Cryptographic Protocol Synthesis and Verification for Multiparty Sessions

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

[1] MSR-INRIA Joint Centre, Orsay, France, [2] Microsoft Research, Cambridge, UK

Abstract

We present a compiler for generating custom cryptographic protocols from high-level multiparty sessions.

Sessions specify pre-arranged patterns of message exchanges between distributed participants and their data accesses to a shared store. We define integrity and confidentiality properties of sessions, in a setting where the network and arbitrary compromised parties may be controlled by an adversary. Our compiler enforces these security properties by guarding the sending and receiving of session messages by efficient cryptographic operations and checks.

Given a session, our compiler generates an ML module and an interface that exposes send and receive functions that can be called by application code for each party. We prove that this generated code is secure by relying on a recent refinement type system for ML. Functions in the module interface are given dependent types that express invariants of the session state. We typecheck the program against this interface, and complete the proof by a brief, hand-crafted argument.

We illustrate and evaluate our implementation on a series of typical protocols, inspired by web services. In comparison with prior work, our source language is more expressive, our implementation more efficient, and our proof technique novel. Most of the proof is performed by mechanized type checking of the generated code, and does not rely on the correctness of our compiler. We obtain the strongest session security guarantees to date in a model that accounts for the actual details of protocol code.

1. Security by compilation

Taking advantage of modern programming tools and generic wire formats, one can sometimes design, develop, and deploy complex distributed protocols in a matter of hours—relying on automated proxy generators, for example, to rapidly expose existing applications as networked services. The situation is less favorable when attempting to ensure application security subject to realistic assumptions on the network and remote parties. The problem is that most widely-available protocols for cryptographic communications (say TLS or IPSEC) operate at a lower level; they provide authenticity and confidentiality guarantees only for messages exchanged between two endpoints (URLs or IP addresses), but they leave the interpretation of these messages and endpoints to the application programmer. Hence, any guarantee that involves more than two parties (say, a multi-tiered web application with a client, a gateway, and two servers) must be carefully established by linking lower-level guarantees on related messages, or by layering ad hoc cryptographic mechanisms on top of communications, for instance by embedding certificates in message payloads.

When considering protocols between participants in an open, networked environment, each participant may belong to a different domain, with its own configuration and security policy; although all participants are willing to run the protocol, they may not trust one another. Although it is straightforward to protect the participants of the protocol from network intruders (using for instance a VPN), it is more delicate to design and verify cryptographic infrastructure to protect these participants from one another.

Rather than hand-crafted protocol design, we advocate the use of compilers and automated verification tools for systematically generating secure, efficient cryptographic protocols from high-level descriptions. We outline first our approach as regards design, implementation, and security verification.

Multiparty sessions We design a language for specifying structured communications protocols between distributed parties, also known as *sessions*, or *workflows*. This language enables simple, abstract reasoning on authentication and secrecy properties of sessions. It features a clear notion of control flow, expressed as asynchronous messages, with a shared, global store that may be updated and read during communication, as well as dynamic selection of additional parties to join the protocol. The global store is subject to fine grained read/write access control and may be used to selectively share and hide data and to commit to values which are are initially blinded and only revealed later during protocol execution. As an example, the correspondence properties traditionally established for cryptographic protocols can be read off session specifications.

A session implementation exposes to the application programmer certain application level choices, such as which session to join, which session messages to send (when the session allows a choice), and which local authorization policy to enforce.

A compiler from sessions to cryptographic protocols for ML Two central design goals guide our work on session implementation. First, all the cryptography required to protect compromised participants is completely hidden from the application programmer, who may reason about the behaviour of a distributed system as if it followed precisely the high level specification. Second, all low-level network activity is in a one-to-one relationship with high-level communication, thus no additional messages are introduced.

We implement a compiler from the session language to custom cryptographic protocols, coded as ML modules, both for F# (Syme et al. 2007) with .NET cryptographic libraries, and for Ocaml with OpenSSL libraries, which can be linked to application code for each party of the protocol. Our compiler combines a variety of cryptographic techniques and primitives to produce compact message formats and fast processing. We illustrate and evaluate our implementation on a series of protocols, inspired by web services.

We shift most of the complexity of our implementation to the compiler, which generates efficient, custom protocols, with the least amount of dynamic processing. Any deviation from the compiled message format is considered hostile and the message silently discarded.

Security verification Cryptographic protocols are notoriously difficult to design and implement correctly; in particular, we need solid correctness properties for the cryptographic code generated by our protocol compiler.

Various work addresses this problem. We build on a recent approach proposed by Bhargavan et al., who aim at verifying executable protocol code, rather than abstract protocol models to narrow the gap between what is verified and what is deployed. Specifically, we use the extend typechecker of Bengtson et al. (2008), which is based on the Z3 SMT solver (de Moura and Bjørner 2008). This is a good match for our present purposes: for each session specification, our compiler generates type annotations (from a predicate logic), which are then mechanically checked against the actual executable code. By doing this, we overcome a significant limitation of protocol code verification techniques: our verification by typechecking is modular, so each function can be checked separately and verification time grows linearly with the number of functions.

We define *compromised* principals as those whose keys are known to the adversary; they include malicious principals as well as principals whose keys have been inadvertently leaked. We say that all other principals are *compliant.* Our goal is to protect compliant principals from an adversary who controls all compromised principals and the network. To this end, we verify the generated protocols, showing strong security for all runs, even when some of the parties involved are compromised. Our proof combines invariants established through typechecking with a general argument on the structure of the protocol (but independent of the code). In combination, we obtain strong security guarantees in a model that accounts for the actual details of our code, without the need to trust our protocol compiler.

An alternative approach would be to verify (or even certify) our compiler. This task appears much more complex; it is beyond the capability of automatic tools at present and would require long, delicate, handwritten proof, such as those we did in prior work (Corin et al. 2007) for a much simpler protocol description language. This approach is also brittle when experimenting with language design, and would not provide direct guarantees at the level of the generated code.

We obtain additional functional properties by typing: any well-typed application code (for ordinary ML typing) linked to our protocol implementation complies with the session specification: at any point in the session, it may send only one of the messages indicated in the global sessions, and it must provide a message handler for every message that may be subsequently received.

On the other hand, we do not consider many other properties of interest, such as liveness (any participant may block our sessions), resistance to traffic analysis (only our payloads are kept secret), or denial-of-service attacks.

Contributions In summary, our main contributions are:

1. A high-level language for specifying multiparty sessions, with support for payload binding, integrity, secrecy, and dynamic principal selection; this language enables simple, abstract reasoning on global control and data flows.
2. A family of custom cryptographic protocols that support our design. We rely on standard symmetric-key cryptographic primitives, as well as standard primitives that provide asynchronous, unprotected communications.
3. A prototype compiler that generates ML interfaces and implementations for our protocols, as well as proof annotations.
4. Experimental results for a series of multiparty sessions of increasing complexity, showing that our approach yields efficient distributed implementations.
5. Security theorems stating that, from the viewpoint of compliant participants, all sessions always run according to their global specification, both as regards integrity and confidentiality, despite active adversaries in control of both the network and compromised participants.
6. New, mostly-automated security proof techniques: to our knowledge, we obtain the first automated generate-and-verify implementation for multiparty cryptographic protocols, and the largest verified protocol-implementations to date.

Related work This work benefits from our experience with a first prototype: Corin et al. (2007) explore the secure cryptographic implementation of session abstractions for a simpler language; Corin and Deniélou (2007) detail its first design and implementation. The main differences are a much-improved expressiveness (with support for value binding, and dynamic selection of principals), a more sophisticated implementation (with more efficient cryptographic mechanisms), a simpler and more realistic model for the adversary, and a new formalization with support for automated proofs.

Session types Honda et al. (2008); Bonelli and Compagnoni (2007); Vasconcelos et al. (2006); Dezani-Ciancaglini et al. (2006, 2005); Gay and Hole (1999); Honda et al. (1998) consider types for concurrent and distributed sessions; however they do not consider implementations or security. More recently, Hu et al. (2008) integrates session types in Java; McCarthy and Krishnamurthi (2008) specifies abstract security protocol narrations in a global way, and then shows functional (but not security) aspects of their projection to local roles (like in Honda et al. 2008). Inference of sessions types from existing Javascript applications is done in Guha and Krishnamurthi (2008).

Verified cryptographic implementations Further related work tackle the secure implementation problem for other programming models. Malkhi et al. (2004), for instance, develop implementations for cryptographically-secured multiparty computations, based on replicated, blinded computation. (Zheng et al. 2003), for instance, develop compilers that can automatically split sequential programs between hosts running at different levels of security, while maintaining information-flow properties.

Contents Section 2 describes and illustrates our design for multiparty sessions. Section 3 presents our programming model for using sessions from ML. Section 4 states our main session-integrity theorem. Section 5 describes the refinement of sessions into local control flow states for roles. Section 6 overviews the cryptographic protection and message formats. Section 7 describes code generation and runtime libraries. Section 8 explains our hybrid security proof. Section 9 provides experimental results obtained with our prototype.

Additional materials, including code for all examples and libraries, are available online at `http://msr-inria.inria.fr/projects/sec/sessions`.

2. Multiparty sessions

We first introduce sessions informally by illustrating their graphical representation and describing their features and design choices. We then give their formal, algebraic presentation and define some sanity and implementability conditions.

2.1 Session design: overview and motivation

Figure 1 represents our three running examples, loosely inspired by Web Services; they involve a client, a web service, and a proxy.

Graphs, roles, and labels The first session, named Ws, is depicted as a directed graph with a distinguished (circled) *initial* node. Each node is decorated by a *role*; here we have two roles,

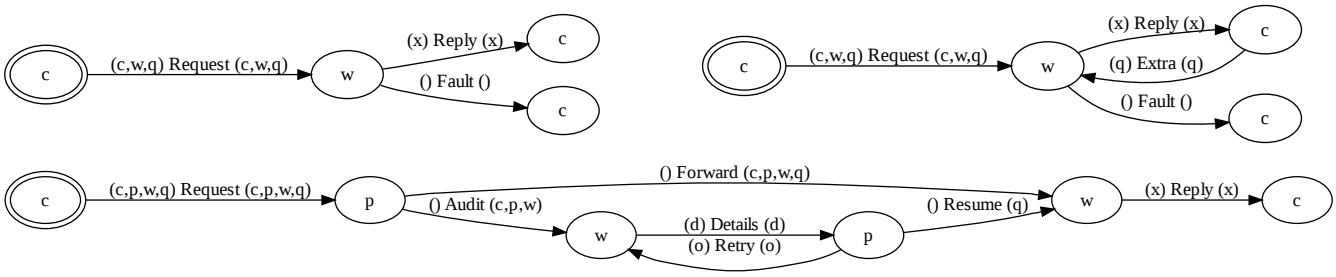

Figure 1. Sample sessions: (a) top-left: a single query (Ws); (b) top-right: an iterated query (Wsn); (c) bottom: a three-party session (Proxy).

c for "client" and w for "web service". Each edge is identified by a unique *label*, in this case, Request, Reply, or Fault. (The other annotations on the edges may be ignored for the moment.) The *source* and *target* roles of an edge (or label) are the roles decorating, respectively, the edge's source and target nodes. Thus, Reply has source role w and target role c. Each role represents a participant of the session, with its own local implementation and application code for sending and receiving messages, subject to the rules of session execution, which we explain next. The precise way in which application code links to the session infrastructure is described in Section 3.

Session execution Sessions specify patterns of allowed communication between the roles: in this case, the client starts a session execution by sending a Request to the web service, which in turn may send either a Reply or a Fault back to the client. Each execution of a session consists of a walk of a single token through the session graph. At each step, the role decorating the token's current node chooses one of the outgoing edges, and the token advances to the target node of the chosen edge. Execution ends when the token reaches a node with no outgoing edges.

