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, 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. For 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 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 received messages.

From sessions to cryptographic protocols Whereas much work has been done on session types and expressiveness, we believe that ours are the first to protect session execution integrity via the systematic generation of cryptographic protocols. We implement a compiler from the session language to custom 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 refinement 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 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 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 implementations for multiparty cryptographic protocols, and our generated protocols are the largest to date with verified implementations.

Related work We build upon earlier work [Corin et al., 2008] which explores the secure cryptographic
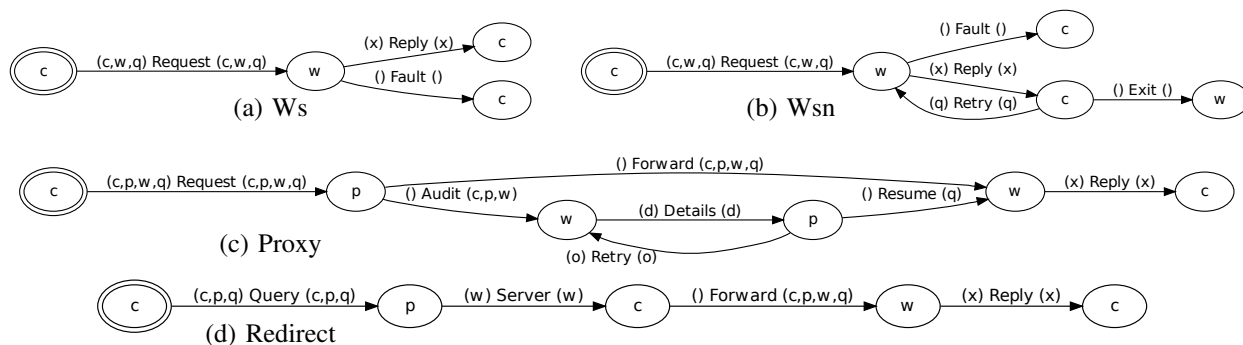
Figure 1. Sample sessions: (a) Ws: a single query; (b) Wsn: an iterated query; (c) Proxy: a three-party negotiation; (d) Redirect: a query forwarded to a dynamically selected web server.

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 session participants); 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 automated proofs (we used manual proofs on a simplified model).

Session types Honda, Vasconcelos, and Kubo [1998], Gay and Hole [1999], Dezani-Ciancaglini et al. [2005, 2006], Vasconcelos et al. [2006], Bonelli and Compagnoni [2007], and Honda, Yoshida, and Carbone [2008] consider type systems 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] globally specify security protocol narrations and show functional (but not security) aspects of their projection to local roles [like in Honda et al., 2008]. Guha et al. [2009] infer sessions types from existing Javascript applications.

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] 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 [Bhargavan et al., 2009] and the code for all examples and libraries, are available online at http://msr-inria.inria.fr/projects/sec/sessions.

2. Multiparty sessions

We introduce sessions by illustrating their graphical representation and describing their features and design choices. Sessions are subject to well-formedness properties (listed in Appendix A), which we also discuss by example. We then give a formal definition of sessions as graphs that obey all well-formedness properties.

Figure 1 represents our four 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 distinct 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 sessions is described in Section 3.

126

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 of Figure 1 extends the graph with a cycle. On receiving a Request or an Retry, 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-Retry loop any number of times before the client chooses to Exit the session or receives a Fault.

The session Proxy of Figure 1 introduces a third role, p for "proxy", that intercedes between the client and the web service and chooses between two branches of the session upon receiving a Request from the client.

In the shorter upper branch, the proxy sends a Forward message to the web service, which gives a Reply to the client.

In the lower branch, the proxy instead sends an Audit message, indicating that further processing is needed before accepting the request. The web service and proxy then loop 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.

If a compromised principal controls a role that chooses between branches, then it may send messages along multiple branches, hence violating the invariant that each session execution has a single token. To detect such violations, we exclude graphs with branches where the set of roles along one branch is not included in the set of roles along the other. This restriction is formally stated as the well-formedness property 4 in Appendix A.

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 session Proxy, 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.

Variables may be hidden from 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 locally read on an edge need not coincide, and all readers are guaranteed to get the same values unless the variable is explicitly rewritten. However, on each edge, the source role must know all the locally read variables, so that it can forward them to the target.

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 session Proxy, the principal acting as client initially assigns principals to all three role variables c, p, and w, writing these in the Request message. (To form this message, the principal must use credentials associated with its claimed identity c.)

In the general case, the first message need not write all the role variables; instead, the principals already involved in a session may negotiate who should fill its remaining open roles. Well-formedness properties on sessions ensure that all principals agree on role assignments: role variables are written only once, and must have been read by all principals already in the session before sending any message to a new participant.

