Secure Sessions for Web Services

Karthikeyan Bhargavan
Microsoft Research
and
Ricardo Corin
University of Twente
and
Cédric Fournet
Microsoft Research
and
Andrew D. Gordon
Microsoft Research

We address the problem of securing sequences of SOAP messages exchanged between web services and their clients. The WS-Security standard defines basic mechanisms to secure SOAP traffic, one message at a time. For typical web services, however, using WS-Security independently for each message is rather inefficient; moreover, it is often important to secure the integrity of a whole session, as well as each message. To these ends, recent specifications provide further SOAP-level mechanisms. WS-SecureConversation defines *security contexts*, which can be used to secure sessions between two parties. WS-Trust specifies how security contexts are issued and obtained. We develop a semantics for the main mechanisms of WS-Trust and WS-SecureConversation, expressed as a library for TulaFale, a formal scripting language for security protocols. We model typical protocols relying on these mechanisms, and automatically prove their main security properties. We also informally discuss some pitfalls and limitations of these specifications.

Categories and Subject Descriptors: C.2.0 [**Computer Communication Networks**]: General—*Security and Protection*; C.2.2 [**Computer Communication Networks**]: Network Protocols—*Protocol Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*

General Terms: Security, Languages, Theory, Verification

Additional Key Words and Phrases: Web Services, XML Security

1. INTRODUCTION

The recent specifications WS-Trust and WS-SecureConversation provide mechanisms for communicating parties to establish shared security contexts and to use them to secure SOAP-based sessions. This paper investigates the security guarantees offered by these specifications by constructing formal models in the TulaFale scripting language [Bhargavan et al. 2004]. We build our models by studying both the specifications and their implementation in WSE [Microsoft Corporation 2004]. Modelling reveals some potential vulnerabilities as well as allowing us to prove formal secrecy and authenticity properties.

Web Services and Security. Web services are a distributed system technology that emphasises flexibility and ease of deployment; typical applications include systems integration within enterprises and transactions between businesses over the Internet. Vogels [2003] presents a succinct, critical overview of the usage of web services, and their rela-

tionship with earlier technologies based on distributed objects.

Web services are built on asynchronous communication of SOAP envelopes [W3C 2003]. SOAP often travels over HTTP, but can also use other transports. The mechanisms of WS-Security [Nadalin et al. 2004] provide means to secure these messages at the application level to achieve end-to-end security, using *security tokens*. Examples of security tokens include X.509 certificates, username tokens, and XML-encoded Kerberos tickets.

In itself, WS-Security provides mechanisms for securing a single envelope. However, typically a web service and a client may interact by exchanging series of messages grouped in sessions. While in principle WS-Security could secure each separate message of the session, this can become inefficient (for example, if X.509 certificates are used in each message). Also, it is often desirable to guarantee integrity of a whole session, and not just each message. For instance, a client querying two services should not be led to attribute a response to the wrong service.

Session establishment is of course not a new issue in cryptography: indeed, numerous classic protocols aim at the mutual authentication of the parties involved in the session, the negotiation of various parameters for the session, and the protection of further traffic. (See for example Diffie and Hellman [1976], Needham and Schroeder [1978], SSL [Freier et al. 1996], and IKE [Harkins and Carrel 1998].) Moreover, their main secrecy and authentication properties have often been thoroughly studied. Most of their concepts and mechanisms can be usefully applied to SOAP-based protocols, but experience also suggests that this adaptation is not straightforward.

WS-Trust and WS-SecureConversation. Building on top of WS-Security, WS-Trust describes how security tokens can be requested and issued by SOAP processors; it relies on a dedicated *security token service* (STS) to evaluate requests and issue tokens. Moreover, WS-SecureConversation describes the usage of one such token, named a *security context token*. The token points to a *security context* (SC) typically shared between a client and a web service; its content can be used to derive keys to protect traffic between these two parties.