Loops and branches The session Wsn in Figure 1(b) extends the graph with a cycle. On receiving a Reply, the client can choose to either terminate the session or send a new message Extra back to the web service; the client and service may then repeat this Reply-Extra loop any number of times before the client terminates.

The session Proxy in Figure 1(c) allows multiple alternate message flows between three parties. It introduces a third role, p, for "proxy", that intercedes between the client and the web service. The client starts by sending a Request to the proxy, which may choose to either transmit a Forward message or an Audit message to the web service. In the later case, the communication may loop between the web service and the proxy via Details and Retry until the proxy is satisfied and sends Resume to web service, which, finally, gives a Reply back to the client.

Binding and receiving values Each session has a finite set of typed mutable variables (though all types are omitted from graphs for brevity). The graph imposes an access control discipline on the writing and reading of variables via the decoration of each edge by two vectors of variables. The vector just before the label constitutes the *written* variables; the vector just after constitutes the *read* variables. At the start of session execution, all variables are uninitialised. At each communication, the written variables decorating the corresponding edge are assigned values by the source role; the values of the edge's read variables are then passed to the target role.

In Figure 1(c), the client writes an initial value into variable q, representing some query, as it sends the Request message since q appears in the written variables of Request. This variable also appears in the read variables of Request, so the proxy may in turn read q and then take a decision whether to carry on with a Forward or Audit. In the former case, the proxy may not modify q since the variable does not appear in the written variables of Forward, so the web service gets the same value of q as the proxy did.

Not all variables are read by all roles. During each iteration of the graph's loop, the web service may modify d as it sends Details, and likewise the proxy may modify o. Both these variables are hidden from the client role, which has no incoming edges where d or o are read.

Intuitively, the graph represents a global viewpoint, so the variables locally written and read on an edge need not coincide, and all readers are guaranteed to get the same values unless the variable is explicitly rewritten.

Assigning principals to roles Roles themselves are treated as a special class of variables and are assigned during session execution to *principals*, representing some participant equipped with a network address and a cryptographic identity.

For example, in Figure 1(c), the client role running on behalf of a particular principal initially assigns principals to itself, as well as to the other roles p and w, and writes these principals in the Request message. In general, the first message need not write all the session's role variables, thus allowing dynamic choice of subsequent principals during session execution. However, role variable must be instantiated before the role is used as a target, and role variables may not be rewritten.

Global session graphs (Definition) We model the global, static view of sessions as directed graphs where nodes are session states tagged with their role, and edges are labelled with message descriptors decorated with written and read variables. We write $\widetilde{v}$ to denote sequences $(v_0 \dots v_k)$. Formally, a session graph $G =$

$$(\mathcal{R}, \mathcal{V}, \mathcal{X}, \mathcal{L}, m_0 \in \mathcal{V}, \mathcal{E} \subseteq \mathcal{V} \times \mathcal{X}^* \times \mathcal{L} \times \mathcal{X}^* \times \mathcal{V}, R : \mathcal{V} \to \mathcal{R})$$

consists of a finite set of roles $r, r', r_i \in \mathcal{R}$; a finite set of nodes $m, m', m_i \in \mathcal{V}$; a set of variables $\mathcal{X} = \mathcal{X}_\mathsf{d} \uplus \mathcal{R}$ (the disjoint union of data variables $\mathcal{X}_\mathsf{d}$ and roles $\mathcal{R}$); a set of labels $f, g, l \in \mathcal{L}$; an initial node m_0; a set of labelled edges $(m, \widetilde{x}, f, \widetilde{y}, m') \in \mathcal{E}$, for which each variable occurs at most once in the vector $\widetilde{x}$ and likewise for $\widetilde{y}$ (though the two vectors may have variables in common); and a function R from nodes to roles.

For an edge $(m, \widetilde{x}, f, \widetilde{y}, m')$, we say that $\mathsf{write}(f) = \widetilde{x}$ and $\mathsf{read}(f) = \widetilde{y}$, the *written* and *read* variables of f (respectively.) Also we use $\mathsf{src}(f) = R(m)$ and $\mathsf{tgt}(f) = R(m')$ for the *source* and *target* roles of f.

A sequence of labels $\widetilde{f}$ for which the target node of each label equals to source node of the next one is called a *path*. We write $\widetilde{f}f$ or $\widetilde{f}\widetilde{g}$ to denote the path $\widetilde{f}$ concatenated with a final f or another path $\widetilde{g}$, respectively. The empty path is written ε. An *initial path* is a path for which the source node of the first label is m_0, the initial node of the graph. An *extended path* is a sequence of alternating labels (not necessarily adjacent) and lists of variables, of the form $(\widetilde{x}_0)f_0 \dots (\widetilde{x}_k)f_k$. We let $\hat{f}$ range over extended paths, and let ε be the empty path.

A role r is *active* on a path when r is the role of any source node of an edge of the path.

Well-formedness properties of sessions We define well-formedness and implementability properties for session graphs to be considered valid. Most of these properties are implementability conditions motivated by our compiler and verification; they ensure that we do not need to send extra messages to protect the security of the session.

1. Edges have distinct source and target roles: if $(m, \widetilde{x}, f, \widetilde{y}, m') \in \mathcal{E}$, then $R(m) \neq R(m')$.
2. Edges have distinct labels: if $(m_1, \widetilde{x_1}, f, \widetilde{y_1}, m_1') \in \mathcal{E}$ and $(m_2, \widetilde{x_2}, f, \widetilde{y_2}, m_2') \in \mathcal{E}$, then $m_1 = m_2$, $m_1' = m_2'$, $\widetilde{x_1} = \widetilde{x_2}$, and $\widetilde{y_1} = \widetilde{y_2}$.
3. Every node is reachable: if $m \in \mathcal{V}$ then either $m = m_0$, the initial node, or there exists an initial path $\widetilde{f}f$ such that $\mathsf{tgtnode}(f) = m$.
4. For any initial paths $\widetilde{f}\widetilde{f_1}$ and $\widetilde{f}\widetilde{f_2}$ ending with distinct roles r_1 and r_2, respectively, there exists a role r active on either $\widetilde{f_1}$ or $\widetilde{f_2}$ such that $r_1 = r$ or $r_2 = r$.
5. On any initial path, every role variable is written at most once.
6. For any initial path $\widetilde{f}f$ ending in role r, we have $r \in \mathsf{knows}(r, \widetilde{f}f)$.
7. For any initial path $\widetilde{f}$ ending in role r, and any role r' active on $\widetilde{f}$, we have $r' \in \mathsf{knows}(r, \widetilde{f})$.
8. For any initial path $\widetilde{f}$ ending in role r for the first time (i.e. r not active on $\widetilde{f}$), every role r' active on $\widetilde{f}$ is such that $r \in \mathsf{knows}(r', \widetilde{f})$.
9. For any initial path $\widetilde{f}f$, if x is read on f and f has source role r then $x \in \mathsf{knows}(r, \widetilde{f}f)$.
10. For any initial path $\widetilde{f}f$ ending in role r, if x is read on f and $x \in \mathsf{knows}(r, f)$ then x is written on f.

We motivate each of these properties in the following informal discussion.

1, 2, 3: These are "sanity" properties. The first ensures that there are no communications internal to a role in the session graph, thus enforcing the intuition that any edge in the graph represents a potentially attackable communication (which the cryptographic implementation needs to defend against). The second property allows us to unambiguously index edges by their labels, which is notationally convenient. The third states that there are no junk nodes unreachable from initial paths.

4: This property ensures that no compromised principal at a forking point in the graph can cause several branches to all execute in order to break the discipline that a single session run consists of a unique execution path through the graph.

5: Throughout the presentation of these properties we use roles as placeholders of fixed but unknown principals to be instantiated during session execution. Without this property, we would be forced to distinguish between the different occurrences of a role depending on the possible rebinding of the role in between, which is unnecessarily complicated.

6: This property says that every role knows the value of its corresponding role variable, which prevents an adversary from diverting a message intended for one principal to another in order to trick the latter to join the session.

7: This ensures that every principal knows all the previous principals who have joined the session so far, thus preventing an adversary from tricking a recipient into believing that some different set of principals has joined.

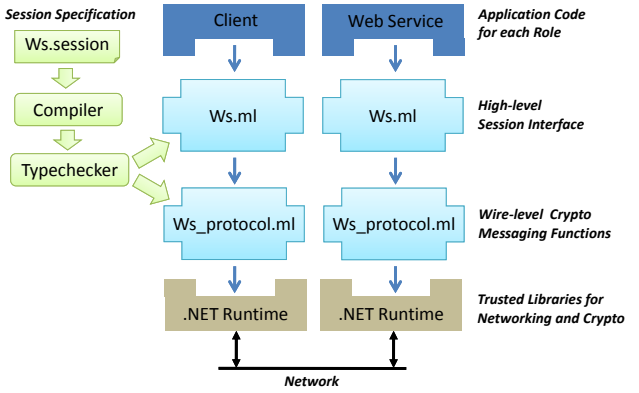

Figure 2. Compiling Session Programs

8: This property prevents an adversary from forking a session execution by concurrently assigning two different principals to the same role.

9: This property prevents graphs in which an adversary that cannot read a particular variable colludes with one who can in order to gain premature access to the variable's value.

10: This property says that a role never reads a variable that it already knows, thus rendering the graphs "more canonical".

3. Programming with sessions

Figure 2 illustrates our programming framework for the example Ws of Figure 1(a).

All our programs are in ML, and may be compiled either by the F# compiler using .NET libraries for networking and cryptography, or by the Ocaml compiler using OpenSSL cryptography libraries. Appendix A defines a formal subset of ML used in this paper.

Each compliant principal runs application code that may enact roles in several sessions. In our example, there are two applications, a client and a web service, participating in the c and w roles, respectively, of the Ws session. The application code for compliant principals may contain arbitrary computations and non-session communications, but for all session communications, it relies on a high-level interface provided by a session module (Ws.ml). (The figure depicts the case where both principals are compliant and use our implementation, but our security theorems do not make this assumption; see Section 4.) The session module ensures that both local and remote principals comply with the session. For the local application, the interface guarantees that messages may only be sent and received in the correct order. To check compliance of remote principals, the session module relies on a protocol module (Ws_protocol.ml) that inserts cryptographic materials on all sent messages and checks them on all received messages.

Our compiler takes a syntactic session description (Ws.session) and generates both the session and protocol modules; it then feeds these modules into the extended typechecker for security verification. To write code that complies with a session, a programmer only needs to read the interface of the session module. In this section, we describe a syntax of sessions and the structure of the generated session module and interface. In later sections, we describe the generated protocol module, and prove that it enforces our session security goals.

Session syntax We declare sessions using a process-like syntax for each role. This is a *local* representation, unlike the global session graphs in Section 2, but we can convert between the two representations. Other works (Honda et al. 2008) explore conditions

under which such interconversions are possible. Since this is not the main focus of our work, we do not detail them, except to note that all the global session graphs in this paper were automatically generated from local syntactic descriptions.

$\tau ::=$ int | string	Payload types
$p ::=$	Role processes
send $(\{+f_i(\widetilde{x_i}); p_i\}_{i<k})$	send
recv $[\{\|f_i(\widetilde{x_i}) \rightarrow p_i\}_{i<k}]$	receive
$\chi : p$	named subprocess
χ	continue with χ
0	end
$\Sigma ::=$	Sessions
(var $x_j : \tau_j)_{j<m}$ (role $r_i{:}\tau_i = p_i)_{i<n}$	initial role processes

We illustrate our syntax through an example. The following session specifies the graph for Ws given in Figure 1:

```
session Ws =
  var q : string
  var x : int
  role c : unit =
    send (Request (c,w,q); recv [ Reply (x) | Fault ])
  role w : string =
    recv [Request (c,w,q) → send ( Reply (x) + Fault )]
```

The session Ws declares two variables and two roles. Each variable (q, x) represents a value communicated in the session and is given a type. Each role (c, w) is defined in terms of a return type and a *role process* that performs a sequence of send and receive actions. At every send, the role process may chooses between several messages, and at every recv, it expects to receive one of several messages. Here, c sends Request and binds variables c, w, and q, before receiving either a Reply (and reading x) or a Fault.