In session Redirect of Figure 1, for instance, the first message assigns c and p but not w, the web server role. Based on the client's query q, the proxy p, acting as a discovery service, can dynamically select a server w and redirects c to continue its session with w. The client can then either contact w, or drop the session if it does not trust w. On the other hand, our compiler would reject a session such that p directly forwards the request to any server of its choice. (In that case, the graph would still indicate that a single server is passed the client request, whereas a compromised proxy may pass the request to multiple servers and then choose which reply it prefers.)

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 $(v_0 \ldots v_k)$. 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, 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 $(\widetilde{x}_0)f_0 \ldots (\widetilde{x}_k)f_k$. We let $\hat{f}$ range over extended paths.

Appendix A formalizes well-formedness properties for session graphs. In the rest of this paper, we only consider graphs that satisfy these properties.

3. Programming with sessions

Figure 2 illustrates our framework for session Ws of Figure 1. 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 a refined type interface (Ws_protocol.ml7) also generated by the compiler (Section 6). 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
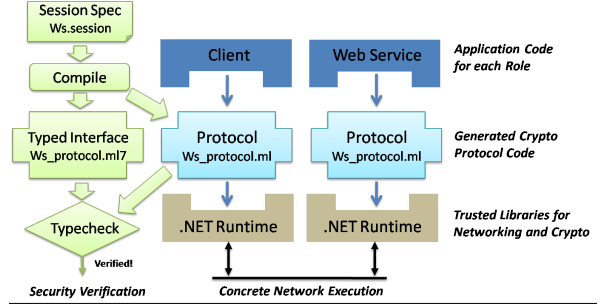

Figure 2. Compiling session programs

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 Ws 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, the four graphs depicted in Figure 1 were automatically generated from syntactic sessions during their 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 instance, the role function for the web service of session Ws is declared by:

val w : principal → m0 → result_w

where result_w is the server return type (here string) 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 var_[cwqx] are the types of the variables. Hence, m0 defines a single message handler for the Request message; the handler takes a triple of values for the read variables c, w, q as input and computes a response of type m1 that is either 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 application code that is well typed (under ordinary ML typing) against our generated protocol interface, a programmer is guaranteed that his code locally complies with the session discipline. 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, the compiler generates 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 all messages sent and received by compliant principals during session executions: irrespective of the behaviour of compromised principals, these events always correspond to correct session traces. A verified $\widetilde{S}$-system is a program composed of the ML modules:

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

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 exported by $Data$, 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 or guess secrets belonging to compliant principals.

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:

Send_$f(a, \widetilde{v})$ Recv_$f(a, \widetilde{v})$ 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 $\mathsf{Leak}(a)$ 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 $\mathsf{Leak}(a)$ 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 subtrace 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 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 a verified $\widetilde{S}$-system, there is a partition of the compliant events that coincides with compliant subtraces 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. We first discuss the design and structure of our generated protocols by example, relying on two informal protocol narrations for selected paths in the graph of Proxy in Figure 1. Later in this section, we describe the general case, the detailed message formats, and the compilation process.

Sample protocol narrations The first narration, in Figure 3, corresponds to the upper path of Proxy, which does not involve any Audit edges. The second narration, in Figure 4, corresponds to the lower path with a single Details-Retry iteration.

For simplicity, we first assume that every pair of principals a, b already shares keys for encrypting and MACing data using symmetric cryptography. We write $enc_b^a\{m\}$ and $mac_b^a\{m\}$ for the encryption and the MAC of m, respectively, using a key shared by a and b. As explained later in this section, the first time a needs keys for b, it generates these keys, encrypts and signs them using asymmetric cryptography, and includes the result in its outgoing message.

Each message consists of a header, a series of variables assignments (either in the clear, or hashed, or encrypted), and a series of MACs (with at most one MAC for every pair of principals of the session).

The message header consists of a session-label tag, a session identifier s, and a message sequence number $(0, 1, 2,\ldots)$. The tag is precomputed as a hash of the whole session definition plus the message label.

