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 the design and implementation of a compiler that, given high-level multiparty session descriptions, generates custom cryptographic protocols.

Our sessions specify pre-arranged patterns of message exchanges and data accesses between distributed participants. They provide each participant with strong security guarantees for all their messages.

Our compiler generates code for sending and receiving these messages, with cryptographic operations and checks, in order to enforce these guarantees against any adversary that may control both the network and some session participants.

We verify that the generated code is secure by relying on a recent type system for cryptography. Most of the proof is performed by mechanized type checking and does not rely on the correctness of our compiler. We obtain the strongest session security guarantees to date in a model that captures the executable details of protocol code. We illustrate and evaluate our approach on a series of protocols inspired by web services.

1. Security by compilation and typing

Taking advantage of modern programming tools, one can sometimes design, develop, and deploy complex distributed protocols in a matter of hours—relying for instance on automated proxy generators to rapidly expose existing applications as networked services. Achieving application security under realistic assumptions on the network and remote parties is more difficult. The problem is that 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 URLs or IP addresses, but leave the interpretation of these messages and addresses to the application programmer. In particular, any guarantee that involves more than two parties (say, clients, gateways, and web servers) must be carefully established by linking lower-level guarantees on messages, or by layering ad hoc cryptographic mechanisms at the application layer, for instance by embedding signatures in message payloads.

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 our approach as regards design, implementation, and security verification.

Multiparty sessions Multiparty sessions, also called session types [Gay and Hole, 1999, Hu et al., 2008], have proved to be an elegant way of specifying structured communications protocols between distributed parties. We define a language for specifying multiparty sessions. This language features a clear notion of control flow, expressed as asynchronous messages, with a shared, distributed 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 initially blinded and later revealed during protocol execution. Our design enables simple, abstract reasoning on authentication and secrecy properties of sessions. As an example, the correspondence properties traditionally established for cryptographic protocols can be read off session specifications.

On the other hand, our sessions do not attempt to capture the behaviour of the local participants: the application programmer remains in charge of deciding which sessions to join (and in particular which principals to accept as remote peers), which message to send (when the session offers a choice), which values to write, and how to treat the messages it receives.

From sessions to cryptographic protocols Whereas much work has been done on session types and expressivity, we believe that ours is the first to protect session execution integrity via the systematic generation of cryptographic protocols. We implement a compiler from the session language to custom cryptographic protocols, coded as ML modules, which can be linked to application code for each party of the protocol.

These modules can be compiled both with F# [Syme et al., 2007] using .NET cryptographic libraries, and with OCaml using OpenSSL libraries. Our compiler combines a variety of cryptographic techniques and primitives to produce compact message formats and fast processing. Hence, we shift most of the complexity of our implementation to the compiler, which generates efficient custom protocols with a minimal amount of dynamic processing. We illustrate and evaluate our implementation on a series of protocols inspired by web services.

As an important design goal, we entirely hide cryptographic enforcement from the application programmer, who may thus reason about the runtime behaviour of a session as if every participant followed precisely its high level specification. In particular, our implementation preserves the communications structure of the session, with exactly one low-level cryptographic message for every message of the session. Similarly, any low-level message that fails to verify is silently discarded and does not affect the state of the session.

Security verification Cryptographic communications are 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 an approach proposed by Bhargavan et al. [2006] 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 extended typechecker of Bengtson et al. [2008] based on the Z3 SMT solver of de Moura and Bjørner [2008]. This is a good match for our present purposes: for each session specification, our compiler generates detailed type annotations (from a predicate logic) which are then mechanically checked against actual executable code. Thus, we overcome a common limitation of cryptographic verification: 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 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). Thus, 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 session compiler (4 300 LOCs in ML). That task appears much more complex; it is beyond the capability of automatic tools at present and would require long, delicate handwritten proofs. That approach is also brittle when experimenting with language design and cryptographic optimization, and would not provide direct guarantees at the level of the generated code.

We obtain additional functional properties by typing: any well-typed user 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 address many other properties of interest, such as liveness (any participant may block our sessions), resistance to traffic analysis (only our payloads are kept secret), and mitigation of denial-of-service attacks.

Contributions In summary, our contributions are:

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

Related work We build upon earlier work [Corin et al., 2007] which explores the secure cryptographic implementation of session abstractions for a sim-

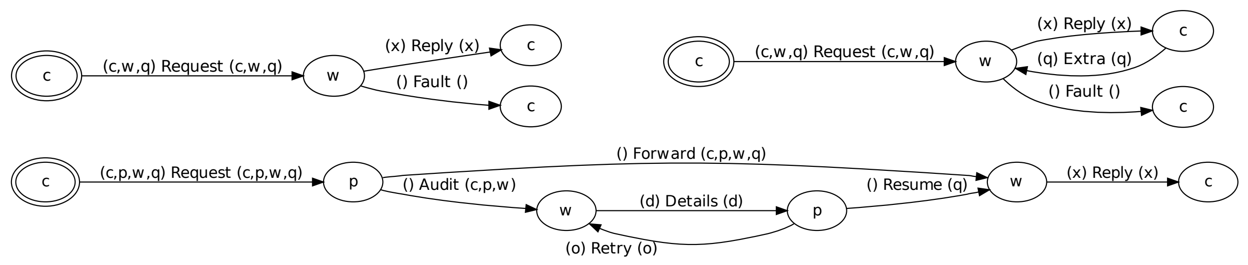

Figure 1. Sample sessions: (a) top-left: a single query (Ws); (b) top-right: an iterated query (Wsn); (c) bottom: a three-party session (Proxy).

pler language; Corin and Deniélou [2007] detail its first design and implementation. The main differences are: a much-improved expressiveness (with support 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 complete formalization of the generated code, with support for automated proofs (we used manual proofs on a simplified model).