For a session to be well-formed, it must additionally satisfy the conditions described in Section 2; for example, sends and receives within a role process must alternate, only one role must send a message with a particular label, and so on. These checks are encoded as part of the early stages of our compiler.

Given a syntactic session, our compiler generates a session module that defines the functions made available to the application. Figure 3 shows an excerpt of the files generated from our Ws example. The file Ws.mli declares the session module interface, WebServer.ml is an example application using this interface, and Ws.ml implements the interface. The figure illustrates the types and functions corresponding to the w role. We describe these three files in order.

Generated session interface For each variable (e.g. c, x) the compiler generates a type constructor (e.g. C, X) that is used by the application to label values assigned to the variable. For each role process (w), it generates a function (w) that executes message send and receive functions in the prescribed order. To use this function, the application must provide a record containing messages for every send and message handlers for every receive, in a continuation-passing style.

In our example, the w function takes two arguments: the name (host) of the principal who wishes to run w, and a record (user_input) of type msg4. The record contains a single message handler, hRequest, that is called whenever a new Request message containing the variables c, w, and q is received. The message handler returns a value of type msg5 that contains the next message to be sent; here it returns either a Reply containing the variable x, or it returns a Fault. Once this message has been sent, no further messages are expected and so the function w terminates its session and returns. In more lengthy sessions, the value returned by the handler would also contain new message handlers for the next messages that may be received. The types msg4 and msg5 are defined here as a set of nested algebraic types that could have been inlined, but, in general, when role processes have loops, these type definitions are mutually recursive.

Application code Any code that only uses the session interface to send session messages is a valid application for a compliant principal. In our framework, application code may be written in OCaml or any .NET language, including F# and C#, but our formal results hold for code written in our subset of ML.

The example shows a usage of the w function; the application invokes the function w on behalf of a principal named `"Bob"`, and provides a message handler for the Request message; if the variable q has been assigned the string `"I want an answer."`, then it responds with a Reply message with variable x assigned the integer 42; otherwise, it returns a Fault.

Ordinary ML typechecking of application code against a session interface provides a form of *local integrity*: it ensures that the application must comply with the role process. For instance, our example code would not typecheck if we forgot to provide a Fault handler, or if we tried to receive a Request instead of a Reply. By adding cryptography, we aim to provide global session integrity, as formalized in the next session, that will hold even if some applications do not use our generated session module.

Generated session module The code implementing role functions performs three tasks: it ensures that sends and receives are performed in the prescribed order by keeping track of the session state, it interfaces with the wire-level cryptographic protocol module to send and receive messages, and it logs security events indicating session progress. The translation of role processes into role functions is straightforward; for brevity, we elide the full translation and instead illustrate the generated code by example.

The session state consists of an internal control flow state ρ (formally defined in Section 5) and a data store st. The internal control flow state ρ represents a path in the session graph that the current session instance has followed so far. Since the number and lengths of paths in sessions with loops is unbounded, ρ is an abstraction of the full path. Control states can be statically precomputed from a session graph (see Section 5).

The store st records the parameters specific to the current session instance, such as the session identifier (st.header.sid), the current timestamp (st.header.ts) and the current values assigned to each session variable (st.vars.c, st.vars.x). It is maintained by the protocol module Ws_protocol.

The code for our example session consists of a function w_ρ for each ρ that ends in a message sent or received by the role w. After sending or receiving one message, each w_{ρ_i} either terminates its participation in the session or calls another function w_{ρ_j}, where ρ_j represents the next message that may be received or sent at w. The function w initializes the store and then calls the function w_{ρ_0} representing the first message that w receives in the session; w_{ρ_0} calls w_{ρ_1} to send the next message and return.

Each message send and receive is a function call to the protocol module. (Section 7 describes the generation of this module.) For each internal control flow state ρ and message f(x), the protocol module defines a function sendWired_f_ρ that takes the current store and a value x and constructs and sends a cryptographically protected message, returning an updated store. For each internal control flow state ρ, it also defines a function receiveWired_ρ that takes the current store and returns a cryptographically verified received message and an updated store. Our example uses two send functions sendWired_Fault_ρ_1 and sendWired_Reply_ρ_1, and one receive function receiveWired_ρ_0.

Before sending and after receiving a message, the function w logs security events asserting that one step in the session has been completed. For example, before sending the reply message, it calls the function assume to log an event, Send_Reply, stating that it

Ws.mli

```
type var_c = C of principal
type var_w = W of principal
type var_q = Q of string
type var_x = X of int
type result_w = string
type msg4 = {
    hRequest : (var_c * var_w * var_q → msg5)}
and msg5 =
    Reply of (var_x * result_w) | Fault of (result_w)
val w : principal → msg4 → result_w
```

WebService.ml

```
Ws.w "Bob"
{ hRequest = function (_, _, Q q) →
    if q = "I want an answer."
    then Reply (X 42, "Reply sent\n")
    else Fault ("Request failed\n") }
```

Ws.ml

```
let rec w (host : principal) (user_input : msg4) =
  let empty_store = empty_store_w host false in
    w_ρ0 empty_store user_input in
and rec w_ρ0 : Ws_protocol.store → msg4 → result_w = fun st handlers →
  match Ws_protocol.receiveWired_ρ0 st with
  | Wired_Request_ρ0 ((c, w, q), newSt) →
      assume (Recv_Request (w, newSt.header.sid, newSt.header.ts, c, w, q));
      let next = handlers.hRequest (C c, W w, Q q) in
      w_ρ1 newSt next
and w_ρ1 : Ws_protocol.store → msg5 → result_w = fun st → function
  | Fault(result) →
      let ts1 = st.header.ts+1 in
      assume (Send_Fault (st.vars.w, st.header.sid, ts1)) in
      Ws_protocol.sendWired_Fault_ρ1 st ; result
  | Reply(X x, result) →
      let ts1 = st.header.ts+1 in
      assume (Send_Reply (st.vars.w, st.header.sid, ts1, x));
      Ws_protocol.sendWired_Reply_ρ1 x st ; result
```

Figure 3. Generated session interface (Ws.mli) and (partial) session implementation (Ws.ml); Handwritten application (WebService.ml)

intends to send a Reply message with the given parameters. These events are used in stating our security goals.

4. Session integrity

We now formalize session integrity. We say that a *system supports sessions* $S_1, \ldots, S_k$ if it consists of the ML modules:

$$Data\ Net\ Crypto\ Prins\ S_1\_protocol\ S_1\ \ldots S_k\_protocol\ S_k\ U$$

where:

- $Data$, Net, $Crypto$, and $Prins$ are symbolic implementations of trusted platform libraries; $Data$ is for bytearrays and strings, Net for networking, $Crypto$ for cryptography, and $Prins$ maps principals to cryptographic keys (see Section 7.1 and Section 8);
- for each session specification S_i, the modules $S_i\_protocol$ and S_i are compiled then typechecked against the refined type interfaces of $S_i\_protocol$, S_i, and the libraries;
- U represents the application code, as well as the adversary, with access to all functions in $Data$, Net, $Crypto$, and S_i, and access to some keys in $Prins$ (as detailed below.)

As depicted in the session module of Figure 3, a system may log events by calling the assume function. A run of a system consists of the events logged during execution. For each session, we define three kinds of events that are observable:

$$\text{Send\_}f(a, \widetilde{v}) \quad \text{Recv\_}f(a, \widetilde{v}) \quad \text{Leak}(a)$$

where f ranges over labels in the session. Send_f asserts that in some run of the session the principal a instantiating role $\mathsf{src}(f)$ commits to sending a message labelled f, with values $\widetilde{v}$ for its written variables. Recv_f asserts that principal a instantiating role $\mathsf{tgt}(f)$, after examining the over-the-wire cryptographic evidence, accepts a message labelled f with values $\widetilde{v}$ for its read variables.

The event Leak(a) states that the principal a is compromised; a Leak event is generated whenever the adversary U accesses a key from the Prins module; in a run where a principal's keys are never accessed by U, this event does not occur, and the principal is treated as compliant. For a given run of a system supporting sessions, we say that a *compliant event of the run* is a Send or a Recv event present in the run whose first argument is a principal a for which there is no Leak(a) event anywhere in the run.

An *instance* of a session is a sequence of Send and Recv events obtained by (globally) instantiating all the bound variables of an initial path of the session.

DEFINITION 1. *The concrete instances of S are as follows:*

1. *let $f_1 \ldots f_k$ be an initial path of S;*
2. *let $\widetilde{x}_i = \mathsf{write}(f_i)$ and $\widetilde{y}_i = \mathsf{read}(f_i)$ be the written and read variables of f_i for $i = 1..k$;*
3. *let $(\alpha_i)_{i=1..k}$ be a sequence of maps from variables $\mathcal{X}$ to values for which*
   - *each map can only differ from the previous for the variables that have just been written: $\Delta_{\widetilde{x}_{i+1}}(\alpha_i, \alpha_{i+1})$ for $i = 1..k$, where Δ is formally defined in Figure 5.*
4. *replace each f_i from the path with two events*

$$\textit{Send\_}f_i(\alpha_i(\mathsf{src}(f_i)), \alpha_i\widetilde{x}_i), \textit{Recv\_}f_i(\alpha_i(\mathsf{tgt}(f_i)), \alpha_i\widetilde{y}_i)$$

5. *optionally discard the final Recv_f_k event.*

Concrete instances capture all sequences of events for a partial run of a system supporting S. In the definition, step (5) accommodates instances that end with a message sent but not yet received. Moreover, the values of the variables recorded in the events are related to each other exactly in accordance with the variable (re)writes allowed by the graph (possibly shadowing each other).

We can now state our session integrity theorem

THEOREM 1. *For any run of a system that supports sessions $\widetilde{S}$, there is a partition of the compliant events of the run into disjoint sequences such that each sequence coincides with the compliant events of a concrete instance of a session in $\widetilde{S}$.*

The theorem states that the compliant events of any run are interleavings of the compliant events that may be seen along execution of initial paths of the sessions. It means that the views of the session state at all compliant principals must be consistent. For example, in a run of the Proxy session, suppose that the client principal playing the role c and the proxy playing p are compliant, but the web service playing w may be compromised. Then, the theorem guarantees that whenever the client receives a Reply message from the web service, it must be that the proxy previously sent the web service a Forward or Resume message; the web service cannot reply to the client before or during its negotiation with the proxy and convince him to accept the message. Moreover, the values of

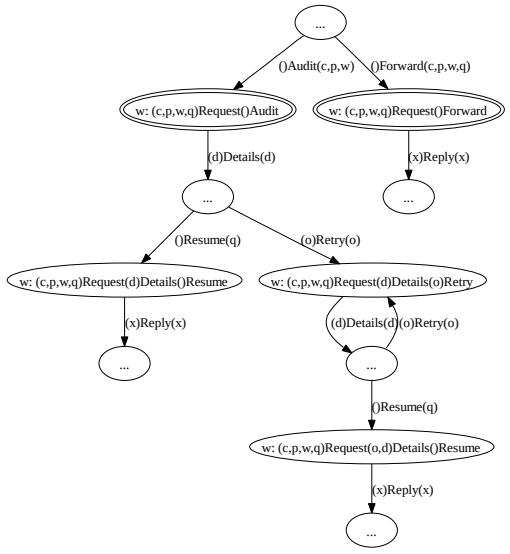

Figure 4. Internal control flow states for role w in Example 1(c).

session variables, such as q, d, o, and x, must be consistent at all compliant principals.

In Section 8, we prove a more precise version of this theorem.

5. Internal control flow states

In Section 2, session graph nodes only contain the role name. At execution time, however, such node actually represents several different session-flow states of a local role. For example, in the Proxy session, the middle node for role w denotes two different session-flow states for w: one when an Audit message is received for the first time, and another for subsequent Retry messages.