		Initially, c, p, w *share symmetric keys for mac and enc*	
	$c:$	*Assign* c, p, w, q	*Application at c assigns c,p,w,q*
	$c:$	*Fresh* s	*Fresh session identifier s*
(Request)	$c \to p:$	let $h_0 = \mathrm{Request}(s, 0)$ in	*Request header*
		let $m_0 = h_0 \mid Vars(c, p, w)$ in	*Plaintext message from c to p*
		let $a_0 = h_0 \mid Hashes(c, p, w, q)$ in	*Authenticated message from c to p, w*
		$m_0 \mid enc_p^c\{q\} \mid mac_p^c\{a_0\} \mid mac_w^c\{a_0\}$	*Formatted Request message*
(Forward)	$p \to w:$	let $h_1 = \mathrm{Forward}(s, 1)$ in	*Forward header*
		let $m_1 = h_1 \mid Vars(c, p, w)$ in	*Plaintext message from p to w*
		let $a_1 = h_1 \mid Hashes(c, p, w, q)$ in	*Authenticated message from p to w, c*
		$m_1 \mid enc_w^p\{q\} \mid mac_w^c\{a_0\} \mid mac_w^p\{a_1\} \mid mac_c^p\{a_1\}$	*Formatted Forward message*
	$w:$	*Assign* x	*Application at w assigns x*
(Reply)	$w \to c:$	let $h_2 = \mathrm{Reply}(s, 2)$ in	*Reply header*
		let $a_2 = h_2 \mid Hashes(c, p, w, q, x)$ in	*Authenticated message from w to c*
		$h_2 \mid enc_c^w\{x\} \mid mac_c^p\{a_1\} \mid mac_c^w\{a_2\}$	*Formatted Reply message*

Figure 3. Protocol narration for upper path of Proxy

The message payload includes the (possibly encrypted) value of each variable that can be read by the receiver. It also includes a cryptographic hash of the value of each variable that has been initially assigned or updated but cannot be read by the receiver. For instance,

- The first message (Request) carries an assignment of principals to c, p, and w (in the clear) and of a request to q (encrypted for p, which reads q).
- The second message (Forward) forwards the same assignment of principals to c, p, and w, and of the same request to q (now encrypted for w).
- The third message (Reply) carries an encrypted assignment to the response x; conversely, there is no need to re-send c, p, w, or q since they have not been assigned since c sent its last message.
- On the lower path in Figure 4, the Audit message does not carry the assignment to q, as w cannot read q yet. Instead, it carries the hash of the value of q, used by w as a commitment as it checks the received MAC.
  Later, the Resume message carries this assignment, as w is granted access to q. At this point, w can check that the received value matches the earlier commitment.

Next, we explain the purpose and contents of the MACs. In contrast with the message payloads, they authenticate the global state of the store, using an incremental hash computation, written *Hashes* here. Implicitly, every message recipient verifies every MAC for which it has the authentication key, by recomputing the presumed content from its local state plus the received assignments and hashes.

- The first message (Request) has two MACs, one for each of the other principals p and w, so that they can both verify that c initiates a session run as the client, with session identifier s and this particular assignment to c, p, w, and q.
- The second message (Forward) forwards c's MAC to w, and includes two new MACs from p, so that the two other principals can verify that p accepted c's request and assignment then selected the Forward message. (Without a MAC forwarded from c to w, for instance, a compromised p may involve c in a session by faking a request from c.)
- The third message (Reply) forwards p's MAC to c, and includes a new MAC from w. In combination, they enable c to verify that p accepted and forwarded its original request to w, and that w accepted and replied to the request with response x.
- On the lower path, each of the two MACs to w carried by the Audit message contains the hashes of c,p,w,q. To verify them, w recomputes the hashes of c,p,w from their values and uses the transmitted hash for q (since w does not read q at this stage).
  Later, the Details-Retry loop involves only p and w. Both have already checked a MAC from c and recorded its assignment, so a single MAC suffices in each message to authenticate the new assignment to d or o. At the same time, the inclusion of strictly-increasing sequence numbers in the MACS prevents any confusion between successive assignments to d.

Protection from replay attacks relies on caching of session identifiers, as follows. When receiving a message, each principal knows from the header whether the message is an invitation to join a session or a continuation for a session in which the principal is already participating. In the former case, the principal accepts the message only if it is not already involved in a session with the same session identifier and the same role; to this end, it maintains a local anti-replay