Session types Honda et al. [2008], Bonelli and Compagnoni [2007], Vasconcelos et al. [2006], Dezani-Ciancaglini et al. [2006, 2005], Gay and Hole [1999], and 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] integrate session types in Java; McCarthy and Krishnamurthi [2008] specify 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 tackles secure implementation problems for other programming models. For instance: Malkhi et al. [2004] develop implementations for cryptographically-secured multiparty computations, based on replicated blinded computation. Zheng et al. [2003] develop compilers that can automatically split sequential programs between hosts running at different levels of security, while maintaining information-flow properties. Fournet and Rezk [2008] also propose cryptographic type systems and compilers for distributed information-flow properties. Those works consider imperative programs, whereas our sessions enable a more structured, declarative view of interactions between roles, leading to simpler, more specific security 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 theorem. Section 5 describes the cryptographic implementation. Section 6 outlines the hybrid security proof. Section 7 reports experimental results with our prototype. Additional materials, including detailed proofs and the code for all examples and libraries, are available upon request.

2. Multiparty sessions

We introduce sessions informally by illustrating their graphical representation and describing their features and design choices. We then give their formal definition and discuss some sanity and implementability conditions. 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, 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 are explained below.) 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 Ws, 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. If the token reaches a node that has no outgoing edges, session execution terminates.

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

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 transmit either a Forward message to the web service or an Audit message, indicating that further processing may be needed before accepting the request. In the later case, the protocol loops between the web service and the proxy via Details and Retry until the proxy is satisfied and sends Resume to the 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 their types are omitted from graphs for brevity) and imposes an access control discipline for writing and reading these variables, via the annotation on each edge in the graph: 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 source role assigns values to the edge's written variables, and the target role receives values of the read variables.

In Proxy (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 Forward or Audit. In both cases, the proxy may not modify q, since q does not appear anywhere else as a written variable, so the web service eventually gets the same value of q as the proxy did.

Not all variables are read by all roles. During each iteration of the Detail-Retry 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, since it 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.

In Proxy, the principal for the client role initially assigns principals to all three role variables c, p, and w, writing these in the Request message. In general, the first message need not write all the role variables, thus allowing dynamic choice of subsequent principals during session execution. However, a role variable must be instantiated before the role is used as the target of a message, and role variables may not be rewritten.

Global session graphs (definition) In preparation for our formal development in Section 4, we define 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 $(v_0 \dots v_k)$. A session graph

$$G = (\mathcal{R}, \mathcal{V}, \mathcal{X}, \mathcal{L}, m_0 \in \mathcal{V}, \mathcal{E}, R : \mathcal{V} \to \mathcal{R})$$

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 $(m, \widetilde{x}, f, \widetilde{y}, m') \in \mathcal{E}$ (where $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{X}^* \times \mathcal{L} \times \mathcal{X}^* \times \mathcal{V}$) 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.

Edges are uniquely identified by their labels, as postulated by well-formedness properties, thus we may unambiguously conflate them. 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). We use $\mathsf{src}(f) = R(m)$ and $\mathsf{tgt}(f) = R(m')$ for the *source* and *target* roles of f.

A *path* is a sequence of labels $\widetilde{f}$ where the target node of each label is the source node of the next one. 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.

Appendix A lists our well-formedness and implementability properties for session graphs. Some of these properties are simple sanity checks; for example, no role may send a message to itself and each edge must have a distinct label (properties 1 and 2). Other

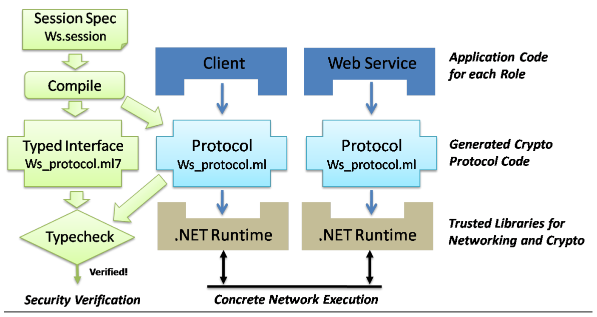

Figure 2. Compiling session programs

properties specify conditions without which the session cannot be securely implemented (at least not without extra messages). For example, property 4 excludes graphs with a branch that would enable a compromised principal to "fork" a session by sending two messages in parallel along different paths.

3. Programming with sessions

Figure 2 illustrates our framework for example Ws of Figure 1(a). The programmer first writes a session description (Ws.session). This file is compiled to generate a library module (Ws_protocol.ml) that implements all cryptographic communications for the session and provides a simple continuation-based API for each role. We verify this protocol implementation by typechecking it against an extended type interface (Ws_protocol.ml7) also generated by the compiler. The programmer then writes application code for each of the roles of the session he wishes to run, here code for the client and for the web service. Compliant principals run application code that uses the generated protocol module to run Ws sessions, but may also participate in other sessions and communications. However, compromised principals are free to use their own session protocol implementation. (The figure depicts the case where both principals are compliant, but our security theorems are more general; see Section 4.)

All our implementation code is in ML, and may be compiled either by the F# compiler using .NET libraries for networking and cryptography, or by the OCaml compiler using the OpenSSL library.

In the rest of this section, we describe the structure of the main elements of our compilation framework.