In this section we present a refinement that generates richer session graphs in which nodes can be mapped to unique session-flow role implementation states; nodes are tagged with role names and internal control flow states. Intuitively, internal control flow states, indexed by ρ, finitely enumerate the possible states a role execution can be; they consist of the last label sent by each of the roles participating in the path and the last occurrence of each variable. The internal control flow states ρ serve as indices for the various receiving and sending code in the generated protocol (see e.g. Section 3), as there are only a finite number of them even if there are an infinite number of cyclic paths. This holds since the number of labels in an internal control flow state is bounded by the size of the session's (finite) role set $\mathcal{R}$ and the number of variables (each of which is used at most once) is bounded by the size of the session's (finite) variables set $\mathcal{X}$.

We say that the extended path is *an internal control flow state*, ranged over by ρ, if it is in the image of the following state function st applied to some initial path. We let $\mathsf{st}(\widetilde{f}) = \widetilde{f} \setminus (\varepsilon, \varepsilon)$ where the filter function $\setminus$ is defined as follows:

$$\varepsilon \setminus (\widetilde{z}, \widetilde{r}) = \varepsilon$$

$$(\widetilde{f} f) \setminus (\widetilde{z}, \widetilde{r}) = \begin{cases} (\widetilde{f} \setminus (\widetilde{x}'\widetilde{z}, r\widetilde{r}))\ (\widetilde{x}')f & \text{if } r \notin \widetilde{r} \\ (\widetilde{f} \setminus (\widetilde{x}'\widetilde{z}, \widetilde{r}))\ (\widetilde{x}') & \text{elsif } r \in \widetilde{r} \end{cases}$$

where $r = \mathsf{src}(f)$, $\widetilde{x}' = \mathsf{write}(f) \setminus \widetilde{z}$, and we quotient all extended paths by the coalescing of adjacent variable vectors, i.e. $(\widetilde{x}_1)(\widetilde{x}_2)f = (\widetilde{x}_1\widetilde{x}_2)f$. Intuitively $\widetilde{f} \setminus (\widetilde{z}, \widetilde{r})$ sweeps through $\widetilde{f}$, right to left, filtering out any labels sent by roles already encountered (as recorded by $\widetilde{r}$) and any written variables already encountered (as recorded by $\widetilde{z}$).

From internal control flow states to graphs Given a session graph, we explore its paths (possibly unfolding through loops) to discover all internal control flow states; we then build a refined graph with nodes as internal control flow states and, using path information, tie them together via appropriate edges.

For example, the refined graph of the Proxy session, projected for role w, is shown in Figure 4. There, states labelled with dots "..." represent unknown states for w, and may comprise in turn of several states between the remaining roles (there is no space to show the full graph here). Each incoming edge to a node of role w represents a message that is received by w, while every outgoing edge represents an message sent by w to another role.

Role w has two initial internal control flow states, depending on whether it receives an Audit or a Forward message. Both states have the common prefix (c,p,w,q)Request, meaning that the initial role c must have sent a Request message writing variables c,p,w,q. In the case w sends a Details message writing variable d, it next jumps to two different states, depending on whether a Resume message arrives or a Retry message arrives. In the latter case, w needs to send another Details message rewriting d, which may result in either another Retry as response or a final Resume message to which w answers finally with a Reply message writing variable x.

Clearly, the internal control flow states of a role provide a finite way to represent the (usually infinite) execution states (accordingly to the session graph paths). From Figure 4, one may project back to the local syntax and obtain a decorated local role process, as follows:

```
role w : string =
  recv [{w:(c,p,w,q)Request()Audit} Audit (c,p,w) →
    send Details (d) ;
    recv [w:(c,p,w,q)Request(d)Details()Resume Resume(q) → Reply(x),
          X: w:(c,p,w,q)Request(d)Details(o)Retry Retry(o);
            send Details(d),
          w:(c,p,w,q)Request(o,d)Details()Resume → send Reply(x)
],
        {w:(c,p,w,q)Request()Forward} Forward(c,w,q) → send Reply(x) ]
```

Although this operation generates graphs with potentially more nodes and edges than in the original one, the generated graph still remains finite. However, the refinement does introduce potentially many new nodes. As a rough estimate, the total number of generated nodes doubles per loop in the original session graph, since we need to unfold loops into the two cases of entering for the first time and subsequent entries (for statistics, see Section 9).

6. Cryptographic protection

We now turn our attention to the cryptographic wire format for the messages exchanged between roles. For an edge of the form $(\widetilde{x})f(\widetilde{y})$, we protect the confidentiality of the written variables $\widetilde{x}$, provide access to the read variables $\widetilde{y}$ to w, and provide message integrity for w, giving evidence that the message comes from intended the source role. However, the situation is more complex than that, as roles may need to forward and check evidence from other involved roles. For example, in the Proxy example of Figure 1(c), when w receives an Audit message in the Proxy example of Figure 4, it requires not only evidence about $()Audit(c, p, w)$ from p, but also evidence that c sent $(c, p, w, q)Request(c, p, w, q)$. This is why we decorate internal control flow states as described above: at the state of receiving the Audit, role w has an internal control flow state that expects evidence from both c and p.

By computing internal control flow states, we know exactly what evidence needs to be forwarded at runtime. Cryptographically, we provide integrity by applying a message authentication code (MAC) to each sent label together with the store variables and their values, which are hashed together with a confounder, to avoid dic-

tionary attacks. For confidentiality, variable values are encrypted. In general, each message consists of (1) a series of MACs from the sender and earlier participants, intended to provide integrity of the session path; (2) a series of (cryptographically hashed) variables needed by the receivers to recompute and verify the MACs; (3) a series of (possibly encrypted) variables with their current values, for the readable variables of the receiver; and (4) a set of encrypted session keys, for the initial contact between the sender and receiver.

Initial Proxy Request message Consider the first receipt of an Request message by p from c at the beginning of the Proxy example. The internal control flow state for p is (c,p,w,q)Request, denoting that variables c, p, w, q are written by c in the initial Request. Target role p has access to all the written variables, i.e. c, p, w, q. The actual format of the message sent by p for c is as follows:

$$sid \mid 0 \mid \ell_0 \mid mac_{c,p}(sid \mid 0 \mid \ell_0 \mid h[c] \mid h[p] \mid h[w] \mid h[q]) \quad (1)$$
$$\mid c \mid c_x \mid p \mid p_x \mid w \mid w_x \mid enc_{c,p}(q \mid q_x) \quad (2)$$
$$\mid asign_c(aenc_p(\text{`}c\text{'} \mid k^a(c,p) \mid k^e(c,p))) \quad (3)$$
$$\mid mac_{c,w}(sid \mid 0 \mid \ell_0 \mid h[c] \mid h[p] \mid h[w] \mid h[q]) \quad (4)$$
$$\mid asign_c(aenc_w(\text{`}c\text{'} \mid k^a(c,w) \mid k^e(c,w))) \quad (5)$$

Here, $\mid$ denotes concatenation, $mac_{x,y}(m)$ denotes the message authentication code of m under the shared key between x and y; $h(m)$ denotes the hash of message m; to ease notation, we write $h[v] = h(sid \mid sr \mid \text{`}v\text{'} \mid v \mid v_x)$ to denote the hash of the (fresh) session id sid, concatenated with the source role principal sr, concatenated with the variable tag 'v', its current value v and finally a fresh confounder v_x; 0 is the initial timestamp; ℓ_0 is the tag '(c,p,w,q) Request'; $enc_{x,y}(m)$ denotes the symmetric encryption of m under shared key between x and y; $aenc_x(m)$ denotes the asymmetric encryption of m under public key x; $asign_x(m)$ denotes the digital signature of m under private key x.

This message has two parts. Components (1)-(3) constitute information computed by c intended for p, while components (4)-(5) are for w, and will be forwarded by p to w in subsequent messages. In (3), component $asign_c(aenc_p(\text{`}c\text{'}, k^a(c,p) \mid k^e(c,p)))$ is added for key establishment, included since it's the first time c communicates with p (this step is done only once). It contains two fresh session keys $k^a(c,p)$ and $k^e(c,p)$ asymmetrically encrypted for the recipient w and digitally signed by the source role p; $k^a(c,p)$ is intended for MACing between c and p, and $k^e(c,p)$ is intended for (symmetric) encryption (so above when we write $mac_{c,p}(\dots)$ we mean $mac_{k^a(c,p)}(\dots)$, and similarly for encryption). (As explained in the next section, we assume a public key infrastructure in place). Once session keys are exchanged, (2) includes variables and their values being readable by p. Variables c, p, and w are principal variables and hence are unprotected and sent in the clear (see Section 2.1). On the other hand, variable q is a data variable and so we must protect it so that it remains confidential; it's encrypted using the shared symmetric key $k^e(cp)$.

Integrity is achieved by the MAC in (1), where all the variable hashes are concatenated, preppended with a session identifier $sid = h(\Sigma, r)$ where r is a fresh nonce chosen by the session initiator and Σ is the session definition. The message contains also the label ℓ_0 in the clear and the current timestamp (0 in this case).

Finally, (4) contains a similar MAC for c but intended for w, and session keys between c and w are included in (5).

Upon receiving this message, the principal running as p processes (3) to obtain the session keys. These keys are used to process the variables in (2); once all variables are processed, p recomputes the hashes and checks the MAC of (1). The principal variables are checked to see if the principal is indeed assigned to its, and that the sid is not a replay. If the checks succeed, role p updates the session store and invokes the user code handler.

Second Proxy message: Audit Now p wants to continue the session and send an Audit message to w; the message includes

$$sid \mid 1 \mid \ell_1 \mid mac_{p,w}(sid \mid 1 \mid \ell_1 \mid h[c] \mid h[p] \mid h[w])) \quad (6)$$
$$\mid c \mid c_v \mid c_x \mid p \mid p_v \mid p_x \mid \quad w \mid w_v \mid w_x \quad (7)$$
$$\mid asign_p(aenc_w(\text{`}p\text{'} \mid k^a(p,w) \mid k^e(p,w))) \quad (8)$$

Besides the above, p forwards the previous (4)–(5) from c. Here, ℓ_1 is '(c,p,w,q)Request()Audit'; (6)–(8) are analogous to (1)–(3), but variable q is not included, and the timestamp is incremented.

7. Cryptographic compilation

We briefly describe our libraries, then explain the compilation process and discuss the implementation—additional code and details are available online.

7.1 Libraries

Our generated protocol implementation use libraries for data, networking, cryptography, and principals. These libraries (and their refined types) are essential parts of our security model.

Data manipulation A first module, $Data$, provides data types bytes (for raw bytes arrays) and str (for strings) used for networking and cryptography; its interface provides e.g. base64: bytes → str for encoding string payloads, and concat: bytes → bytes → bytes for assembling messages.

```
type str
type bytes
val cS : string → str
val iS : str → string
val base64 : bytes → str
val ibase64 : str → bytes
val utf8 : str → bytes
val iutf8 : bytes → str
val concat : bytes → bytes → bytes
val iconcat : bytes → (bytes * bytes)
```

Cryptographic library $Crypto$ provides the functions rsa_encrypt, rsa_decrypt, sym_encrypt, sym_decrypt for encryption (RSA and AES). For automated verification by typing (Section 8), keys are given types α hkey when they are MACing keys and α symkey when they are symmetric keys. MACing (HMACSHA1) is handled by functions mac and mac_verify while hashing (SHA1) uses sha1 and sha1_verify. A fresh nonce is created by a call to function mkNonce. Marshalling to and from bytearrays is realized through pickle and unpickle.