		Initially c, p, w *share symmetric keys for mac and enc*		
	c :	*Assign* c, p, w, q	*Application at* c *assigns* c,p,w,q	
	c :	*Fresh* s	*Fresh session identifier* s	
(Request)	c $\to$ p :	let $h_0 = \text{Request}(s, 0)$ in	*Request header*	
		let $m_0 = h_0 \,|\, Vars(c, p, w)$ in	*Plaintext message from* c *to* p	
		let $a_0 = h_0 \,|\, Hashes(c, p, w, q)$ in	*Authenticated message from* c *to* p, w	
		$m_0 \,|\, enc_p^c\{q\} \,|\, mac_p^c\{a_0\} \,|\, mac_w^c\{a_0\}$	*Formatted Request message*	
(Audit)	p $\to$ w :	let $h_1 = \text{Audit}(s, 1)$ in	*Audit header*	
		let $m_1 = h_1 \,|\, Vars(c, p, w) \,|\, Hashes(q)$ in	*Plaintext message from* p *to* w	
		let $a_1 = h_1 \,|\, Hashes(c, p, w, q)$ in	*Authenticated message from* p *to* w	
		$m_1 \,|\, mac_w^c\{a_0\} \,|\, mac_w^p\{a_1\}$	*Formatted Audit message*	
	w :	*Assign* d	*Application at* w *assigns* d	
(Details)	w $\to$ p :	let $h_2 = \text{Details}(s, 2)$ in	*Details header*	
		let $a_2 = h_2 \,|\, Hashes(c, p, w, q, d)$ in	*Authenticated message from* w *to* p	
		$h_2 \,|\, enc_c^w\{d\} \,|\, mac_p^w\{a_2\}$	*Formatted Details message*	
	p :	*Assign* o	*Application at* p *assigns* o	
(Retry)	p $\to$ w :	let $h_3 = \text{Retry}(s, 3)$ in	*Retry header*	
		let $a_3 = h_3 \,|\, Hashes(d, o)$ in	*Authenticated message from* p *to* w	
		$h_3 \,|\, enc_c^w\{o\} \,|\, mac_w^p\{a_3\}$	*Formatted Retry message*	
	w :	*Assign* d$'$	*Application at* w *re-assigns* d	
(Details)	w $\to$ p :	let $h_4 = \text{Details}(s, 4)$ in	*Details header*	
		let $a_4 = h_4 \,|\, Hashes(o, d')$ in	*Authenticated message from* w *to* p	
		$h_4 \,|\, enc_c^w\{d'\} \,|\, mac_p^w\{a_4\}$	*Formatted Details message*	
(Resume)	p $\to$ w :	let $h_5 = \text{Resume}(s, 5)$ in	*Resume header*	
		let $a_5 = h_5 \,|\, Hashes(d')$ in	*Authenticated message from* p *to* w	
		let $a_5' = h_5 \,|\, Hashes(c, p, w, q, o, d')$ in	*Authenticated message from* p *to* c	
		$h_5 \,|\, enc_w^p\{q\} \,|\, mac_w^p\{a_5\} \,|\, mac_c^p\{a_5'\}$	*Formatted Resume message*	
	w :	*Assign* x	*Application at* w *assigns* x	
(Reply)	w $\to$ c :	let $h_6 = \text{Reply}(s, 6)$ in	*Reply header*	
		let $m_6 = h_6 \,|\, Hashes(o, d')$ in	*Hashes for MAC verification*	
		let $a_6 = h_6 \,|\, Hashes(c, p, w, q, o, d', x)$ in	*Authenticated message from* w *to* c	
		$m_6 \,|\, enc_c^w\{x\} \,|\, mac_c^p\{a_5'\} \,|\, mac_c^w\{a_6\}$	*Formatted Reply message*	

Figure 4. Protocol narration for lower path of Proxy with one Details−Retry loop iteration

cache, and updates it whenever it joins a session. In the latter case, the principal knows the state of the session as it sent its last message, and updates this state after accepting the message, which prevents any replay (because it only ever accepts messages within the session with strictly increasing sequence numbers).

On the lower path, for instance, once w accepts a first Audit message, (1) it will never again accept any Forward message with the same session identifier, so a compromised p cannot trick it into running an "upper path" session by reusing c's MAC; (2) it will not accept any message for the session before sending Details; (3) it may then accept either a Retry or a Resume.

We now explain our general compilation scheme. As shown in the protocol narrations, each session path determines a particular sequence of valid messages, protected accordingly with cryptography. Thus, the compilation proceeds in two steps. First, it explores all possible execution paths by extracting the control flow graph for each role. Then, it protects cryptographically each message, depending on this control flow.

Obtaining control flow states The control flow graph of each role is used to determine its possible states during execution; a role in a particular state only accepts certain incoming messages but not others. 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 rely on an updated analysis of the session graph from the one presented in Corin et al. [2008]. The main idea is the same, namely that it is sufficient to check a signed *partial* history of the session's messages at each execution step. This notion, which we called *visibility* in our previous work, allows for an optimized protocol that only requires the transmission at each step of one MAC from each of the

roles that have been involved in the session since the receiver's last involvement. The significant difference is that freshly bound variables are now included in the MACs to account for the distributed store. This compact design allows for a finite statically-computed state-based implementation.

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 then serve as indices for the various receiving and sending functions in the generated protocol module.

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

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

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

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

Intuitively, $\mathsf{filter}(\widetilde{f}, \widetilde{z}, \widetilde{r})$ sweeps through an initial path $\widetilde{f}$, right to left, filtering out any labels sent by roles encountered to the right (accumulated in $\widetilde{r}$) and any written variables (accumulated in $\widetilde{z}$). Only the last occurrence of each variable and the last label sent by each role are kept. This way, an internal control flow state gives the position of each of the roles in the graph and, by the interleaving of variables and labels, the current knowledge of the store by each the roles.