Syntactic sessions We explain the syntax of sessions by example. (Appendix B gives the full syntax, with support for loops and joins). The file Ws.session specifies the first session graph of Figure 1 as follows:

```
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 has two variables (q, x) and two roles (c, w). Each variable represents a value communicated in the session and is given a type. Each role is given a return type and a *role process* that describes the local sequences of alternating send and receive actions for this role. At every send, the role process expresses an internal choice (+) of messages that may be sent, depending on the application. At every recv, it expresses an external choice (|) of messages that may be received, depending on the other roles. Here, c sends Request, writing c, w, and q, then receives either a Reply (reading x) or a Fault.

Our syntax for sessions is local, with a process for each role, unlike the global session graphs in Section 2, but we can convert between the two representations. For example, each graph depicted in Figure 1 was automatically generated from a syntactic session during its compilation. Our compiler checks that each session yields a well-formed graph with respect to the properties in Appendix A; for example, sends and receives within a role process must alternate, and only one role may send a message with a particular label.

Generated protocol module: role functions The generated protocol module provides send and receive functions for the messages that may be sent or received in a session. These functions are not used directly by application code; instead, the protocol interface exports a function for each protocol role, called a *role function*, that implements the corresponding role process by performing all session-related sends and receives for a given role.

The first argument of a role function contains information about the running principal (IP address and asymmetric keys). Using a continuation-passing style, the second argument allows the application to bind variables and choose which message is to be sent at each state. For example, the role function for w in the Ws session is declared by:

```
val w : principal → m0 → result_w
```

where result_w is the return type of role w and the type m0 encodes the role process for w as:

