarks

Our concrete implementation links the generated code against concrete cryptographic implementations (as opposed to a symbolic model, used to formally prove security, which uses algebraic datatypes). We provide two variants of concrete libraries: one using Ocaml and wrappers for OpenSSL, and another using F#/Microsoft .NET cryptography. (Unfortunately the two implementations do not yet interoperate, due to incompatibilities among certificates.) The data and cryptographic functions we use are as follows. For cryptography, we use SHA1 for hashing, RSASHA1 for signing, and the standard pseudorandom function for nonce generation. Signing uses certificates in '.key' format for OpenSSL and '.cer' for Microsoft .NET. As for data, we use Base64 for encoding the messages in a communicable format. We use UDP-based communication (although in the future we plan to move to TCP-based communications).

Benchmarks. We executed the CMS example using the Ocaml/OpenSSL concrete implementation in a setting in which every loop is iterated 500 times. This table reports the benchmarks for a Pentium D 3.0 GHz running linux-2.6.17-x86_64:

	No cryptography	Signing but not Verifying	Signing and Verifying
first loop	0.231s	2.79s	2.95s
second loop	0.468s	5.62s	6.11s
third loop	0.243s	2.81s	2.98s
total	0.942s	11.22s	12.04s

These results show that most execution time is devoted to cryptography, as expected: the generated code `s2ml` consists of optimally compact, specialized message handlers.

6 Conclusions

We present a simple language for specifying sessions between roles, and we detail its usage as a secure communication abstraction on top of ML. Our compiler `s2ml` 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.

Whilst in previous work we focus on establishing (theoretical) security guarantees for the generated code of `s2ml`, here we concentrate on describing the inner workings of the compiler, and explore its applicability to the concrete examples of an RPC exchange and a rather large conference management system. This latter case study is treated smoothly by `s2ml`, providing confidence for its usability as a concrete tool for structuring and securing distributed programming.

Future Work. We are exploring variants of our design to increase the expressiveness of sessions: session-scoped data bindings that ensure 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 subject to agreement among the current principals. We are also interested on providing support for communicating richer payload types, by studying the extension of `s2ml` with general and secure marshalling.

Acknowledgments. This work benefited from discussions with K. Bhargavan, C. Fournet, J. Leifer, and J-J. Lévy. Special thanks to James Leifer for his insightful comments. An earlier version of `s2ml` was developed while visiting at Microsoft Research Cambridge, UK.

References

1. Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop, (CSFW)*, pages 139–152, July 2006.
2. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Chris Hankin, editor, *Programming Languages and Systems, 16th European Symposium on Programming (ESOP)*, LNCS. Springer, 2007.
3. Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 197–213. Springer, 2006.
4. Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 45–57, January 2002.

5. R. Corin, P.M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, Venice, Italy, July 2007. IEEE. To appear.
6. Ricardo Corin, Pierre-Malo Dénielou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. Secure sessions project. At `http://www.msr-inria.inria.fr/projects/sec/sessions/`, 2007.
7. Mariangiola Dezani-Ciancaglini, Dimitrios Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *20th European Conference for Object-Oriented Languages*, July 2006.
8. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
9. Dot. At `http://www.graphviz.org/`.
10. Manuel Fahndrich, Mark Aiken, Chris Hawblitzel, Galen Hunt Orion Hodson, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EUROSYS*, 2006.
11. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
12. Objective Caml. At `http://caml.inria.fr`.
13. D. Syme. *F#*, 2005. At `http://research.microsoft.com/fsharp/`.
14. Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.
15. Nobuko Yoshida and Vasco 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.