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 5 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. In the case of an Audit, the reached internal control flow state is (c,p,w,q)Request()Audit, from which it is possible to deduce that w expects two MACs: a MAC of Request from c and a MAC of Audit from p. Each of those MACs would include the hashes of the variables bound since the last involvement of w,
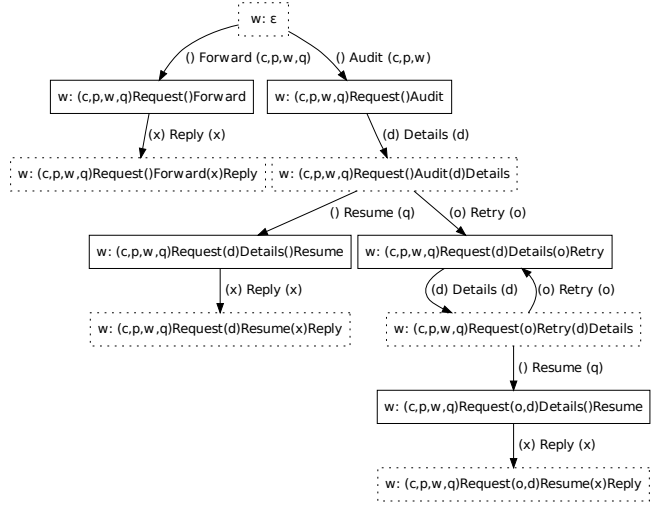

Figure 5. Refined graph for Proxy projected for w

namely (c,p,w,q). This is consistent with the protocol narration described above. When the next messages are Details and Resume, the internal control flow state becomes (c,p,w,q)Request(d)Details()Resume. Since Details was sent by w, only a MAC of Resume from p with the hash of d is expected. The internal control flow state also gives the information that all the roles know the value of (c,p,w,q), but only p and w know the value of d.

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.

Cryptographically protecting messages Once the compiler calculates the control flow states for a role, it can generate sending and receiving functions that emit and expect the correct messages for each particular session state. The receiving functions inspect the incoming label to decide whether the message is expected, in which case, it is be further verified with cryptographic checks; otherwise, the message is just ignored.

We now turn our attention to the (cryptographic) protection of messages exchanged by roles, elucidating the elements included in each message which were first introduced in the narration earlier in this section. Consider an edge $(\widetilde{x})f(\widetilde{y})$ with target role w and source role u. The compiler generates a sending function for role u where a low-level representation of this edge is sent over the network. This message embeds values for $\widetilde{y}$, plus the tag f (as well as additional auxiliary information, detailed below).

The functional and cryptographic goals that this message needs to accomplish are:

- provide the values for $\widetilde{y}$ to w, since $\widetilde{y}$ can be read by w;
- protect the confidentiality of $\widetilde{y}$, as otherwise private values of the store may be leaked;
- ensure session integrity by giving evidence to w and subsequent roles that the message comes from the source role u and that the session specification was respected until then.

Symmetric session keys are initially established between principals via a key-establishment protocol relying on asymmetric encryption and signature; this is the only use of public-key cryptography in the generated protocol. Key establishment is carried out by including data in the regular session messages, so it does not require further messaging.

Each session message consists of

- a series of MACs from the sender and earlier participants, intended to provide integrity of the session path, and a series of (cryptographically hashed) variables needed by the receivers to recompute and verify the MACs;
- a series of (possibly encrypted) variables with their current values, for the readable variables of the receiver;
- a set of encrypted session keys, for the initial contact between the sending and receiving principals.

As an example, we detail the structure for the initial message of the Proxy session, the first message in Figures 3 and 4.

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 v, let v_c be a fresh confounder for the session run s; confounders are used when hashing variables to protect them from dictionary attacks. The format of the message sent by c to p is as follows:

$$s \mid 0 \mid \ell_0 \tag{1}$$
$$\mid mac_p^c\{s \mid 0 \mid \ell_0 \mid h[c] \mid h[p] \mid h[w] \mid h[q]\} \tag{2}$$
$$\mid c \mid c_c \mid p \mid p_c \mid w \mid w_c \mid enc_p^c\{q \mid q_c\} \tag{3}$$
$$\mid assign_c(aenc_p(\text{`}c\text{'} \mid k^a(c,p) \mid k^e(c,p))) \tag{4}$$
$$\mid mac_w^c\{s \mid 0 \mid \ell_0 \mid h[c] \mid h[p] \mid h[w] \mid h[q]\} \tag{5}$$
$$\mid assign_c(aenc_w(\text{`}c\text{'} \mid k^a(c,w) \mid k^e(c,w))) \tag{6}$$

To ease notation, we write $h[v] = h(s \mid sr \mid \text{`v'} \mid v \mid v_c)$ to denote the hash of the concatenation of the (fresh)

session identifier s, the source principal sr (c for Request), the variable tag 'v', its current value v, and the confounder v_c; 0 is the initial timestamp; ℓ_0 is the tag '(c,p,w,q) Request'; $aenc_x(m)$ denotes the asymmetric encryption of m under the public key of principal x; $assign_x(m)$ denotes the digital signature of m under the private key of x.