```
type m0 = {hRequest: (var_c ∗ var_w ∗ var_q → m1)
and m1 = Reply of (var_x ∗ result_w) | Fault of result_w
```

where the var_$\{c, w, q, x\}$ are the types of the variables. Hence, m0 defines a single message handler for the Request message; the handler takes the received values (c, w, q) as input and computes a response of type m1 that may either be a Reply or a Fault.

In addition to enforcing a role process, role functions have two features. They log security events before sending and after receiving each message. We use these events to formalize our security goals (see section 4). For example, before sending the Reply message, the web server must log an event, Send_Reply, that states that it intends to send a reply with some specific value for x. Second, they are locally sequential; in particular, they never send messages in both branches of a session.

Typechecking By writing well-typed (for ordinary ML typing) application code using the protocol module, a programmer can guarantee that his code is locally compliant with the session. However, compromised principals playing remote roles may not comply with the session and may collude with a network-based adversary to confuse compliant principals. Hence, the protocol module uses cryptography to ensure global session integrity—all compliant principals have consistent states (Section 4).

To verify that the generated cryptographic protocol code meets this security goal, we typecheck it against an interface (Ws_protocol.ml7) that encodes the session graph in terms of logical pre- and post-conditions on the protocol functions that send and receive session messages. This interface uses an ML syntax extended with refinement types that allow the embedding of formulas in a first-order logic; a specialized typechecker verifies these formulas by calling out to an automated theorem prover [Bengtson et al., 2008].

4. Session integrity

In this section, we formalize our main security theorem for protocol implementations generated by our compiler after they have been verified by typechecking. The theorem is stated in terms of the events emitted by the implementation for each message sent and received during session execution: independent of the behaviour of compromised principals, the events of the compliant ones always correspond to a correct trace of the session. An $\widetilde{S}$-system is a program composed of the ML modules:

$$\mathit{Data}, \mathit{Net}, \mathit{Crypto}, \mathit{Prins}, (S_i\_\mathit{protocol})^{i=1..k}, U$$

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 5);
- S_i_protocol is the verified module generated by our compiler from the session description S_i and then typechecked against its refined typed interfaces and those of the libraries, for $i = 1, \ldots, k$.
- U represents application code, as well as the adversary. It ranges over arbitrary ML code with access to all functions in Data, Net, Crypto, and S_i_protocol, and access to some keys in Prins (as detailed below).

The module U has the usual capabilities of a Dolev-Yao adversary: it controls compromised principals that may instantiate any of the roles in a session; it may intercept, modify, and send messages on public channels, and perform cryptographic computations, but it cannot break cryptography, guess secrets belonging to compliant principals, or tamper with communications on private channels.

A run of an $\widetilde{S}$-system consists of the events logged during execution. For each session, we define three kinds of events that are observable:

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

where f ranges over labels in the session. Send_f asserts that, in some run of the session, principal a instantiating the source role of f commits to sending a message labelled f with values $\widetilde{v}$ for its written variables. Recv_f asserts that principal a instantiating the target role of 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; this event is generated whenever the adversary U demands 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. (This functionality of Prins formally models selective key compromise; it is of course disabled in our concrete implementation.) For a given run of an $\widetilde{S}$-system, 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.

We now relate events and session graphs: a *session trace* 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 (Session traces): *The traces 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 α_i and α_{i+1} may differ only on $\widetilde{x}_i$, for $i = 1..k-1$;*

4) *replace each f_i from the path with two events*

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

5) *optionally discard the final $Recv\_f_k$ event.*

For a given run of an $\widetilde{S}$-system, a compliant trace of a session $S \in \widetilde{S}$ is a projection of a trace of S where non-compliant events are discarded. Session traces capture all sequences of events for a partial run of an S-system. Moreover, the values of the variables recorded in the events are related to one another other exactly in accordance with the variable (re)writes allowed by the graph (possibly shadowing each other).

We are now ready to state our main security result:

Theorem 1 (Session Integrity): *For any run of an $\widetilde{S}$-system, there is a partition of the compliant events coinciding with compliant traces of sessions from $\widetilde{S}$.*

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. Hence, all principals who use protocol implementations ($S_i\_protocol$) generated by our compiler and verified by typechecking are protected against adversaries U.

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

5. Protocol design and cryptography

For each session, our compiler generates a custom cryptographic protocol by first extracting the control flow graph for each role, and then selecting cryptographic protection for each message according to this control flow.

Internal control flow states In the session graphs of Section 2, each path represents a different session execution, with potentially different subsets of active roles and assigned variables. Thus, for a given node, the messages that the role may send or receive and their cryptographic protections may depend on the path followed so far. For example, in the Proxy session of Figure 1(c), the middle node for role w represents two different runtime states for w: one when an Audit message is received for the first time; another for subsequent Retry messages.

To precisely capture the runtime states for each role, we rely on the fact that the content of a given message to be sent is determined by the position of each of the roles in the session graph and their current knowledge of the variables' contents. Each of these states can thus be indexed by a role name (the sender) and an extended path containing the last label sent by each of the roles and the last occurrence of each written variable. We call *internal control flow states* these paths, indexed by ρ, and give a formal definition below.

The states ρ form a refined graph of the original session graph. The refinement roughly duplicates every node within a loop to distinguish the case when the loop is entered for the first time from subsequent iterations. Hence, the refined graph can contain more nodes and edges than the session graph, but it remains finite. The states ρ can thus serve as indices for the various receiving and sending functions in the generated protocol module (e.g. Section 3).

Definition 2: *An* internal control flow state*, denoted ρ, is an extended path that is in the image of the state function* st *(defined below) applied to some initial path. Let $\mathsf{st}(\widetilde{f}) = \widetilde{f} \setminus (\varepsilon, \varepsilon)$ where the filter function $\setminus$ is defined as*