A Output for Rpc

```
$ ./client.exe
[Impl] Executing role client with principal
alice...
[Impl] Preparing message: Query(Number?)
[Prins] sent:
MDg0MDAyTkowNjEwMDEwMDAxMTAyMKMdkFskmfB0Eb9KUk1/E2
Fzn50OMDExMDA1YWxpY2Vib2KVmxw7s9N6gWZ+SEs1XyrZMDA1
UXVlcnlOdW1iZXI/MDAxMDEyOHp+3gwAG+ae388U5fV5L42sLU
7t6hMOUT9JlPyHnVMPCxavHAYq5GhnOwjFnY5RWGywCtw1E+F1
miECRTcFLwNOLB3tdEZUaOxqOd7jwq4deiC+WWwZoRoEyi0FE1
HgO3H2FJIrqD19Twy1UDBQqmG+Nnoq0Vgo3CwH+KfbRbhaMDAx
MDAwMA==

[Prins] received:
MDgyMDAyTkowNjEwMDExMDAxMDAyMKMdkFskmfB0Eb9KUk1/E2
Fzn50OMDExMDA1YWxpY2Vib2KVmxw7s9N6gWZ+SEs1XyrZMDA4
UmVzcG9uc2U0MjAwMTExMjiP/uJ+eHiWg6gbgY98CjinyGoSmK
8/p1jfxT7d2lW7o46a/uz3G6iJx+xoSArCRqpMIr3frIZHr6PF
yf0ATnS2IGkcGtVlNQ39tpZh/PRjHGbdltpD21KhlxNN5m8jHe
6Q/JPg1oaTQ9UZegYS6SeEWl45g4WXxwj74J4KJMf91jAwMTAw
MDA=
[Impl] Accepted correct message: Response(42)
[Impl] Updating signatures ...
[Impl] Signatures updated!
Answer is 42
```

B CMS Code for author, pc and confman

B.1 Interface

```
type result_pc = string

type msg0 =
    Cfp of (string ∗ msg1)
and msg1 = {
    hPaper : (principals → string → msg2) ;
    hRetract : (principals → unit → result_pc)}
and msg2 =
    Close of (unit ∗ msg3)
  | ReqRevision of (string ∗ msg1)
and msg3 = {
    hDone : (principals → unit → msg4)}
and msg4 =
    Accept of (string ∗ result_pc)
  | Reject of (string ∗ result_pc)
  | Shepherd of (string ∗ msg7)
and msg7 = {
    hRebuttal : (principals → string → msg4)}

val pc : principals → msg0 → result_pc

type result_confman = string

type msg18 = {
    hUpload : (principals → string → msg19)}
and msg19 =
    Ok of (unit ∗ msg20)
  | BadFormat of (unit ∗ msg18)
and msg20 = {
    hSubmit : (principals → string → msg21) ;
    hWithdraw : (principals → unit → msg26)}
and msg21 =
    Paper of (string ∗ msg22)
and msg22 = {
    hClose : (principals → unit → msg23) ;
    hReqRevision : (principals → string → msg25)}
and msg23 =
    Done of (unit ∗ result_confman)
and msg25 =
    Revise of (string ∗ msg20)
```

```
and msg26 =
   Retract of (unit ∗ result_confman)

val confman : principal → msg18 → result_confman
```

B.2 User code

```
open Conf
open Printf

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

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

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

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

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

```
open Conf
open Printf

let (prins:principals) =
  { pc = "charlie";
    author = "alice";
    confman = "bob" }

let cfp = "Call for papers"
let rec handler_discuss = {
  hRebuttal = (fun _ _ →
    Accept("Ok then ...", {
        hFinalVersion = (fun _ s → "We accepted
        the following paper: "^s)
      }))
  }

let rec handler_paper = {
    hPaper = (fun _ s → if s.[0] = 'S'
      then ReqRevision("Make it better!", handler_paper)
      else Close((),{
          hDone = (fun _ _ → Shepherd("Do you really want
          to be in?", handler_discuss))
        }));
    hRetract = (fun _ _ → "Retracted") }


let _ =
  let result = pc prins (Cfp(cfp, handler_paper)) in
  printf "PC: session complete: %s\n\n" result

open Conf
open Printf

let rec handler_decision = {
    hClose = (fun _ _ → Done((), "No more revisions"));
    hReqRevision = (fun _ r → Revise(r, handler_submission))
}

and handler_submission = {
    hSubmit = (fun _ submit → Paper(submit, handler_decision));
    hWithdraw = (fun _ _ → Retract((), "Retracted"))
  }

let rec handler_paper prins draft =
  if String.length draft > 12
  then BadFormat("Make it shorter!",{hUpload = handler_paper})
  else Ok((),handler_submission)

let _ =
  let result = confman "bob" {hUpload = handler_paper} in
  printf "ConfMan: session complete: %s\n\n" result
```