The correspondence between the components of the message and those shown in the narration of Request in Figure 3 is as follows:

- Line (1) is equivalent to h_0.
- Line (3) is the expansion of

$$Vars(\text{c}, \text{p}, \text{w}) \mid enc_\text{p}^\text{c}\{\text{q}\}$$

Variables c, p, and w are principal variables and hence are sent in the clear. On the other hand, variable q is a data variable and must be kept confidential; hence, it is encrypted for p.
- Line (2) is the expansion of $mac_\text{p}^\text{c}\{a_0\}$, revealing the full definition of $Hashes(\text{c}, \text{p}, \text{w}, \text{q})$ (intentionally elided in the narration) as a concatenation of the hashes of the variables c, p, w, and q.
- Line (5) is the expansion of $mac_\text{w}^\text{c}\{a_0\}$.
- Lines (4) and (6) are there for key establishment and therefore do not directly correspond to anything in the narration figure, where we omit key establishment.

In (4), the component

$$assign_c(aenc_p(\text{`}c\text{'}, k^a(c,p) \mid k^e(c,p)))$$

serves for key establishment, necessary only the first time the principal c sends a message to p. 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 ($mac_p^c\{\cdot\}$), and $k^e(c,p)$ is intended for symmetric encryption ($enc_p^c\{\cdot\}$).

Upon receiving the Request message, the principal p can choose whether to join the session in role p. It then processes (4) to obtain the session keys. These keys are used to decrypt 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 role, and that the s is not a replay. If all the checks succeed, role p updates the session store and invokes the application code handler for the Request message.

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.

Trusted platform libraries Our code relies on a set of trusted libraries for data manipulation, cryptography, networking, and managing principals. For each library, we provide two implementations: a *concrete* implementation that is used to compile and execute the session, and a *symbolic* implementation that abstractly models the library and meets a refined typed interface. Concrete libraries are not verified in our method; they form part of our trusted computing base. Theorem 1 applies to protocol code linked with symbolic 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 concatenation.

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

In this section, we outline the proof of 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. Due to space limitations, we confine most of the details to the long version of this paper [Bhargavan et al., 2009].

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 describe families of predicates that capture invariants that must be maintained by a session implementation; 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 We model a store as a session identifier, herein written s, and a variable store, written σ. We treat σ as a triple of partial maps $\sigma_v, \sigma_h, \sigma_c$, where σ_v maps variables to (their known) values, σ_h maps variables to hashes, σ_c maps variables to confounders.

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 $\text{tgt}(f)$ 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 the long version of this paper.

Invariant path predicates A pair of mutually recursive families of predicates, Q and Q', serves as the invariant at each send and receive event emitted by the generated implementation code for sessions. This 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 $\text{src}(f)$ in a session instance with identifier s is

satisfied that its global execution has followed an initial path whose image under the state function st (see Definition 2) 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.

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.

Proofs by typing Our compiler generates a refined 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 refined type interface for these two functions is of the form:

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

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; in particular, it requires that c must have sent a Request to w and that w must agree with c on the values of c, w, and q. 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 describing w's store at the internal control flow state (cwq)Request(x)Reply.

Typechecking a program against its refined interface guarantees that in every execution of the system with an active adversary, whenever a function is called, its pre-condition holds, and when it returns, its post-condition holds. The typechecker relies on the logical properties of MACs and encryptions, expressed as refined types for the $Crypto$ library, and on the secrecy and usage of keys, expressed as a session key type generated by our compiler. It analyzes the protocol code and generates proof obligations in first-order-logic, which are discharged by the Z3 SMT solver. Hence, by typechecking we establish that path predicates are maintained as an invariant by the protocol module generated by our compiler.

Lemma 1: *For any run of a verified $\widetilde{S}$-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 $a = \sigma_{\mathsf{v}}(\mathsf{tgt}(f))$, $\widetilde{v} = \sigma_{\mathsf{v}}\widetilde{y}$, and $Q'\_\rho(s, ts', ts, \sigma)$ where $\widetilde{y}$ are the read variables of f;*

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 complete our proof by hand (as our typechecker does not keep track of linearity), with a lemma establishing that the implementation of each role in a session is locally sequential:

Lemma 2: *In any run of a verified $\widetilde{S}$-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 receives such that for any adjacent pair of 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 $ts_2 + 1 = ts_3$.