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

where $r = \mathsf{src}(f)$ and $\widetilde{x}' = \mathsf{write}(f) \setminus \widetilde{z}$.

Intuitively, $\widetilde{f} \setminus (\widetilde{z}, \widetilde{r})$ sweeps through $\widetilde{f}$, right to left, filtering out any labels sent by roles encountered to the right (accumulated in $\widetilde{r}$) and any written variables (accumulated in $\widetilde{z}$).

The refined graph can be projected to give the execution states of a given role (in the same way that roles processes are projection of the global session graph). For example, Figure 3 outlines the refined graph of the Proxy session projected for role w. We obtain the projection by keeping the nodes in which role w is receiving or sending a message, and by replacing the subgraphs that involve communications between the other roles with "dotted" nodes (whose annotations are explained below).

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.

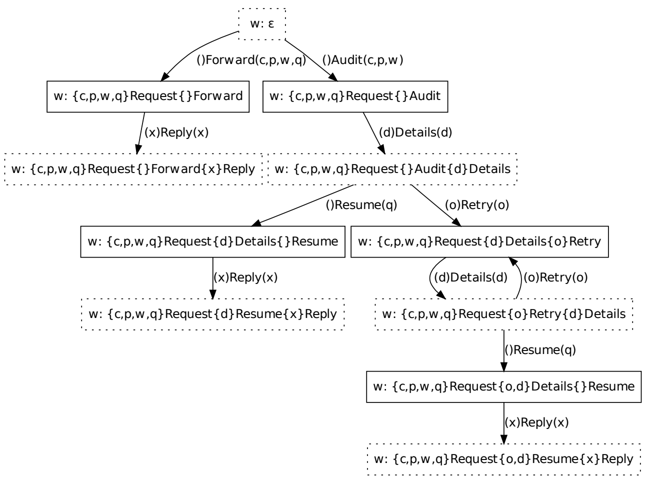
w: ε
()Forward(c,p,w,q)
()Audit(c,p,w)
w: {c,p,w,q}Request{}Forward
w: {c,p,w,q}Request{}Audit
(x)Reply(x)
(d)Details(d)
w: {c,p,w,q}Request{}Forward{x}Reply
w: {c,p,w,q}Request{}Audit{d}Details
()Resume(q)
(o)Retry(o)
w: {c,p,w,q}Request{d}Details{}Resume
w: {c,p,w,q}Request{d}Details{o}Retry
(x)Reply(x)
(d)Details(d)
(o)Retry(o)
w: {c,p,w,q}Request{d}Resume{x}Reply
w: {c,p,w,q}Request{o}Retry{d}Details
()Resume(q)
w: {c,p,w,q}Request{o,d}Details{}Resume
(x)Reply(x)
w: {c,p,w,q}Request{o,d}Resume{x}Reply

Figure 3. Projected refined graph for role w in Example 1(c)

In the case w sends a Details message writing variable d, it may next jump to one of two different states, depending on whether a Resume message arrives or a Retry message arrives. In the latter state, w needs to send another Details message re-writing d, which may result in either another Retry or a final Resume message, to which w answers finally with a Reply message writing variable x.

The "dotted" nodes represent states of the protocol implementation of a role where this role is inactive (e.g. waiting for a message). Each of these nodes can be uniquely designated by the internal control flow state of the initial node of the abstracted subgraph.

We use the notation $\rho \lhd \rho'$ between internal control flow states to express the existence of an edge in a projected refined graph between a "dotted" node ρ and a regular node ρ' (see section 6 and appendix D).

Cryptographic protection We now turn our attention to the (cryptographic) protection of messages exchanged by roles. An edge $(\widetilde{x})f(\widetilde{y})$ with target role w maps to a single low-level network message embedding values for $\widetilde{y}$, plus the tag f (as well as additional auxiliary information, detailed below). We protect the confidentiality of $\widetilde{y}$, provide access of $\widetilde{y}$ to w, and provide integrity that gives evidence to w that the message comes from the source role of f. This protection uses session keys which are initially established between roles via a key-establishment protocol relying on public-key cryptography. We provide integrity by applying a message authentication code (MAC) to each sent label, including the store variables and their values (they are hashed together and, for the variables that should be kept secret from the destination role, salted with an extra confounder; the hash is also communicated outside the MAC to provide variable commitment for the target role, and hence the confounder is needed to prevent dictionary attacks). For confidentiality, variable values are (symmetrically) encrypted.

As opposed to a standard wire protection where one uses e.g. encrypt-then-MAC, here we use MACs only for the hashed variables, and encrypt separately; this design allows to selectively encrypt only the needed variables, and allowing the sending in the clear of other public ones. The integrity of the received variables (encrypted or not) are checked by reconstructing the hashes and then checking the MAC.

Low-level messages include also auxiliary data, since in order to ensure path integrity, our protocol forwards evidence down the flow. For example, in the Proxy example, when w receives an Audit message, 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).

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. As an example, we detail the structure for the initial message of the Proxy session.

Example message: Proxy Request Consider the first receipt of a Request message by p from c in a run 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. Let s be the session identifier, c, p, w be the principals instantiating roles c,p,w; and let q be the value for q. Moreover, for each variable, let v_c be a fresh confounder for the session run s. The format of the message sent by c to p is as follows:

$$\begin{aligned}
&s \parallel 0 \parallel \ell_0 && (1)\\
&\parallel mac_{c,p}(s \parallel 0 \parallel \ell_0 \parallel h[c] \parallel h[p] \parallel h[w] \parallel h[q]) && (2)\\
&\parallel c \parallel c_c \parallel p \parallel p_c \parallel w \parallel w_c \parallel enc_{c,p}(q \parallel q_c) && (3)\\
&\parallel asign_c(aenc_p(\text{‘}c\text{’} \parallel k^a(c,p) \parallel k^e(c,p))) && (4)\\
&\parallel mac_{c,w}(s \parallel 0 \parallel \ell_0 \parallel h[c] \parallel h[p] \parallel h[w] \parallel h[q]) && (5)\\
&\parallel asign_c(aenc_w(\text{‘}c\text{’} \parallel k^a(c,w) \parallel k^e(c,w))) && (6)
\end{aligned}$$

where $\parallel$ 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 $h[v] = h(s \parallel$

sr ‖ 'v' ‖ v ‖ v_c) to denote the hash of the (fresh) session identifier s, concatenated with the source role principal sr, concatenated with the variable tag 'v', its current value v and (when needed, see above) confounder v_c; 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)-(4) constitute information computed by c intended for p, while components (5)-(6) are for w, and will be forwarded by p to w in subsequent messages. In (4), component $asign_c(aenc_p(\text{'}c\text{'}, k^a(c,p) \parallel k^e(c,p)))$ is added for key establishment, necessary 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 below, we assume a public key infrastructure in place). Once session keys are exchanged, (3) 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. On the other hand, variable q is a data variable and so we must protect it so that it remains confidential; it is encrypted using the shared symmetric key $k^e(cp)$.

Integrity is achieved by the MAC in (2), where all the variable hashes are concatenated, preppended with a session identifier $s = h(\Sigma, r)$ 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, (5) contains a similar MAC from c but intended for w, and session keys between c and w are included in (6). A hash of q has to be included in the Audit message from p to w so that w can check (1) the forwarded MAC from c and (2) the commitment to q when w eventually learns its value in Resume.

Upon receiving the Request message, the principal running as p processes (4) to obtain the session keys. These keys are used to process the variables in (3); once all variables are processed, p recomputes the hashes and checks the MAC of (2). The principal variables are checked to see if the principal is indeed assigned to its, and that the s is not a replay. If the checks succeed, role p updates the session store and invokes the user code handler.

Generated protocol module Our compiler generates a protocol module with functions that send and receive messages like the one above. The detailed structure of the generated code is shown in Appendix C.