This paper is based on the two initial versions of WS-Trust and WS-SecureConversation, both published in May 2004 (version 1.1 [Kaler et al. 2004b; 2004a]) and revised in February 2005 (version 1.2 [Gudgin et al. 2005b; 2005a]) by an industrial working group including IBM, Microsoft, RSA Security, and Verisign. These specifications were contributed to the OASIS Web Services Secure Exchange (WS-SX) Technical Committee in December 2005. At the time of writing, the latest draft standards were both published in September 2006 (version 1.3, Committee Draft 0.1 [Gudgin et al. 2005c; Barbir et al. 2006]). The newer versions extend the core mechanisms initially proposed in the specifications, but do not significantly change them. In this paper, their differences are unimportant, so we focus on version 1.2, and carefully discuss any part of our model that is not in direct correspondence with this version of the specifications.

Figure 1 shows a typical usage scenario of WS-Trust and WS-SecureConversation. It roughly corresponds to the sample protocol given in a WSE tutorial [Gudgin 2004]; we refer to this tutorial for additional implementation details for this protocol.

Three SOAP processors are displayed: a client, a security token service (STS), and a web service. For simplicity, both the STS and the service are co-located and share a session cache (the dashed line in the figure). The STS is configured to establish security contexts (SCs) with authenticated clients, to be used between clients and services. The first

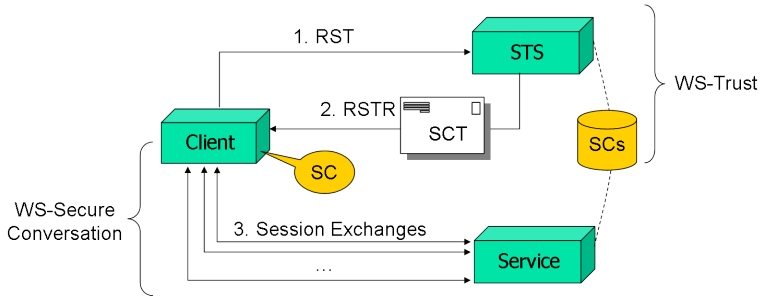

Fig. 1. A typical protocol relying on WS-Trust and WS-SecureConversation

two steps rely on mechanisms covered by WS-Trust, while the later exchanges (step 3) are specified by WS-SecureConversation.

The execution proceeds as follows. At step 1, the client contacts the STS with a *request security token* (RST) message, including some form of identity token plus information about the target service. The STS, after authorization of the request, generates a new security context SC, caches it, and replies with a *request security token response* (RSTR) message that includes a security context token (SCT) to indicate that a new SC has been created (step 2). The RSTR contains enough information to allow the client to compute the same SC (with the same shared secret) as the cached version. This allows the client and service to start exchanging messages (step 3) protected using keys derived from the shared secret of the SC.

Our Contribution. We propose a formal counterpart to web services security specifications for session management, as a collection of predicates and processes reflecting their semantics. We describe a realistic threat model for web services, essentially an active attacker with some access to insider secrets. We develop simple, typical protocols relying on these specifications. Some of these protocols correspond directly to example code distributed with WSE [Microsoft Corporation 2004], while others are protocols specified for testing interoperability of different implementations of WS-Trust and WS-SecureConversation [WS 2004]. We check experimentally that the message formats in our formal protocols correspond to the XML documents exchanged by their implementations. We state and prove a series of core security properties for these protocols, thereby gaining confidence in our model of these specifications. Our proofs make the standard symbolic assumptions about cryptography introduced by Dolev and Yao [1983]. We also informally discuss some limitations and potential vulnerabilities.

Web services documents leave many details unspecified, some of them essential for security modelling, so we adopt the following method: we comply with the details of the specifications whenever available; otherwise, we observe the implementation decisions embedded in WSE and follow their details; finally, we fill any remaining gaps in the model from our experience of similar protocols. We systematically discuss these two subsidiary cases. To the best of our knowledge, ours is the first formal analysis of these two web services specifications, and these are the most complex SOAP-based security protocols yet formalized.