The interface crypto.mli is as follows:

```
(* Marshalling *)
type α pickled
val pickle: α → α pickled
val unpickle: α pickled → α
(* Cryptography *)
type α hkey
type α symkey
val mkNonce : unit → bytes
val sha1 : bytes → bytes
val sha1_verify : bytes → bytes → unit
val rsa_encrypt : bytes → bytes → bytes
val rsa_decrypt : bytes → bytes → bytes
val mac : α hkey → α pickled → bytes
val mac_verify : α hkey → bytes → bytes → α pickled
val sym_encrypt : α symkey → α pickled → bytes
val sym_decrypt : α symkey → bytes → α pickled
```

Principal and networking Library The *Prins* library manages principals and their associated data, along with principal-related networking function. The interface is as follows:

```
type pr = { id:string; cert:string; ip:string; port:int;}
val register : pr → unit
val psend : str → str → unit
val precv : str → str
val bind : str → unit
val close : unit → unit
val check_cache : string → string → bytes → unit
val gen_keys : str → str → bytes
val reg_keys : str → str → bytes → unit
val get_privkey : str → bytes
val get_pubkey : str → bytes
val get_symkey : str → str → α symkey
val get_mackey : str → str → α hkey
```

The *Prins* library maintains a database of principals. recording the principal name, its public-key certificate, and its network address. We assume an existing public key infrastructure (PKI) in which each principal has a public/private keypair and knows the other principals' public keys. *Prins* also maintains an anti-replay cache for each principal, containing session identifiers and roles for all sessions it has joined, to ensure that it never joins the same session twice in the same role. During a session, whenever a principal contacts another principal for the first time, function gen_keys generates fresh session keys and protect them using asymmetric cryptography. Function reg_keys is used to process incoming keys and registering them (in the next section we explain this in more detail). In each session instance, our implementation can then use the symmetric and MAC keys via functions get_symkey and get_mackey.

The *Prins* library also provides functions psend and precv for sending and receiving messages between principals, relying on the *Net* library. *Net* provides networking functions, but is never accessed directly by our code; it may be accessed by application code for non-session communications.

7.2 Code generation

Generating the cryptographic protocol implementation requires preparatory computations on the refined graph presented in Section 5. First, internal control flow states along with the refined graph are used to compute a *visibility* relation, which details for every receiving state a list of MACs to be checked in incoming messages, along with the expected contents; (potentially MACs are expected in a state, from each of the roles involved since its own last involvement). Each MAC is expected to contain a hash of the variables that have been bound or rebound so far in the path.

From the *visibility* relation, a *future* relation is derived to associate each message sent in the refined graph with a list of the roles that may expect a MAC of that message, and which variables that role is expecting. Relation *future* yields the *fwd_macs* relation associating messages with MACs to be transmitted along the way.

We also compute a *learnt* relation which hashes (coming from commitments or supporting hashes) does this role learn in a given message. From this relation, commitment checks are inferred. It is also used (with *future*) to derive the *fwd_hashes* relation associating messages with hashes to forward. Finally, *future* is used to derive a *fwd_keys* relation which associates messages with (encrypted, session-establishment) keys that need to be forwarded.

These relations are helpful to statically know which MACs are generated before sending a message, what the content of each message is and which verifications are done upon reception.

Store Updated throughout the execution, the store contains the values of the variable received, some hashes of variables, some MACs, a logical clock, and the session id. We use the notation ← to designate a store with an updated field.

```
type store = {
  vars : { for each (x:t) ∈ 𝒳, [ x: t ] } ;
  hashes : { for each x ∈ 𝒳, [ hx : hashstore ] } ;
    (* hashstore is a container for hash values and confounders *)
  macs : { for all label l with variables x̃ in a visible sequence
         received by r, [ rlx̃ : bytes ] } ;
  keys : { for each pair r,r' of roles, [ key_r r': bytes ] } ;
  header = { ts : int ; sid : bytes }}
```

Auxiliary functions The following content functions build the MACs used in the protocol (as described in Section 6).

```
For all state ρ mentioned in a visible sequence with variables z̃
[let content_ρ_z̃ = fun ts store →
 fold over z ∈ z̃ [let hashes = concat store.hashes.hz.hash hashes in]
  let state = utf8 (cS "ρ") in
  let payload = concat state hashes in
  let header = concat store.header.sid (utf8 (cS (string_of_int ts))) in
  concat header payload]
```

The mac_verify functions each check the correctness of a particular MAC, given the store, time-stamp, key and the received MAC. Their definition as individual functions is required by our proof technique.

```
  For all state ρ mentioned in a visible sequence received by role r
   with variables z̃
[let mac_verify_r_ρ_z̃ = fun ts store key mac →
  let content = content_ρ_z̃ ts store in
  mac_verify k m content ]
```

Sending functions For each message (i.e. edge) in the refined graph, the compiler generates a sendWired function that builds and sends a message (as detailed in Section 5).

```
For all edge ρ --(x̃) f (ỹ)--> ρ' of the refined graph, the sending role is r
r' is the receiving role.
 [let sendWired_f_ρ store = fun x̃ store →
  fold over x ∈ x̃ [ store.vars.x ← x ; store.hashes.hx ← sha1 x ;]
   store.header.ts ← store.header.ts + 1;
  fold over r, r' ∈ fwd_key(ρ)
   [let keyrr' = gen_keys store.vars.r store.vars.r' in
    let keys = concat keyrr' keys in]
  for all (r'', ρ'', z̃) ∈ future(ρ, l) (header is built as in content)
   [let content = content_ρ''_z̃ store.header.ts store in
    let mackeyrr'' = get_mackey store.vars.r store.vars.r'' in
    let macmsg = mac mackeyrr'' (pickle content) in
    let r'fx̃ = concat header macmsg in]
  ... Marshalling sent MACs (fwd_macs) ...
  ... Marshalling hashes (fwd_hashes) ...
  fold over y ∈ ỹ [let keyrr' = get_symkey store.vars.r store.vars.r' in
                   let encr_y = sym_encrypt keyrr' (pickle mar_y) in
                   let variables = concat encr_y variables in]
  ... marshalling of confounders for variables ỹ ...
  ... Building header and message ...
  let () = psend store.vars.r' msg in
  store]
```

Receiving functions For each receiving state sequence, the compiler generates a sum type with a constructor for each possible return values of the receiveWired functions.

```
For all receiving state ρ,
 [type wired_ρ = for each f that can be received in state ρ
                 [ | Wired_f_ρ of [types of Read(f)] * store] ]
```

The receiveWired function checks whether the received message is initial or not: in the former case, the cache needs to be checked for guarding against replay attacks; in the latter, only session id verification and time-stamp progress are necessary.

Once the header of an incoming message is checked, the receiving code verifies the included visible sequence is acceptable. Then, the protocol unmarshalls and decrypts variables and keys (read and fwd_keys), checks commitments (that is, adequacy between an already known hash (i.e. not *learnt*) of a value that has now become readable), unmarshalls hashes (fwd_hashes), unmarshalls MACs (fwd_macs), checks MACs (visib), and finally returns the corresponding Wired data type.

```
For all receiving state ρ of role r
[let receiveWired_ρ : store → wired_ρ = fun store →
  let msg = precv store.vars.r in
  ... Header/Cache checks x̃ ...
  let tag,payload = iconcat content in
  match iS (iutf8 tag) with
 for each v visible sequence accepted at state ρ
 [| "v" →
  ... Unmarshalling read variables ỹ and confounders ...
  ... Unmarshalling and registering of new keys (fwd_keys) ...
  ... Decrypting variables ...
  ... Verifying variables against known hashes (learnt) ...
  ... Unmarshalling hashes (fwd_hashes) ...
  ... Unmarshalling MACs (fwd_macs) ...
  for all (r'', ρ', z̃) ∈ v (header is built as in content)
   [ ... checking time-stamp order ...
    let mackeyrr'' = get_mackey store.vars.r store.vars.r'' in
    let content = content_ρ'_z̃ ts store in
    mac_verify mackeyrr'' store.macs.r''f'z̃ content; ]
    Wired_f_ρ((x̃),store) ]
```

Proxy code The types corresponding to user code are generated based on the local roles.

```
type msg0 =
    Query of (var_q * var_c * var_s * msg1)
and msg1 = {
    hReply : (var_x → result_c)}
```

The proxy functions are indexed by state views.

```
let rec c_Query (st:store) : msg1 → result_c = function handlers →
  let r = receiveWired_c_Query st in
  match r with
  | Wired_Reply_Query ((x), newSt) →
    let next = handlers.hReply (X x) in
    next
and c_start (st:store) : msg0 → result_c = function
  | Query(Q q, C c, S s,next) →
    let newSt = sendWired_Query_start (q,c,s) st in
    c_Query newSt next
```

8. Sessions as path predicates

We prove our main theoretical result, Theorem 1, which states the integrity of session executions as observed by compliant principals despite the presence of arbitrary coalitions of compromised ones.

Our proof proceeds as follows: we first enrich send and receive events with additional parameters and lift the definition of concrete instances accordingly (Definition 2); for each session, we define families of predicates that capture invariants that must be maintained by a session implementation (Figure 5); we define typed interfaces for our generated code, and show that if the code meets these types, then it maintains these invariants (Lemma 1); we prove, by hand, that the implementation of each role is locally sequential (Lemma 2); using the invariants and local sequentiality, we establish the integrity theorem for all code that is generated by our compiler and typechecked (Theorems 1 and 2).

8.1 Stores, timestamps, and enriched events

We introduce a series of definitions and helper functions, then enrich the contents of events.

The function write collects the variables written on an extended path (Section 2):

$$\mathsf{write}((\widetilde{x}_0)f_0 \dots (\widetilde{x}_k)f_k) = \{\widetilde{x}_0, \dots, \widetilde{x}_k\}$$

Given a role r and an initial path $\widetilde{f}$, the function $\mathsf{knows}(r, \widetilde{f})$ collects the variables in scope for role r after $\widetilde{f}$:

$$\mathsf{knows}(r, \varepsilon) = \emptyset \quad \text{and} \quad \mathsf{knows}(r, \widetilde{f}f) = (\mathsf{knows}(r, \widetilde{f}) \setminus \widetilde{x}) \cup \{\widetilde{z}\}$$

where $\widetilde{x}$ are the written variable of f; and $\widetilde{z}$ are either the written variables of f if $r = \mathsf{src}(f)$, the read variables of f if $r = \mathsf{tgt}(f)$, or $\emptyset$ otherwise. For instance, for Example 1(a) we have that $\mathsf{knows}(c, \text{Request Reply}) = \{c, w, q, x\}$ and $\mathsf{knows}(c, \text{Request Fault}) = \{c, w, q\}$.

Stores, ranged over by σ (and decorated variants), consist of triples $\sigma_\mathsf{h}, \sigma_\mathsf{v}, \sigma_\mathsf{c}$, where each component maps variables $\mathcal{X}$ to values. The intended use of the components is as follows: σ_h maps variables to hashes, σ_v maps variables to (their known) values, and σ_c maps variables to confounders.

The store σ is *consistent* for variables $\widetilde{z}$ with a session identifier s, written $H_{\widetilde{z}}(\sigma, s)$, if the hash map applied to a variable in $\widetilde{z}$ yields an identical hash to the one computed from the value and confounder components (see Figure 5).

Enriched events are obtained from those of Section 4 by adding timestamps and stores (following the implementation in Section 7):

$$\text{Send\_}f(a, s, ts, \widetilde{v}, \sigma) \qquad \text{Recv\_}f(a, s, ts', ts, \widetilde{v}, \sigma)$$

Timestamps, ranged over by ts (and decorated variants), are natural numbers. The timestamp ts in Send and Recv events records the time at which the event is issued and we refer to it as the *upper timestamp* of the event; in Recv events, ts' also records the time at which the role $\mathsf{tgt}(f)$ previously sent a message, or 0 if no previous message was sent; σ is the local store of a when the Send and Recv event is issued.