For each role, the generated send and receive functions operate on a data store (of type store) that contains all the session parameters, such as the session identifier, the current timestamp, and the values of all the variables known to the role. In addition, the store contains the hashes of all the variables bound in the session so far, including the ones whose values are not known locally. Finally, the store contains cryptographic materials for the session, such as established session keys, received MACs, and confounders for hashes, as described later in this section. During any run of the session, the current internal control flow state in combination with the value of the current data store describes the full state of each role in the session.

Our code relies on a set of trusted libraries for data manipulation, cryptography, networking, and managing principals. These libraries (and their refined types) are essential parts of our security model.

Trusted libraries 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 $\rightarrow$ str for encoding string payloads, and concat: bytes $\rightarrow$ bytes $\rightarrow$ bytes for assembling messages.

The $Crypto$ library provides functions for encryption (RSA and AES), MACing (HMACSHA1), hashing (SHA1), and generating fresh nonces and keys.

The $Prins$ library maintains a database of principals. The database records, for every other principal, the principal name, its public-key certificate, its network address, and any session keys shared with it. 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 locally 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, it generates fresh session keys and registers them in the database.

Relying on the networking library, Net, $Prins$ also provides functions for sending and receiving messages between principals; Net is never accessed directly by our generated code, but may be accessed by application code for non-session communications.

6. 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.

In this section we proceed as follows: we first enrich send and receive events with additional parameters and lift the definition of session traces accordingly; for each session, we define families of predicates that capture invariants that must be maintained by a session implementation (Figure 5 in Appendix D); 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 (via Lemma 3) the integrity theorem for all code that is generated by our compiler and typechecked (Theorem 1).

Stores, timestamps, and enriched events For simplicity, we decouple the notion of a "store" s (Section 3) from its fields consisting of a session identifier, herein written s, and a variable store, written σ. We treat σ as a triple of partial maps $\sigma_{\mathsf{v}}, \sigma_{\mathsf{h}}, \sigma_{\mathsf{c}}$, where σ_{v} maps variables to (their known) values, σ_{h} maps variables to hashes, σ_{c} maps variables to confounders. The store σ is *consistent* for variables $\widetilde{z}$ with 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 in Appendix D).

Enriched events are obtained from those of Section 4 by adding timestamps and stores (following the implementation in Section 5):

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

Timestamps, ranged over by ts, 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 can lift Definition 1 accordingly to specify the traces of enriched events, as shown in Appendix D.

Invariant path predicates Figure 5 in Appendix D defines a pair of families of predicates, Q and Q', that serves as the invariant at each send and receive event emitted by the generated implementation code for sessions. The invariant reflects the full complexity of our optimized protocol and is established by typechecking.

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 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 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.

The definition of Q' in Figure 5 in Appendix D relies on a formula abbreviation $\lfloor C \rfloor_a$ that stands for the disjunction $C \vee \text{Leak}(a)$, that is, either C holds or the principal a is compromised. The definition of Q relies on the notion of an internal control flow state ρ *preceding* an internal control flow state ρ', written $\rho \lhd \rho'$, which informally means that an initial path leading to ρ may be extended by a suffix to ρ'. More generally, we can extend this binary notion to vectors of internal control flow states (used in Q'), writing $(\rho_0, \rho_1, \ldots, \rho_k)_{\lhd}$. We present these defintions formally in Appendix D.

Expanding the Q and Q' predicates at a particular internal control flow state yields a tree of disjunctions and conjunctions of assertions that when combined, characterizes the valid traces of send and receive events at the compliant principals in a trace of the session (Definition 3).

For example, in the Ws session introduced in Figure 1(a), after a Request is received, the internal control flow state at role w is (cwq)Request, and the predicate $Q'_{\text{\_(cwq)Request}}(s, 0, 1, \sigma')$ implies the assertions:

$\text{Recv}_{\text{\_Request}}(\sigma'_{\mathsf{v}}(\text{w}), s, 0, 1, \sigma'_{\mathsf{v}}(\text{c}, \text{w}, \text{q}), \sigma')$,
$H_{\text{c,w,q}}(\sigma', s)$, and
either $\text{Leak}(\sigma'_{\mathsf{v}}(\text{c}))$
or $\text{Send}_{\text{\_Request}}(\sigma'_{\mathsf{v}}(\text{c}), s, 1, \sigma'_{\mathsf{v}}(\text{c}, \text{w}, \text{q}), \sigma'')$.

That is, the receive event must have been logged, the hashes and value of variables in the store must be consistent, and either both c and w agree on the message (Request) and variable assignments (c, w, q) so far, or c has been compromised.

After sending the subsequent Reply message, the internal control flow state at w is (cwq)Request(x)Reply, and the predicate $Q\_{\text{(cwq)Request(x)Reply}}(s, 2, \sigma)$ implies:

$\text{Send\_}_{\text{Reply}}(\sigma_{\mathsf{v}}(\text{w}), s, 2, \sigma_{\mathsf{v}}(\text{x}), \sigma)$,
$H_{\text{c,w,q,x}}(\sigma, s)$,
$Q'\_{\text{(cwq)Request}}(s, 0, 1, \sigma')$, and
$\Delta_{\text{x}}(\sigma'_{\mathsf{h}}, \sigma_{\mathsf{h}})$.

That is, the send event must have been logged, the current store σ must be consistent, and the receive predicate $Q'\_{\text{(cwq)Request}}(s, 0, 1, \sigma')$ must have previously held at w with some store σ', where the hashes in σ' and σ only differ for x (because it is the only variable written in this step).

Proofs by typing Our compiler generates an extended type interface that uses path predicates as pre- and post-conditions for the session messaging functions. For example, for the Ws session, the generated protocol module contains a role function w that calls the messaging functions recv_w_Request and send_w_Reply. The extended type interface for these functions is of the form:

val recv_w_Request: (s:store)$\{Q\_{\varepsilon}$(s.sid,0,0,s)$\}$ $\rightarrow$
(s':store)$\{Q'\_{\text{(cwq)Request}}$(s'.sid,1,s')$\}$
val send_w_Reply: (x:int) $\rightarrow$
(s:store)$\{Q'\_{\text{(cwq)Request}}$(s.sid,1,s)$\}$ $\rightarrow$
(s':store)$\{Q\_{\text{(cwq)Request(x)Reply}}$(s'.sid,2,s')$\}$

The curly braces after (s:store) enclose a logical formula that must hold about the store s. In general, formulas that appear in the type of function arguments represent pre-conditions; formulas in the their result type represent post-conditions. The pre-condition on recv_w_Request says that, initially, the store s must be empty; its post-condition is the Q' predicate on w's store at the internal control flow state (cwq)Request. The pre-condition on send_w_Reply is the same as the post-condition of recv_w_Request; hence, a Reply may be sent only after a Request is received; its post-condition is the Q predicate on w's store at the internal control flow state (cwq)Request(x).