The development of the web services specifications emphasises flexibility and compositionality, partly as a means of achieving consensus between different vendors. Simi-

larly, developer workshops emphasise interoperability testing between different implementations. Flexibility, compositionality, and interoperability are attractive, of course, but they also complicate security analysis. There are insecure as well as secure ways to compose these specifications. The extreme flexibility of message formats and XML signatures, for example, can lead to misinterpretations. In the end, the actual security of a SOAP-based system that complies with the specifications still needs to be checked specifically. Testing interoperability is independent of establishing security properties. Hence, the results of this paper—compositional formal libraries and analyzes of typical combinations of web services protocols—usefully complement the standardization process and the interoperability workshops.

Structure of the Paper. Section 2 reviews our prior work on modelling web services security. Section 3 outlines the WS-Trust specification and Section 4 formalizes a basic RST/RSTR exchange conforming to WS-Trust. Theorem 1 shows that the exchange allows the two parties to reach agreement on a security context. Theorem 2 shows that the key associated with the newly-established security context is a secret shared between the two parties. Section 5 outlines the WS-SecureConversation specification and Section 6 develops a formal model of request/response exchanges conforming to WS-SecureConversation, that builds on an initial RST/RSTR exchange. Theorem 3 shows that the two parties can agree on the contents and correlation of the first request and response, and that the secrecy of the request and response bodies is preserved. Theorem 4 generalizes these results to unbounded sequences of exchanges, and shows that the parties agree on the contents of the whole session, and that the secrecy of all bodies is preserved. Section 7 applies our modelling approach to an interoperability scenario designed to test WS-Trust and WS-SecureConversation. Theorem 5 shows agreements between the parties for each of the six messages of the scenario. Finally, Section 8 discusses related work and Section 9 concludes.

Appendix A provides the proof of Theorem 4; the proof relies on lemmas proved by hand as well as lemmas proved automatically. The paper includes only small excerpts from the TulaFale scripts used to establish its main results. The complete scripts, including our library for WS-Trust and WS-SecureConversation, are included in the TulaFale distribution, available at `http://Securing.WS`.

Earlier versions of this paper are an abridged workshop paper [Bhargavan et al. 2004a] and a technical report [Bhargavan et al. 2004b].

2. MODELLING SYSTEMS AND THREATS

The security results in this paper are relative to the formal threat model of Dolev and Yao [1983]. In this model, protocol participants use idealized cryptographic primitives, and there is an active attacker able to record, compute, and send messages, but not simply to guess secrets.

Our formalizations are based on TulaFale [Bhargavan et al. 2004], an XML version of the Dolev-Yao model embedded within the pi calculus. The pi calculus [Milner 1999] is a theory of concurrency in which concurrent computations are expressed within a small syntax of message-passing *processes* that communicate on named channels. A computation in the pi calculus consists of a sequence of *reductions* in which a message is passed from a sender to a receiver process. When considering cryptographic protocols in the pi calculus, protocol participants are written as explicit processes, whereas the active attacker

is thought of as an arbitrary unknown process running in parallel with the protocol participants [Abadi and Gordon 1999; Abadi and Fournet 2001]. There is a wide range of formal techniques, including automated tools, for analyzing such models of cryptographic protocols expressed in variations of the pi calculus. In particular, our TulaFale tool makes use of the ProVerif protocol analyzer [Blanchet 2001; 2002].

This section divides into three parts. We first review the TulaFale language; we then explain the threat model considered in this paper; we finally present our model for principals and their cryptographic materials.

2.1 Systems Modelling in TulaFale

This section reviews the TulaFale language; the complete syntax appears in Figure 2.