We lift Definition 1 accordingly:

DEFINITION 2. *The concrete instances of S are as follows:*

1. *let $f_1 \dots f_k$ be an initial path of S;*
2. *let $\widetilde{x}_i = \mathsf{write}(f_i)$ and $\widetilde{y}_i = \mathsf{read}(f_i)$ be the written and read variables of f_i for $i = 1..k$;*
3. *let s be a value, and $(\sigma_i)_{i=1..k}, (\sigma'_i)_{i=1..k}$ two sequences of stores such that*
   - *$H_{\mathsf{knows}(\mathsf{src}(f_i), f_1 \dots f_i)}(\sigma_i, s)$ for $i = 1..k$;*
   - *$H_{\mathsf{knows}(\mathsf{tgt}(f_i), f_1 \dots f_i)}(\sigma'_i, s)$ for $i = 1..k$;*
   - *each hash map may differ from the previous only on variables that have just been written: $\Delta_{\widetilde{x}_{i+1}}(\sigma_{\mathsf{h}i}, \sigma_{\mathsf{h}i+1})$ and $\Delta_{\widetilde{x}_{i+1}}(\sigma'_{\mathsf{h}i}, \sigma'_{\mathsf{h}i+1})$ for $i = 1..k-1$ (see Figure 5 for the definition of Δ);*
   - *the send and receive hash maps are equal: $\sigma_{\mathsf{h}i} = \sigma'_{\mathsf{h}i}$ for $i = 1..k$.*
4. *let $(ts'_i)_{i=1..k}$ and $(ts_i)_{i=1..k}$ be timestamps such that $ts_1 \leq \dots \leq ts_k$ and $ts'_i \leq ts_i$ for $i = 1..k$;*
5. *replace each $(\widetilde{x}_i)f_i$ in the path with two events*

$$\textit{Send\_}f_i(\sigma_{\mathsf{v}i}(\mathsf{src}(f_i)), s, ts_i, \sigma_{\mathsf{v}i}\widetilde{x}_i, \sigma_i),$$
$$\textit{Recv\_}f_i(\sigma'_{\mathsf{v}i}(\mathsf{tgt}(f_i)), s, ts'_i, ts_i, \sigma'_{\mathsf{v}i}\widetilde{y}_i, \sigma'_i)$$

6. *optionally discard the final Recv_f_k event.*

Meta predicates:

Consistency of stored hashes $\quad H_{\widetilde{z}}(\sigma, s) \triangleq \bigwedge_{x \in \widetilde{z}} \sigma_{\mathsf{h}} x = h(s \mid \text{`}x\text{'} \mid \sigma_{\mathsf{v}} x \mid \sigma_{\mathsf{c}} x)$

Store updates $\quad \Delta_{\widetilde{z}}(\sigma, \sigma') \triangleq \bigwedge_{x \in \mathcal{X} \setminus \widetilde{z}} \sigma(x) = \sigma'(x)$

Up to compromise $\quad \lfloor C \rfloor_a \triangleq C \vee \mathrm{Leak}(a)$

Base cases for Q and Q': $\forall s.\ Q\_\varepsilon(s, 0, \emptyset)$ and $\forall s.\ Q'\_\varepsilon(s, 0, 0, \emptyset)$.

Inductive case for Q: For every internal control flow state $\rho = \hat{f}\,(\widetilde{x})f$ we let:

$$\forall s, ts, \sigma.\ Q\_\rho(s, ts+1, \sigma) \Leftrightarrow \mathrm{Send}\_f(\sigma_{\mathsf{v}}(\mathsf{src}(f)), s, ts+1, \sigma_{\mathsf{v}}\widetilde{x}, \sigma) \wedge H_{\widetilde{x}}(\sigma, s) \wedge \bigvee_{\rho' \lhd \rho} \left(\exists \sigma', ts'.\ Q'\_\rho'(s, ts', ts, \sigma') \wedge ts' < ts \wedge \Delta_{\widetilde{x}}(\sigma'_{\mathsf{h}}, \sigma_{\mathsf{h}}) \right)$$

Inductive case for Q': For every non-empty internal control flow state $\rho_k = \hat{f}\,(\widetilde{x}_1)f_1 \ldots (\widetilde{x}_k)f_k$ for which $\mathsf{tgt}(f_k)$ is not active on $f_1 \ldots f_k$ and either $\hat{f}$ is empty or the source role of the last edge in $\hat{f}$ is $\mathsf{tgt}(f_k)$, with $\widetilde{y} = \mathsf{read}(f_k)$, $\widetilde{x}'_i = \mathsf{write}(f_i)$ for $i = 1..k$, and $\widetilde{z} = \mathsf{write}(\rho_k) \setminus (\widetilde{x}_1 \ldots \widetilde{x}_k \cup \{\mathsf{src}(f_1), \ldots \mathsf{src}(f_k)\})$, we let:

$$\forall s, ts_0, ts_k, \sigma.\ Q'\_\rho_k(s, ts_0, ts_k, \sigma) \Leftrightarrow \mathrm{Recv}\_f_k(\sigma_{\mathsf{v}}(\mathsf{tgt}(f_k)), s, ts_0, ts_k, \sigma_{\mathsf{v}}\widetilde{y}, \sigma) \wedge H_{\widetilde{y}}(\sigma, s) \wedge \bigvee_{(\rho_0, \rho_1, \ldots, \rho_k)_\lhd} \begin{pmatrix} \exists \sigma_0, \ldots, \sigma_k, ts_1, \ldots, ts_{k-1}. \\ \wedge_{i=0..k-1} \left(ts_i < ts_{i+1} \wedge \Delta_{\widetilde{z}\widetilde{x}'_{i+1}}(\sigma_{\mathsf{h}i}, \sigma_{\mathsf{h}i+1}) \right) \wedge \Delta_{\widetilde{z}}(\sigma_{\mathsf{h}k}, \sigma_{\mathsf{h}}) \wedge \\ \wedge_{i=1..k} \left(\lfloor Q\_\rho_i(s, ts_i, \sigma_i) \rfloor_{\sigma_{\mathsf{v}}(\mathsf{src}(f_i))} \right) \wedge Q\_\rho_0(s, ts_0, \sigma_0) \end{pmatrix}$$

Figure 5. Definition of the predicates Q and Q'.

Note that this definition does not specify a strict ordering on the timestamps in order to avoid having to distinguish between compliant events (where there is a strict ordering as seen in Lemma 2) and non-compliant ones (where we cannot guarantee strictness).

8.2 Invariant path predicates

Figure 5 defines a pair of families of predicates that serves as the invariant at each point an enriched send or receive event is emitted by the generated implementation code for sessions. The invariant reflects the complexity of our optimized protocol; the invariant proof is established mechanically by typechecking.

The two predicates, Q and Q', hold at each send and receive event respectively. Informally, they have the following interpretation. Consider any internal control flow state $\rho = \hat{f}\,(\widetilde{x})f$; then:

- $Q\_\rho(s, ts, \sigma)$ asserts that the principal playing role $\mathsf{src}(f)$ in a session instance with identifier s is satisfied that its global execution has followed an initial extended path whose image under st is ρ, with the final step of the execution being the send of f at timestamp ts; moreover, the current values for all the variables written along ρ (i.e. the state after the send) are in the store σ.
- $Q'\_\rho(s, ts', ts, \sigma)$ asserts that the principal playing role $\mathsf{tgt}(f)$ in a session instance with identifier s is satisfied that its global execution has followed an initial extended path whose image under st is ρ, with the final step of the execution being the receive of f at timestamp ts; the last time the role sent a message was at timestamp ts' (or 0 if this is the first time the role enters the session); moreover, the current values for all the variables written along ρ (i.e. the state after the receive) are in the store σ.

For simplicity, we assume in this presentation that all labels, extended paths, and internal control flow states are session specific. (Our implementation systematically qualifies them, so that two labels f belonging to two different sessions definitions are considered distinct.)

The definitions rely on a formula abbreviation $\lfloor C \rfloor_a$ that stands for the disjunction $C \vee \mathrm{Leak}(a)$, that is, either C holds or the principal a is compromised. (The generated code itself does not which principals are compromised, but this does not prevent our invariant to mention $\mathrm{Leak}(a)$.)

The definition of Q also relies on the notion of an internal control flow state ρ *preceding* an internal control flow state ρ', written $\rho \lhd \rho'$, which holds if there exists an initial path $\widetilde{f}f$ such that $\rho = \mathsf{st}(\widetilde{f})$ and $\rho' = \mathsf{st}(\widetilde{f}f)$. In this way we induce an edge relation $\lhd$ on internal control flow states that refines the underlying edge relation $\mathcal{E}$ on nodes, since a single node may correspond to several possible internal control flow states depending on the history of the session execution up to that node.

We generalize this binary notion to vectors of internal control flow states (used in Q'), writing $(\rho_0, \rho_1, \ldots, \rho_k)_\lhd$ iff there exists an initial path $\widetilde{f}_0\widetilde{f}_1 f_1 \ldots \widetilde{f}_k f_k$ such that the active roles of $\widetilde{f}_i$ are included in the active roles of $f_i, \ldots, f_k$ for $i = 1..k$; $\rho_i = \mathsf{st}(\widetilde{f}_0\widetilde{f}_1 f_1 \ldots \widetilde{f}_i f_i)$ for $i = 1..k$; $\rho_0 = \mathsf{st}(\widetilde{f}_0)$. In this definition, $f_1, \ldots, f_k$ are the *last* edges from the k rightmost active roles in ρ: this is true because each step from f_{i+1} to f_i skips over the edges $\widetilde{f}_{i+1}$ all of whose active roles are already used in $f_i \ldots f_k$. Therefore, each ρ_i is the internal control flow state for the last message send performed by role $\mathsf{src}(f_i)$, for $i = 1..k$.

8.3 Proofs by typing

The following lemma states that the path predicates are invariants maintained by any system generated by our compiler and verified by our typechecker.

LEMMA 1. *For any run of a system that supports sessions $\widetilde{S}$, for any session $S \in \widetilde{S}$, for any session identifier s running S, for any compliant principal a,*

- *the event $\mathrm{Send}\_f(a, s, ts, \widetilde{v}, \sigma)$ in the run implies that there exists an internal control flow state ρ of S ending in the sent label f, such that $a = \sigma_{\mathsf{v}}(\mathsf{src}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{x}$, and $Q\_\rho(s, ts, \sigma)$ where $\widetilde{x}$ are the written variables of f;*
- *the event $\mathrm{Recv}\_f(a, s, ts', ts, \widetilde{v}, \sigma)$ in the run implies that there exists an internal control flow state ρ of S ending in the received label f, such that $a = \sigma_{\mathsf{v}}(\mathsf{tgt}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{y}$, and $Q'\_\rho(s, ts', ts, \sigma)$ where $\widetilde{y}$ are the read variables of f;*

As outlined below, our compiler generates a dependently-type interface for the protocol module that decorates the type of each send and receive function with the preconditions and postconditions that must hold at that stage of the session. The interfaces generated by our compiler can be seen as a proof outline that is then filled-in and verified by our typechecker. We rely on an existing type system for an extended core of F#(Bengtson et al. 2008); Appendix A highlights the syntax and some typing rules of this extended language. Typechecking shows that a program meets its declared interface; we then prove the lemma by showing that the generated interface for our code guarantees the invariants.

Generating a typed interface For every compiled session S, the compiler generates the protocol module S_protocol.ml, as well as an interface S_protocol.ml7. The interface first defines the predicates Q_ρ and Q'_ρ for every internal control flow state ρ derived from the session graph. These predicates are written in the first-order-logic syntax of our type system as predicates over the full stores maintained by the implementation. Then, for each send function sendWired_f_ρ, where $\mathsf{src}(f) = a$, and the variables sent on f are $\widetilde{x}$, the interface declares the type:

private val sendWired_f_ρ:
(s:store){ Q'_ρ_0(s) ∧ Send_f(s.vars.a,s.header.sid,s.header.ts+1,s.vars.$\widetilde{x}$)}
→ (s':store){Q_ρ(s') ∧ $\Delta_{\widetilde{x}}$(s,s')}

Here the function must be given a store s such that the predicate Q'_ρ_0 holds for s; moreover, the event Send_l with the given parameters must have been assumed before the function is called. If these preconditions are satisfied, the function returns a store s' that satisfies Q_ρ(s'), where only the variables in $\widetilde{x}$ have changed from s. The type says nothing about the message being sent on the network; it simply ensures that the relevant invariants are satisfied. Since, in the session module, every Send_f is followed by a call to this function, the precondition to the function guarantees that the invariant is preserved.