Typechecking a program against an extended interface guarantees that in every execution of the system with an active adversary, whenever a function is called, its precondition holds. Hence, by typechecking we establish that the path predicates are maintained as an invariant by the generated protocol module.

Lemma 1: *For any run of an $\widetilde{S}$-system, for any session $S \in \widetilde{S}$, for any session identifier s running S, for any compliant principal a,*

- *the event 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 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 $a = \sigma_{\mathsf{v}}(\mathtt{tgt}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{y}$, and $Q'\_\rho(s, ts', ts, \sigma)$ where $\widetilde{y}$ are the read variables of f;*

During the design of our compiler, we found several bugs (violations of Lemma 1) 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 7.

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

Lemma 2: *In any run of an $\widetilde{S}$-system, 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 an alternation of sends and receive events such that for any adjacent pair of such events*

Send_$f(a, s, ts_0, \widetilde{v}, \sigma_0)$, Recv_$g(a, s, ts_1, ts_2, \widetilde{w}, \sigma_2)$ or
Recv_$g(a, s, ts_1, ts_2, \widetilde{w}, \sigma_2)$, 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 of the generated role functions in the protocol module.

We then show from Lemma 2 that the predicates Q and Q' imply that the events constitute session traces from which Theorem 1 follows.

Lemma 3: *For every run, if $Q\_\rho(s, ts, \sigma)$ or $Q'\_\rho(s, ts', ts, \sigma)$ holds, then there is a session trace of an initial path ending in state ρ that matches a subsequence of the compliants events.*

Secrecy This paper focuses on integrity rather than secrecy, whose formulation is more technical (as it involves the behaviour of user code, not just protocol code). We only outline our secrecy results.

By typechecking, we 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. (Bengtson et al. [2008] also provide a discussion of secrecy by typing.) The value assigned to a variable in a session run may be obtained by the adversary only if a compromised

Session S	Roles	S.session (lines)	Application code (lines)	Graph (.dot lines)	Refined Graph (.dot lines)	S_protocol.ml (lines)	S_protocol.ml7 (lines)	Verification (seconds)
Ws (Figure 1a)	2	8	33	14	24	592	414	8.8
Rpc	2	15	24	11	18	472	315	6.1
Commit	2	16	29	14	24	603	399	10.3
Wsn (Figure 1b)	2	10	44	17	48	1143	813	23.6
Fwd	3	15	38	11	19	581	357	8.6
Proxy (Figure 1c)	3	28	65	26	80	2181	1939	154.1
Login	4	28	54	29	74	2053	1542	103.4

Figure 4. File sizes and verification times for example sessions

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.

7. Performance evaluation

We finally present compilation and verification results for a series of examples. In addition to the sessions of Figure 1, it includes a simple remote call (Rpc); a session with early commitment to values; a session with message forwarding (Fwd); and a 4-ary session between a client, a gateway, a database, and a late-bound web server (Login). For each session, we experimentally confirmed that the generated protocol is functional, using simple testing.

Our compiler is written in around 4300 lines of ML. The trusted libraries for networking, cryptographic primitives, and principals shared by all session implementations have 780 lines of code (although their concrete implementation mostly relies on much-larger system libraries).

For each session example (S), Figure 4 first gives the numbers of roles; the size of the input file (S.session); the size of the handwritten application code we used for testing the session; the size of the session graph generated by our compiler, both before and after refining the graph to separate different internal control flow states; the size of the output files, both for the ML implementation (S_protocol.ml) and for its refinement-typed interface (S_protocol.ml7); and finally the time spent typechecking our generated implementation against refinement-typed interfaces at the end of the compilation process. (The rest of the compilation is relatively fast.)

Even when programming complex multi-party sessions with many roles and messages, the application programmer only needs to write less than one hundred lines of code. The generated modules are larger and more complicated—in fact the auxiliary formulas that annotate their function declaration are as large as their actual code. However, the programmer can ignore their details and rely instead on the typechecker. The verification time is roughly linear in the size of the generated code. (Pragmatically, the programmer may gain additional confidence in the verification process by reviewing just the location and content of the security events in generated code, which involve only a small part of that code.)

We also measured the overhead of cryptographic protection for three simple sessions. 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. We used a Pentium 3GHz with 1G RAM, running Windows XP with .NET cryptography, and only local communications. The first line is for variants of the generated protocols that do not perform any cryptographic operations (with no protection). The second line is for the same variants, plus transmitting all messages on TLS connections (with transport protection). The third line is for the unmodified protocol (with session integrity).

Cryptography	Wsn	Ws	Proxy
none	1.81	2.37	9.43
TLS	2.58 (+29%)	3.01 (+21%)	n/a
our protocol	3.48 (+47%)	3.94 (+39%)	15.48 (+39%)

The cryptographic overhead is thus around 40%; however, the benchmarks are done for a single machine; for real distributed settings, we expect this overhead to be often negligible in the face of networking overheads. When compared with running sessions over .NET's SSL layer (second row), our protocol is about 18% slower, but it offers stronger cryptographic protection. (We did not measure SSL for multi-party sessions since SSL only protects two-party communications).

Acknowledgments This work benefited from comments from Jérémy Planul and Nobuko Yoshida.

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.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

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.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335. ACM Press, Jan. 2008.

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.

Appendix A. Well-formed conditions for global sessions

We list all the well-formedness properties that we require of session graphs. 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. They use some additional notation and terminology.

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 $x \in \mathsf{knows}(r, \widetilde{f})$ then x is written on f.

Appendix B. 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 $[\{\|f_i(\widetilde{x_i}) \to p_i\}_{i<k}]$	receive
$\chi : p$	named subprocess
χ	continue with χ
0	end
$\Sigma ::=$	Sessions
(var $x_j : \tau_j)_{j<m}$ (role $r_i{:}\tau_i = p_i)_{i<n}$	

Appendix C. 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 specifying 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 roles with (encrypted, session-establishment) keys that need to be forwarded. As shown in the code below, *fwd_keys* is used to lookup (if it already exists) or generate a new shared key, using function *gen_keys*.

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 $\leftarrow$ to designate a store with an updated field.