A TulaFale script ($D_1 \cdots D_n$ P) consists of a series of declarations $D_1 \cdots D_n$ followed by a TulaFale process P that defines a system of multiple processes running in parallel, representing protocol participants. We focus first on the syntax for processes. The null process 0 does nothing; P |Q is the parallel composition of P and Q; the replication **!**P behaves as an infinite number of copies of P running in parallel. The restriction **new** x:s;P generates a fresh name x, then behaves as P. (Fresh names can be used to model nonces or session keys.) The conditional construct **if** F **then** P **else** Q tests the formula F and then behaves as P or Q accordingly. The process **filter** F $\rightarrow x_1,\cdots,x_n$; P evaluates the formula F, and then behaves as P with messages that satisfy F substituted for the variables $x_1,\cdots,x_n$. (Logical formulas are used in TulaFale to construct and check messages; they are explained below.)

Processes interact by sending messages (**out** c(M)) and receiving messages (**in** c(x)) on fixed channels. These channels are introduced by the declarations **channel** $c(s_1,\cdots,s_n)$ and **private channel** $c(s_1,\cdots,s_n)$; when present, the optional keyword **private** indicates that the channel is not initially available to the attacker. For instance, in this paper, we model SOAP messaging between web services and their clients as message sends and receives on a single public 'soap' channel, also available to the attacker.

Processes finally record the progress of the protocol using events (introduced by the declaration **event** $f(s_1,\cdots,s_n)$). Specifically, processes mark the initiation (**begin** f(T)) and apparent completion (**end** f(T)) of each phase of the protocol using events parameterized by a tuple of values T that record data such as the identity of the protocol participants and the content of messages exchanged.

Messages are terms in an algebraic model of XML extended with signature and encryption primitives that represent idealized cryptographic functions [Bhargavan et al. 2005]. Messages have sorts; for instance, a **string** is an XML string, an **item** is an XML element or string, and a **bytes** is an array of bytes. Simple messages include literal quoted strings, such as `"WS-Security"`, alphanumeric identifiers that stand for variables, such as u, and an anonymous variable _. More complex messages include terms that can be constructed by applying functions to messages, by collecting messages in tuples or lists, or by inserting them in XML elements or attributes. Lists of items are usually bracketed, as in [<Body>rstr</><RelatesTo>rto</>], although we omit the brackets when the list is the body of an XML element. The infix function @ represents list concatenation. Function symbols include constructors and selectors. Each constructor is introduced by a declaration **constructor** $f(s_1,\cdots,s_n)$:s. Each selector is introduced by a declaration **destructor** $f(s_1,\cdots,s_n)$:s **with** $f(M_1,\cdots,M_n)$=M, where the equation explains how to rewrite messages that include the selector, by pattern matching. Function symbols such

x,c,f,p	identifiers
tag	XML tags
s,t ::= **string** |**item** |**items** |**att** |**atts** |**bytes**	sorts
M, N::=	messages
`"string"`	literal string
x	variable
_	anonymous variable
$f(M_1,\cdots,M_n)$	constructor/selector application
$(M_1,\cdots,M_n)$	tuple
$[M_1 \cdots M_n$ @ M]	list (with optional tail)
`<tag` $M_1 \cdots M_n$`>`$N_1 \cdots N_k$ `</>`	XML element
tag = M	XML attribute
F, G, H::=	formulas
M = N	equality
M **in** N	list membership
$p(M_1,\cdots,M_n)$	predicate application
F,G	conjunction
P, Q, R::=	processes
0	null process
P |Q	parallel composition
!P	replication
new x:s; P	fresh name generation
if F **then** P **else** Q	conditional (with optional **else**)
out c(M); P	message output
in c(x); P	message input
filter F $\rightarrow x_1,\cdots,x_n$; P	formula evaluation
begin f(T); P	begin event
end f(T); P	end event
D::=	declarations
private channel $c(s_1,\cdots,s_n)$.	channel (optionally private)
event $f(s_1,\cdots,s_n)$.	event constructor
predicate $p(x_1{:}s_1,\cdots,x_n{:}s_n)$:– F.	predicate clause
constructor $f(s_1,\cdots,s_n)$:s.	constructor
destructor $f(s_1,\cdots,s_n)$:s **with** $f(M_1,\cdots,M_n)$=M.	selector rule
S ::= $D_1 \cdots D_n$ P	TulaFale script

Fig. 2. TulaFale Syntax

as base64, utf8, and psha1 are abstract representations of operations on the data model; the function psha1 is an idealized hash function with no inverse. For the sake of brevity, TulaFale omits all XML namespace information, and uses some non-standard abbreviations, such as omitting the tag in a closing element bracket `</>`.