Session S (see Figure 1)	Roles	S.session (lines)	Application (lines)	Graph (.dot lines)	Refined Graph (.dot lines)	S_protocol.ml (lines)	S_protocol.ml7 (lines)	Verification (seconds)
Rpc	2	8	24	11	18	472	315	6.1
Ws (a)	2	8	33	14	24	592	414	8.8
Commit	2	16	29	14	24	603	399	10.3
Wsn (b)	2	21	47	20	60	1,171	921	45.1
Fwd	3	15	38	11	19	581	357	8.6
Proxy (c)	3	28	65	26	80	2,181	1,939	154.1
Redirect (d)	3	21	34	17	31	788	550	29.3
Login	4	28	54	29	74	2,053	1,542	103.4

Figure 6. Code sizes and verification results for sample sessions

For example, in the session Ws, the type of the role function w (see Section 3) ensures that it receives and sends messages in strict alternation and that it cannot send two messages in parallel. Lemma 2 says that all role functions generated by our compiler have this property. The proof follows from the structure of the compiler code that generates role functions.

We use Lemma 2 to show that the predicates Q and Q' imply session integrity:

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 compliant events.*

Theorem 1 follows from Lemmas 1 and 3.

Secrecy This paper focuses on integrity rather than secrecy, whose formulation is more technical (as it involves the behaviour of application 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 outside the session. (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 principal plays a role in the session that can read the variable. To verify this property, we annotate each session encryption key with a refined type that enforces that the key is kept secret and that it is only used to encrypt secret values, unless the principals sharing the key have been compromised.

7. Performance evaluation

We present compilation and verification results for a series of examples. In addition to the sessions of Figure 1, our examples include a simple remote call (Rpc); a session with early commitment to values (Commit); a session with message forwarding (Fwd);

and a four-party distributed login session between a client, a gateway, a database, and a dynamically selected web server (Login). For each session, we experimentally confirmed that the generated protocol is functional, by testing it over a network.

Our compiler is written in around 4300 lines of ML. The trusted libraries, $Data$, $Crypto$, $Prins$, and Net, are shared by all session protocols and have around 800 lines of code, both for concrete (F#/.NET, Ocaml/OpenSSL) and symbolic implementations, although the concrete implementations rely on much-larger system libraries.

For each session example (S), Figure 6 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 with internal control flow states; the size of the generated ML implementation (S_protocol.ml) and its refinement-typed interface (S_protocol.ml7); and finally the time spent typechecking our implementation against refinement-typed interfaces at the end of the compilation. (The rest of the compilation is relatively fast.)

Even when programming complex multi-party sessions with many roles and messages, the application programmer needs to write less than a hundred lines of code. The generated modules are larger and more complicated—in fact the auxiliary formulas that annotate function declarations 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 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 several simple cases. Figure 7 gives the total runtimes (in seconds) for completing a number of instances of session Wsn (Figure 1(b)), which involves a loop between two roles. We used a Core 2 Duo

Session	Wsn			
Instances	5,000	5,000	50	50
Messages per session	2	20	200	2,000
Total messages	10,000	100,000	10,000	100,000
Total runtime (seconds)	4.02	11.12	1.52	6.62
consisting of				
symmetric encryption	0.10	1.01	0.09	0.68
MAC computations	0.21	1.09	0.09	0.63
key exchange	2.17	2.21	0.60	0.74
unprotected messaging	1.54	6.41	0.74	4.47

Figure 7. Cryptographic overhead

E8400 at 3.00GHz with 4GB RAM, running Linux 2.6.24 with OpenSSL cryptography and only local communication. The parameters of our runs are the number of instances (5,000 twice, then 50 twice) and the number of messages exchanged in each session (successively 2, 20, 200 and 2,000).

We measure the total runtime, first for the automatically generated cryptographic protocol then by successively disabling the payload encryption, the MAC generation and verification, and the key exchange of the protocol. We report the overhead for each of these operations. (The last line thus corresponds to sessions that run without any protection.)

The cryptographic overhead is of two kinds. For the key-establishment mechanism, it depends only on the number of instances. For the symmetric encryption and MACs, the overhead depends on the number of messages, and is around 10-15% each per message. In terms of global performance, we also measured the total runtime of establishing 5,000 SSL connections (using the openssl command line), which yields a comparable $15.93s$ in the same conditions.

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 *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. Long version of this paper, at http://www.msr-inria.inria.fr/projects/sec/sessions/, 2009.