```
type store = {
  vars : { for each $(x{:}t) \in \mathcal{X}$, [ $x$: $t$ ]} ;
  hashes : { for each $x \in \mathcal{X}$, [ h$x$ : hashstore ] } ;
    (* hashstore has hashes and confounders *)
  macs : { for all $l$ with $\widetilde{x}$ visible
        received by $r$, [ $rl\widetilde{x}$ : 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 5).

```
For all state $\rho$ in a visible sequence with $\widetilde{z}$
[let content_$\rho$_$\widetilde{z}$ = fun ts store $\to$
 fold over $z \in \widetilde{z}$
 [let hashes = concat store.hashes.h$z$.hash hashes in]
  let state = utf8 (cS "$\rho$") in
  let payload = concat state hashes in
  let header = concat store.header.sid
                 (utf8 (cS (string_of_int ts))) in
  concat header payload]
```

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 $\rho \xrightarrow{(\widetilde{x})\ f\ (\widetilde{y})} \rho'$ of the refined graph, sending role $r$
$r'$ is the receiving role.
[let sendWired_$f$_$\rho$ (s:store) = fun $\widetilde{x}$ s $\to$
  fold over $x \in \widetilde{x}$ [ s.vars.$x \leftarrow x$ ; s.hashes.h$x$ $\leftarrow$
 sha1 x ;]
   s.header.ts $\leftarrow$ s.header.ts + 1;
  fold over $r, r' \in$ fwd_key$(\rho)$
   [let key$rr'$ = gen_keys s.vars.$r$ s.vars.$r'$ in
    let keys = concat key$rr'$ keys in]
  for all $(r'', \rho'', \widetilde{z}) \in$ future$(\rho, l)$
 (header is built as in content)
   [let content = content_$\rho''$_$\widetilde{z}$ s.header.ts s in
    let mackey$rr''$ = get_mackey s.vars.$r$ s.vars.$r''$ in
    let macmsg = mac mackey$rr''$ (pickle content) in
    let $r'f\widetilde{x}$ = concat header macmsg in]
   ... Marshalling sent MACs (fwd_macs) ...
   ... Marshalling hashes (fwd_hashes) ...
  fold over $y \in \widetilde{y}$
  [let key$rr'$ = get_symkey s.vars.$r$ s.vars.$r'$ in
   let encr_$y$ = sym_encrypt key$rr'$ (pickle mar_$y$) in
   let variables = concat encr_$y$ variables in]
   ... marshalling of confounders for variables $\widetilde{y}$ ...
   ... Building header and message ...
   let () = psend s.vars.$r'$ msg in
   s]
```

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 $\rho$,
  [type wired_$\rho$ =
     for each $f$ that can be received in state $\rho$
       [ | Wired_$f$_$\rho$ 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),

Meta predicates: Consistency of stored hashes $H_{\widetilde{z}}(\sigma, s) \triangleq \bigwedge_{x \in \widetilde{z}} \sigma_{\mathsf{h}} x = h(s \parallel `x\text{'} \parallel \sigma_{\mathsf{v}} x \parallel \sigma_{\mathsf{c}} x)$
Store updates $\Delta_{\widetilde{z}}(\sigma, \sigma') \triangleq \bigwedge_{x \in \mathcal{X} \setminus \widetilde{z}} \sigma(x) = \sigma'(x)$
Up to compromise $\lfloor C \rfloor_a \triangleq C \vee \mathrm{Leak}(a)$

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:

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

Inductive case for Q': For every non-empty internal control flow state $\rho_k = \hat{f}\,(\widetilde{x}_1)f_1 \dots (\widetilde{x}_k)f_k$ for which $\mathsf{tgt}(f_k)$ is not active on $f_1 \dots f_k$ 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 $\widetilde{z} = \mathsf{write}(\rho_k) \setminus (\widetilde{x}_1 \dots \widetilde{x}_k \cup \{\mathsf{src}(f_1), \dots \mathsf{src}(f_k)\})$, we let:

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

Figure 5. Definition of predicate families Q and Q'

checks MACs (visib), and finally returns the corresponding Wired data type.

Appendix D. Definitions used in Section 6

Helper functions 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 $\widetilde{f}$:

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

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 $\mathsf{knows}(c, \text{Request Reply}) = \{c, w, q, x\}$ and $\mathsf{knows}(c, \text{Request Fault}) = \{c, w, q\}$.

Definition 3 (Session traces with enriched events): *The traces 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: $\Delta_{\widetilde{x}_{i+1}}(\sigma_{\mathsf{h}i}, \sigma_{\mathsf{h}i+1})$ for $i = 1..k-1$ (see Figure 5 in Appendix D 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 f_i in the path with two events*
   $$Send\_f_i(\sigma_{\mathsf{v}i}(\mathsf{src}(f_i)), s, ts_i, \sigma_{\mathsf{v}i}\widetilde{x}_i, \sigma_i),$$
   $$Recv\_f_i(\sigma'_{\mathsf{v}i}(\mathsf{tgt}(f_i)), s, ts'_i, ts_i, \sigma'_{\mathsf{v}i}\widetilde{y}_i, \sigma'_i)$$
6) *optionally discard the final $Recv\_f_k$ event.*

Relations on internal control flow states Formally, $\rho \lhd \rho'$ 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 $(\rho_0, \rho_1, \dots, \rho_k)_\lhd$ iff there exists an initial path $\widetilde{f}_0 \widetilde{f}_1 f_1 \dots \widetilde{f}_k f_k$ such that the active roles of $\widetilde{f}_i$ are included in the active roles of $f_i, \dots, f_k$ for $i = 1..k$; $\rho_i = \mathsf{st}(\widetilde{f}_0 \widetilde{f}_1 f_1 \dots \widetilde{f}_i f_i)$ for $i = 1..k$; $\rho_0 = \mathsf{st}(\widetilde{f}_0)$. In this definition, $f_1, \dots, f_k$ 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 $f_i \dots f_k$. Therefore, each ρ_i is the internal control flow state for the last message send performed by role $\mathsf{src}(f_i)$, for $i = 1..k$.

Predicates

These appear in Figure 5.