Formulas enable processes to construct and check messages using logic programming. Formulas can be declared using named predicates. For example, the predicate below precisely models the structure of username tokens [Nadalin et al. 2004, Section 6.2], which are typically used to identify principals in messages, and defines the computation of a shared authentication key from a password associated with the principal.

```
predicate isUserToken (tok:item,u,pwd:string,n:bytes,t:string,k:bytes) :–
  tok = <UsernameToken>
          <Username> u </>
          <Nonce> base64(n) </>
          <Created> t </>
```

```
        </>,
  k = psha1(pwd,concat(utf8("WS-Security"),concat(n,utf8(t)))).
```

In the predicate, the message tok is matched against an XML element that includes the username u, the nonce n, and the timestamp t; the key k is then computed from a password plus the nonce and timestamp of the token. Given certain implementability constraints [Bhargavan et al. 2005], predicates may be used in different modes, that is, with different bound input and output parameters. For example, a process may call the predicate above to build a username token tok and compute its associated key k by instantiating all the other parameters.

All the predicates shown in the remainder of the paper are extracted from the code of our formal model, which is included in the TulaFale distribution.

2.2 Threat Model and Security Goals

We model the unknown active attacker as an implicit process that runs alongside the explicit processes, and which may interact with them via public channels, such as soap. By default, all channels are public; the attacker process has no direct access to any channels marked private, which typically model private databases shared between one or more clients and servers.

For a given TulaFale script, a *run* is any series of (potentially nondeterministic) pi calculus reductions and events starting from the explicit system composed with the attacker. The attacker process is arbitrary, except it may not itself generate any events. The observable result of a run is a set of events.

Our authentication results are one-to-many correspondences [Woo and Lam 1993] (also known as non-injective agreements [Lowe 1997]) between events. Correspondences are parameterized by the specific data, such as participant identities and key material, established by a part of a protocol. The goals of session establishment may seem obvious and universal, but in fact the details are subtle and vary from protocol to protocol [Gollmann 2003]; in general, the use of correspondences allows us to articulate precisely the goals of particular protocols. We formulate correspondences as *robust safety* theorems: in every run of the system, every occurrence of an **end** event has a corresponding **begin** event carrying the same data. (We qualify these safety properties as *robust* to indicate they hold despite the presence of an arbitrary active attacker.) For instance, authentication of an RST message is expressed as a correspondence between events marking the client sending and the server receiving the RST.

Our secrecy results concern the absence of direct flows of data from the system to the attacker. We formulate these as *secrecy* theorems asserting that certain names do not come into the possession of the opponent. (Names in the pi calculus are atomic values representing channels, keys, or other data.) For instance, we show that a name representing a freshly generated session key does not flow from system to attacker.

Proof Techniques. We rely on the ProVerif protocol analyzer for proving the security properties of our TulaFale models. ProVerif is a specialized, automated tool that can establish authentication and secrecy properties for all runs of a given pi calculus system, against all pi calculus attackers. In contrast with model checkers, ProVerif does not bound the number of parallel sessions or the size of messages. It relies on a series of abstractions, an internal compilation to Horn clauses, and a resolution-based algorithm. Since the target properties are undecidable, ProVerif may also diverge. In this paper, however, we could

eventually verify all stated security properties for the sample protocols, after some careful tuning of our TulaFale libraries and tools.