E. Bonelli and A. B. Compagnoni. Multipoint session types for a distributed calculus. In *3rd Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *LNCS*. Springer, 2007.

R. Corin and P.-M. Deniélou. A protocol compiler for secure sessions in ML. In *3rd Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *LNCS*. Springer, 2007.

R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008. An earlier version appeared at the 20th IEEE Computer Security Foundations Symposium (CSF'07).

L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'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 *1st Symposium on Trustworthy Global Computing (TGC'05)*, Apr. 2005.

M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *20th European Conference on Object-Oriented Languages, Nantes, France (ECOOP'06)*, 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 *8th European Symposium on Programming (ESOP'99)*, pages 74–90, 1999.

A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *18th International World Wide Web Conference*, pages 561–570, 2009.

K. Honda, V. T. Vasconcelos, and M. Kubo. Lan-

guage primitives and type disciplines for structured communication-based programming. In *7th European Symposium on Programming (ESOP'98)*, volume 1381, pages 22–138. Springer, 1998.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 273–284. ACM, 2008.

R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *22nd European Conference on Object-Oriented Programming, Paphos, Cyprus (ECOOP'08)*, pages 516–541, 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'08)*, 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'03)*, pages 236–250, 2003.

Appendix A.
Well-formed properties for session graphs

We list all the well-formedness properties that we require of global 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 $\widetilde{f}f_1$ and $\widetilde{f}f_2$ 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. (Honda et al. [2008] also explore conditions under which such interconversions are possible.) All the global session graphs in this paper are automatically generated from local syntactic descriptions.

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

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 yield a *visibility* relation which details for each receiving state the list of MACs (with their contents) that have to be checked in each incoming message. In a symmetric way, a *future* relation associates to each sending state the list of MACs that need to be produced to convince future receivers. These two relations yield a *fwd_macs* relation that specifies the MACs that need to be transmitted along each message.

The knowledge that each role has of the store at each state is recorded in the *learnt* relation. It distinguishes

the knowledge of hashes and values in order to deduce the commitment checks and the *fwd_hashes* relation that lists the hashes to be forwarded in each message.

Finally, an independant *fwd_keys* relation that associates messages with the keys that need to be forwarded is computed: it is used to propagate the symmetric shared keys of the session between each pair of roles. As shown in the code below, *fwd_keys* is also 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. It corresponds to the local view of the global distributed store. 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 has hashes and confounders ∗)
    macs : {  for all l with x̃ visible
            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 content of the MACs used in the protocol (as described in Section 5). It is used by the MAC generation and verification functions.

For all state ρ that is MACed with variables $\widetilde{z}$
⌈let content_ρ_$\widetilde{z}$ = fun ts store →
 fold over $z \in \widetilde{z}$
 ⌈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⌉

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). Confounder management has been removed for clarity.

For all $\rho \xrightarrow{(\widetilde{x}) f (\widetilde{y})} \rho'$ of the refined graph
The sending role is r, the receiving role is r'
⌈let sendWired_f_ρ $\widetilde{x}$ (s:store) =
 fold over $x \in \widetilde{x}$
 ⌈ s.vars.x ← x ; s.hashes.hx ← sha1 x ;⌉
 s.header.ts ← s.header.ts + 1;
 fold over $r, r' \in fwd\_key(\rho)$
 ⌈let keyrr' = gen_keys s.vars.r s.vars.r' in
  let keys = concat keyrr' keys in⌉
 for all $(r'', \rho'', \widetilde{z}) \in future(\rho, l)$

(*header* is built as in content)
⌈let content = content_ρ''_$\widetilde{z}$ s.header.ts s in
 let mackeyrr'' = get_mackey s.vars.r s.vars.r'' in
 let macmsg = mac mackeyrr'' (pickle content) in
 let $r''f\widetilde{z}$ = concat *header* macmsg in
 let store.macs.$r''f\widetilde{z}$ ← $r''f\widetilde{z}$ in⌉
fold over $(r'', l, \widetilde{z}) \in fwd\_macs(\rho, f)$
 ⌈let macs = concat store.macs.$r''l\widetilde{z}$ macs in⌉
fold over $z \in fwd\_hashes(\rho, f)$
 ⌈let hashes = concat store.hashes.hz hashes in⌉
fold over $y \in \widetilde{y}$
⌈let keyrr' = get_symkey s.vars.r s.vars.r' in
 let encr_y = sym_encrypt keyrr' (pickle mar_y) in
 let variables = concat encr_y variables in⌉
 let security = concat keys (concat hashes macs) in
 let payload = concat variables security in
 let content = concat (utf8 (cS "ρ")) payload in
 let msg = base64 (concat *header* content) in
 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 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.