Similarly, for each receive function receiveWired_ρ that may receive one of the messages $f_1, \ldots, f_n$ at role a with values $\widetilde{y}_1, \ldots, \widetilde{y}_n$ to extend ρ to $\rho_1, \ldots, \rho_n$, respectively, the interface defines the type:

private val receiveWired_ρ: (s:store){Q_ρ_0(s)}→ (wired:wired_ρ){
(∃$\widetilde{y}_1$,s'. wired = Wired_f_1_ρ($\widetilde{y}_1$),s') ∧ $\Delta_{\widetilde{y}_1}$(s,s') ∧
(Recv_f_1(store.vars.a,store.header.sid,
store.header.ts,store,vars.$\widetilde{y}_1$) ⇒ Q'_ρ_1(s'))) ∨
...
(∃$\widetilde{y}_n$,s'. wired = Wired_f_n_ρ($\widetilde{y}_n$),s') ∧ $\Delta_{\widetilde{y}_n}$(s,s') ∧
(Recv_f_n(store.vars.$a$store.header.sid,
store.header.ts,store,vars.$\widetilde{y}_n$) ⇒ Q'_ρ_n(s')))}

Here, the precondition states that the store must satisfy Q_ρ_0, and the postcondition guarantees that, for each received message with label f_i, given the Recv_f_i event, the returned store satisfies Q'_ρ_i and only modifies the variables in $\widetilde{y}_i$. Since, in the session module, every Recv_f is preceded by a call to this function, the postcondition to the function guarantees that the invariant is preserved.

Next, we describe additional type annotations used to typecheck our generated code. (The online paper has more details.)

Type interfaces for libraries For every library module used by the implementation, such as Crypto, Prins, and Net, we provide a hand-crafted refined interface encoding our assumptions. For example, we assume that hashing using sha1 is non-invertible; this assumption is encoded in the type declaration in crypto.ml7:

val sha1 : b:bytes → h:bytes{h = Hash(b)}

The postcondition says that the returned hash can be thought of as a (one-one) constructor Hash applied to the argument. The types for mac and mac_verify encode the assumption that a MAC is a faithful and unique representation of the data being MAC-ed:

val mac : (k:α hkey) → (v:α pickled) → (m:Data.bytes)
val mac_verify : (k:α hkey) → (m:Data.bytes) →
(v':Data.bytes) → (v:α pickled){v'=v}

The type α pickled represents the marshalled representation of a value of type α. Concretely, both the abstract types α hkey and α pickled are implemented as bytestrings, and type α is used only to link the types of key and the MAC-ed value. The function mac can only be used to mac marshalled values v whose types match the type of the key k; mac_verify takes a mac m and checks that it is a MAC of the value v'; when it succeeds it returns the marshalled value v, which is the same as v but with the additional type information that it matches the type of the key.

These library interfaces encode a symbolic model of cryptography, in the tradition of Dolev and Yao. Their concrete implementations are not verified, and hence are trusted. Following Bhargavan et al.; Bengtson et al. (2008), we also define symbolic implementations of these modules that use channels for communication, algebraic datatypes for cryptography, and a private channel for the principals database. These symbolic implementations are part of our model and are the ones that appear in our theorems. The interfaces for all the library modules are *public*, in the sense that the adversary may use these functions to decrypt and encrypt messages with his own keys, send, receive, and intercept messages on the network. However, he does not get to read the keys of honest principals (he does not have access to the get_key function).

Dual implementations We follow the approach of Bhargavan et al. and provide a symbolic implementation in addition to the standard concrete implementation of these libraries.

For example, the concrete implementation of the cryptographic library uses standard cryptographic algorithms. There, the datatype for bitstrings is implemented as a byte array, and encryption is implemented as a symmetric encryption function (AES).

The symbolic implementation for the cryptographic library, on the other hand, uses algebraic datatypes and datatype constructors to model cryptographic operations. For example, the type for bitstring is defined as an algebraic datatype, and encryption is implemented as the application of a binary constructor SymEnc that represents encrypted bytes.

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 adversary can control compromised 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 adversary cannot break cryptography, guess secrets belonging to compliant principals, or tamper with communications on private channels.

Annotations for keys and auxiliary functions To prove local properties about the store, the typechecker uses function preconditions, code structure, and library interfaces, and annotations on auxiliary functions in the protocol module, such as content_ρ_$\widetilde{z}$, to collect logical constraints. For example, it tracks the relationship between the old and new local stores to prove the $\Delta_{\widetilde{x}}$ predicate and that the hashes and known variables are consistent.

To prove properties linking stores at different locations, the typechecker relies on the types of the keys used for generating and checking MACs. The interface declares a type (;sp,rp) mackey that defines all the possible usages of a MAC key in the session, between the principals sp and rp. It defines the precise structure of the value being MAC-ed and the invariants that must hold at the sender for the MAC to be generated. For example, the key type for the simple Ws session is as follows:

type (;sp:principal,rp:principal) mackey = (c:bytes){

Session S	Roles	S.session (lines)	Application code (lines)	S.mli (lines)	Graph (.dot lines)	Refined Graph (.dot lines)	S_protocol.ml+ S.ml (lines)	S_protocol.ml7 + S.ml7 (lines)	Typechecking Time
Ws	2	8	33	29	14	24	499 + 93	371 + 43	8.8s
Rpc	2	15	24	27	11	18	390 + 82	274 + 41	6.1s
Commit	2	16	29	30	14	24	505 + 98	351 + 48	10.3s
Wsn	2	10	44	33	17	48	998 + 145	754 + 59	23.6s
Fwd	3	15	38	34	11	19	485 + 96	309 + 48	8.6s
Proxy	3	28	65	53	26	80	1954 + 227	1848 + 91	2m34.1s
Login	4	28	54	63	29	74	1816 + 237	1441 + 101	1m43.4s

Figure 6. Generated file sizes and typechecking times for example sessions

```
(∃s. (s.vars.c=sp) ∧ (s.vars.w=rp) ∧
     (Q_c_cwqRequest(s) ∨ Leak(sp)) ∧
     (c = Concat(Concat(s.header.sid,
                        Utf8(Literal(SofI(s.header.ts)))),
                 Concat(Utf8(Literal ("c_cwqRequest")),
                        Concat(s.hashes.hc,
                               Concat(s.hashes.hw,
                                      Concat(s.hashes.hq,
                                             Utf8(Literal ("")
     ))))))))
∨ (∃s. (s.vars.w=sp) ∧ (s.vars.c=rp) ∧
     (Q_w_Fault_cwqRequest(s) ∨ Leak(sp)) ∧ (c = . . .)) ∨
∨ (∃s. (s.vars.w=sp) ∧ (s.vars.c=rp) ∧
     (Q_w_xReply_cwqRequest(s) ∨ Leak(sp)) ∧ (c = . . .))
} Crypto.hkey
```

This type allows three usages of the key, one for each state sequence ρ where a send is possible. To understand the intuition behind the first disjunct, recall that the mac function only allows a key to be used with a value that matches its type. Hence, this key type ensures that when a MAC is being generated, either the sending principal is compromised (Leak(sp)), or it satisfies the predicate Q_cwqRequest. Moreover, the value being MAC-ed must have the detailed concatenated structure shown in the predicate (this is the structure generated by the content_cwqRequest_cwq function). Conversely, the mac_verify function returns a value matching the key type; hence, by checking a MAC on a received Request message, the receiver knows that the predicate Q_cwqRequest holds at the remote store. Moreover, by comparing the structure of the locally constructed MAC-ed value with the one provided within the predicate, it can verify that the local and remote store are consistent over the MAC-ed hashes. These two properties are enough for it to establish the Q'_w_cwqRequest predicate.

Discussion During the design of our compiler, we found several bugs by typechecking. More often, we found that our type annotations were not strong enough to establish our results, or that our typechecker required predicates to be structured in a specific way. Discovering sufficiently strong annotations for keys, libraries, and auxiliary functions, and designing a compiler that automatically generates them requires some effort, but is rewarded with an automated verification method. We have used this method to typecheck several examples; their verification time and other statistics are listed in Section 9.

Secrecy By typechecking, we also obtain secrecy for values assigned to session variables, under the assumption that the application code run by compliant principals is trusted to provide secret values for these variables and not leak them to the adversary. Informally, the value assigned to a variable in a session run may be obtained by the adversary only if a compromised principal plays a role in the session that can read the variable. To verify this property we annotate the encryption and decryption keys in the protocol module with refined types, and check that these types are met by all encryption and decryption operations.

8.4 Local sequentiality

We now complete the proof by hand (as our typechecker does not keep track of linearity). We establish that the implementation of each role in a session must be locally sequential:

LEMMA 2. *In any run of a system that supports session S, if the principal a is compliant, then for any role r and session identifier s for S, the series of events emitted by a with s in role r forms a alternation of sends and receive events such that for any adjacent pair of such events*

$$\mathit{Send\_f}(a, s, ts_0, \widetilde{v}, \sigma_0), \mathit{Recv\_g}(a, s, ts_1, ts_2, \widetilde{w}, \sigma_2) \text{ or } \mathit{Recv\_g}(a, s, ts_1, ts_2, \widetilde{w}, \sigma_2), \mathit{Send\_f}(a, s, ts_3, \widetilde{v}', \sigma_3)$$

we have $ts_0 = ts_1$, $ts_1 < ts_2$, and $ts_2 + 1 = ts_3$.

The proof is by inspection of the code structure in the generated session module. For example, in the generated session module Ws.ml of Figure 3, we observe that the sequence of events emitted by the role function w must be strict alternation, and by the path invariants of Lemma 1, the timestamps must be in the order prescribed.

PROOF: There are two possible ways for a principal a to enter a session: either it has the initial role and it initiates the session with a fresh session identifier s. Or, it joins an existing session in a non-initiator role, in which case the anti-replay cache prevents a from joining two sessions with the same identifier s and role r.

In both cases, the events emitted by principal a with s in role r are entirely determined by a single sequential execution of the code that implements r. For all the events, the session identifier is statically bound to s.

After emitting a Send event, the code that follows is an affine context that guards a single emission of a Recv event, and vice versa. This is a structural property of the code and is not dependent on state or cryptography.

We now consider the proof obligations concerning timestamps. For each message received from the network with timestamp ts_2, the code performs a check before emitting the corresponding $\mathrm{Recv\_g}(a, s, ts_1, ts_2, \widetilde{w}, \sigma_2)$ event to verify that ts_2 is strictly greater than the last timestamp ts_0 the same code sent (or 0 if this is the first receive) and then chooses $ts_1 = ts_0$ when emitting the event. For each event $\mathrm{Send\_f}(a, s, ts_3, \widetilde{v}', \sigma_3)$ recording a message send, the code chooses $ts_3 = ts_2 + 1$. □

8.5 Proof of integrity

We can now prove a more precise version of Theorem 1, whose statement we recall here: "For any run of a system that supports sessions $\widetilde{S}$, there is a partition of the compliant events of the run into disjoint sequences such that each sequence coincides with the compliant events of a concrete instance of a session in $\widetilde{S}$." When proving this theorem, a certain subtlety arises in characterising of the "partitions". Whereas events with different session identities are in separate partitions, we need a finer grained partitioning that

distinguishes some events with the same session identity. This is because an adversary may start two sessions for S giving both instances the same session identity s. The implementation of compliant principals renders such attempts to subvert integrity harmless: the anti-replay cache of each principal ensures that a principal never joins s twice in the same role. Thus a principal may participate in two instances of a session with the same session identity, but the roles it plays in each instance are disjoint.

We say that one event is *independent* of another either if they have distinct session identities, or if they have the same session identities but their stores do not contain information about the other's role. For example, the events

$$\text{Send\_}f(a, s, ts, \widetilde{v}, \sigma) \qquad \text{Recv\_}f'(a', s', ts'', ts', \widetilde{v}', \sigma')$$

are independent if $s \neq s'$ or if $s = s'$ and $\sigma(\mathsf{tgt}(f')) \neq a'$ and $\sigma'(\mathsf{src}(f)) \neq a$.

We now formally define the complementary notion of *dependence*, which we use in stating our results.

DEFINITION 3. *A pair* (s, σ) depends *on an event Send_*$f(a, s', \ldots)$ *(respectively Recv_*$g(a, s', \ldots)$*) if* $s = s'$ *and* σ *maps* $\mathsf{src}(f)$ *(respectively* $\mathsf{tgt}(g)$*) to* a*. An event with session id* s *and store* σ *depends on another event if* (s, σ) *depends on the latter event.*

We then show that the predicates Q and Q' imply that the events constitute concrete instances, from which Theorem 2, and hence Theorem 1 follow.

LEMMA 3. *For every run, if* $Q\_\rho(s, ts, \sigma)$ *or* $Q'\_\rho(s, ts', ts, \sigma)$ *holds, then there is an instance of an initial extended path ending in state* ρ *that matches the subsequence of all compliant events depended upon by* (s, σ) *with upper timestamps at most* ts*.*

Moreover any event with timestamp at most ts *that depends on an event in this subsequence is itself in the subsequence.*

PROOF: (Sketch.) By mutual induction on ts with lexical ordering so that we may use the property for Q at ts to prove it for Q' at ts (but not vice versa). By expanding the Q and Q' predicates with upper timestamp ts, we show that Q' and Q holds (respectively) for an upper timestamp $ts' < ts$. By induction, we get a concrete instance matching compliant events whose upper timestamps are at most ts'. We then extend these concrete instances to include the ts event. By the "no fork" condition on graphs (Property 4), we show that there is no "junk" event with upper timestamp between ts' and ts which the original ts event depends on. By the "no fork" condition, we also have that any event with timestamp at most ts that depends on an event in the concrete instance is itself in the concrete instance. □

THEOREM 2. *For any run of a system that supports sessions* $\widetilde{S}$*, for any session identifier* s*, there is a partition of the compliant events of the run into subsequences such that (1) two events are in the same subsequence iff one depends on the other; and (2) each subsequence coincides with the compliant events of a concrete instance of a session in* $\widetilde{S}$*.*

PROOF: Let $\widetilde{t}$ be the compliant events of the run and induct on the the number of events in $\widetilde{t}$. In the base case, $\widetilde{t}$ is empty and there is nothing to do.

For the inductive case, consider an event $\text{Send\_}f(a, s, ts, \widetilde{v}, \sigma)$ or $\text{Recv\_}f(a, s, ts', ts, \widetilde{v}, \sigma)$ in $\widetilde{t}$ with a maximal timestamp ts. By construction, a is a compliant principal. By typing (Lemma 1):

- if the event is $\text{Send\_}f(a, s, ts, \widetilde{v}, \sigma)$ then there exists a an internal control flow state ρ of S ending in the sent label f, such that $a = \sigma_{\mathsf{v}}(\mathsf{src}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{x}$, and $Q\_\rho(s, ts, \sigma)$, where $\widetilde{x}$ are the written variables of f;
- if the event is $\text{Recv\_}f(a, s, ts', ts, \widetilde{v}, \sigma)$ there exists a an internal control flow state ρ of S ending in the received label f, such that $a = \sigma_{\mathsf{v}}(\mathsf{tgt}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{y}$, and $Q'\_\rho(s, ts', ts, \sigma)$, where $\widetilde{y}$ are the read variables of f;

Thus by Lemma 3, there is an instance of an initial extended path ending in state ρ that that matches the subsequence of all compliant events whose upper timestamps are at most ts and which are depended upon by (s, σ).

Consider any other "competitor" event not in this subsequence. Its timestamp is at most ts, since ts was chosen to be maximal. By Lemma 3, all the events depended upon by the competitor event are not in the subsequence, so we can remove the subsequence from $\widetilde{t}$ to get a strictly smaller trace and, by induction, obtain the rest of the disjoint subsequences. □

9. Evaluation

We present compilation and verification results for a series of examples. (Some of these additional examples are described in the online paper.) Figure 6 shows, for each example session S, the lines of code for the input files (S.session, Application Code), the generated session interface S.mli, the internal graph representations, and the generated session implementation modules and their refined type interfaces; the last row shows typechecking times. Our compiler is written in around 6000 lines of F#; the trusted platform libraries are 780 lines of code (plus the .NET framework); their alternate symbolic implementation is written in 520 lines of F#.

We find that even when programming with large multi-party sessions, the user only writes a few hundred lines of code and needs to read the session interface which is less than a hundred lines of code. The generated modules are several thousand lines long, but the user can rely on the typechecker to verify them in a few minutes. However, to have additional confidence in the results of typechecking, the user may want to read the predicates generated in S_protocol.ml7 and check that they correspond to his intuitive understanding of the session, but for large examples, even the size of these annotations can be prohibitively large.

We also measure the cryptographic overhead when running concrete implementations. The table below gives the total runtimes (in seconds) for completing 5000 instances for each of the sessions Wsn, Ws, and Proxy of Example 1. For this experiment, we use a Pentium 3GHz with 1G RAM, running Windows XP with .NET cryptography, and only local communications.

Cryptography	Wsn	Ws	Proxy
no crypto	1.812	2.367	9.428
SSL	2.582 (+29%)	3.008 (+21%)	n/a
our protocol	3.477 (+47%)	3.942 (+39%)	15.476 (+39%)

The cryptographic overhead of our protocol is around 40% (third row), in comparison with a protocol with no protection (first row). The benchmarks are done in a single machine; for real distributed settings, we expect this overhead to be often negligible in the face of networking overheads. We also run sessions over .NET's SSL layer (second row). For the binary sessions (Wsn and Ws), our protocol is about 18% slower, even though it is doing many more cryptographic operations. (We did not measure SSL for multi-party sessions since SSL only provides protection for two-party communications). Finally, we also implemented libraries that use OpenSSL and UDP; we find that its performance is worse, mainly due to networking (e.g., the Wsn example takes 14.994s to complete, as opposed to 3.477s in .NET/Tcp).

Cryptographic overhead We measure the cryptographic overhead. The table below shows executions for the sessions Wsn, Ws, and Proxy from Example 1.

Crypto	Wsn	Ws	Proxy
no crypto	1.812	2.367	9.428
SSL	2.582 (+29%)	3.008 (+21%)	n/a
our protocol	3.477 (+47%)	3.942 (+39%)	15.476 (+39%)

We run 5000 iterations on a Pentium 3GHz, 1G RAM, on Windows XP with Cygwin and using .NET cryptography. As can be seen, the cryptographic overhead is around 40% slower (third row) than running without cryptographic protection at all (first row). (The benchmarks were done in a single machine; for real distributed settings, we expect the cryptographic overhead to be less important than network communications.)

In order to compare the speed of our cryptographic protocol, we code a variant of our networking library in which, instead of using ordinary TCP communications, uses .NET's SSL layer (second row of the table). For the binary sessions (Wsn and Ws), we obtain that our cryptographic protocol introduces only about 18% more overhead than .NET's SSL, even though our protocol is doing many more cryptographic operations. (We did not measure SSL for multi-party sessions like Proxy since SSL only provides protection for binary, two-party communications).

We also implemented the cryptographic and networking layers using OpenSSL and UDP; due mainly to networking, the performance is worse (e.g., for the Wsn example it takes 14.994s to complete, as opposed to 3.477s in .NET/Tcp).

10. Conclusion

References

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, 2008. To appear.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. pages 139–152.

E. Bonelli and A. B. Compagnoni. Multipoint session types for a distributed calculus. In *TGC*, volume 4912 of *LNCS*. Springer, 2007.

R. Corin and P.-M. Deniélou. A protocol compiler for secure sessions in ml. In *Trustworthy Global Computing, Third Symposium (TGC'07)*, 2007.

R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 170–186, July 2007.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAC'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object-Oriented language with Session types. In *International Symposium of Trustworthy Golbal Computing*, Apr. 2005.

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.

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.

A. Guha and S. Krishnamurthi. Fingerprinting the innocent: Using static analysis for ajax intrusion detection. 2008. Draft.

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, pages 22–138. Springer, 1998.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.

R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *To appear at ECOOP08*, 2008.

D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, 2004.

J. McCarthy and S. Krishnamurthi. Cryptographic protocol explication and end-point projection. In *European Symposium on Research in Computer Security (ESORICS)*, 2008.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

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

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. Refinement types for F#

We highlight some features of the refinement type system for F# used in this paper. The reader is referred to (Bengtson et al. 2008) for the full details.

We extend a concurrent core subset of F# with the notion of an abstract log that records logical formulas variables and names in the environment. The expression assume C takes a formula C and adds it to the log, returning unit. Conversely, the expression assert C checks whether the formula C is derivable from the set of formulas recorded in the log. If it is derivable, we say the assertion *succeeds*; otherwise, we say the assertion *fails*. Either way, it always returns unit.

Formulas are written in an ordinary propositional first-order logic with equality between terms, where terms include all F# values (including functions). Any term constructed can be used as a function or predicate symbol.

We extend standard F# types with dependent functions, dependent pairs, and refinement types. These extended types appear in the (refined) type interfaces of modules, but never appear in code. A value of type $x : T \to U$ is a function that given a value N of type T returns a value of type $U\{N/x\}$. A value of type $x : T * U$ is a pair (M, N) such that M has type T and N has type $U\{M/x\}$. A value of type $\{x : T \mid C\}$ is a value M of type T such that the formula $C\{M/x\}$ follows from the log.

As usual, we define syntax-directed typing rules for checking that the value of an expression is of type T, written $E \vdash A : T$, where E is a *typing environment* The judgment $E \vdash C$ means C is deducible from the formulas mentioned in refinement types in E. For example, if E includes $y : \{x : T \mid C\}$ then $E \vdash C\{y/x\}$. When typechecking, we delegate such logical derivations to the Z3 SMT solver (de Moura and Bjørner 2008).

The introduction rule for refinement types is as follows.

- If $E \vdash M : T$ and $E \vdash C\{M/x\}$ then $E \vdash M : \{x : T \mid C\}$.

The type system includes a subtype relation $E \vdash T <: T'$, and the usual subsumption rule. Refinement relates to subtyping as follows: $\{x : T \mid C\} <: \{x : T \mid \text{True}\} <: T$.

We typecheck assume and assert as follows.

- $E \vdash$ assume $C : \{\_ : \mathbf{unit} \mid C\}$.
- If $E \vdash C$ then $E \vdash$ assert C : unit.

By typing the result of assume as $\{\_ : \mathbf{unit} \mid C\}$, we track that C can subsequently be assumed to hold. Conversely, for a well-typed assert to be guaranteed to succeed, we must check that C holds in E.

We model the adversary as some arbitrary (untyped) expression O which is given access to the protocol and the library interfaces. Our formal goal is *robust safety*, that no assert fails, despite the best efforts of an arbitrary adversary. To allow type-based reasoning about the adversary, we introduce a *universal type* Un of data known to the adversary. By definition, Un is type equivalent to (both a subtype and a supertype of) all of the following types: all base types, $(x : \text{Un} \to \text{Un})$, $(x : \text{Un} * \text{Un})$, $(\text{Un} + \text{Un})$, and $(\mu\alpha.\text{Un})$. Hence, we obtain *adversary typability*, that O : Un for all adversaries O. Secret values, such as keys, are represented by *public* types that are not subtypes of Un. Verification of code versus an arbitrary adversary is based on *robust safety by typing*; that the program is well-typed and all the variables and functions in its interface have public types.

THEOREM 3. *If $\emptyset \vdash A$: Un then A is robustly safe.*