Our main technical result, Theorem 4, establishes a general correspondence property whose statement involves an unbounded number of events (each representing a message in a given session). Its proof is beyond ProVerif. Instead, we decompose this proof into ProVerif lemmas concerning auxiliary scripts, plus composition lemmas with handwritten proofs based on the pi calculus theory. This delicate proof is detailed in Appendix A.

2.3 Principals, Authentication Materials, and Key Leakage

Our models assume the following participants and authentication materials:

—A single certification authority (CA), with public/private key pair Ker/sr, that issues X.509 public-key certificates identifying clients and services, signed by the private key sr.

—Multiple principals, each identified by a username u, and equipped with passwords or X.509 certificates issued by the CA.

We assume a single trusted database (coded as messages on a private channel, as in prior work [Abadi et al. 2004]) that relates usernames to passwords or private keys and certificates. We allow each principal to have multiple passwords and multiple certificates. A certificate for principal u has subject name u. This database is not accessible to the attacker, but is accessible to client and server processes acting on behalf of users. (In practice, each principal would access instead a local database that includes only a small subset of all passwords and private keys, possibly associating remote principals with different passwords or keys. Our assumption is safe for proving security properties, inasmuch as our single database may in particular contain the union of all these local databases.)

We do not fix a particular principal population; instead, we provide public channels to allow the attacker to trigger the generation of fresh authentication materials for arbitrary usernames. Similarly, we do not fix any particular protocol sessions or bound the number of concurrent sessions. We allow the attacker to initiate sessions with arbitrary principals in the roles of clients and servers, and with other parameters chosen by the attacker.

We assume the attacker never gains knowledge of the private key of the CA. However, to model insider attacks, we allow passwords, other private keys, and security contexts to leak to the attacker. In our setting, we say a principal is *unsafe* if any of their passwords or private keys has been leaked to the attacker; otherwise, we say the principal is *safe*. Similarly, we say a WS-Trust security context is *unsafe* if it has been leaked to the attacker, and is *safe* otherwise.

In summary, our system model provides an interface—a set of public channels—to the attacker, giving it the following abilities:

—to send and receive on the soap channel;

—to trigger the generation of a fresh password or a new certificate for any principal;

—to initiate sessions and provide their parameters to clients and servers;

—to cause the leak of passwords or certificates for any principal (but not the certificate authority);

—to cause the leak of established security contexts.

This amounts to a realistic threat model for XML rewriting attacks on web services; it is essential to consider vulnerabilities due to unsafe principals—insider attacks—and indeed

we describe such vulnerabilities. (Other threats to web services outside the scope of this model include SQL injection attacks in SOAP payloads and buffer overruns on the networking stack.) Our formal properties concern safe principals, and hold despite the active attacker's ability to craft messages using the authentication materials of unsafe principals. This model of systems and potentially unsafe principals is similar to the TulaFale model in an earlier paper [Bhargavan et al. 2004], and comparable to the system model in other formalisms [Paulson 1998].

3. WEB SERVICES TRUST LANGUAGE

WS-Trust "provides a framework for requesting and issuing security tokens, and to broker trust relationships" [Kaler et al. 2004b]. We survey and discuss its contents, focusing on the parts modelled in this paper. We refer to the specification for additional information.

3.1 WS-Trust as a Protocol Framework

WS-Trust introduces dedicated web services, named *security token services* (STS), that handle requests for security tokens (RSTs) and send responses (RSTRs). Like any SOAP messages, envelopes carrying RSTs and RSTRs may be protected using a selection of mechanisms described in WS-Security.

WS-Trust is deliberately abstract; it provides a general terminology, some precise XML syntax for exchanged data, and an informal description of how to establish security contexts. On the other hand, it avoids defining complete protocols; for instance, it never prescribes any kind of authorization procedure for establishing a security context.

In a common case, a single exchange establishes context: the client sends an RST as the body of an envelope; the STS replies with an RSTR as the (partly encrypted) body of another envelope; and both envelopes include security headers for authentication.

However, other flows of RSTs and RSTRs are possible. In more complex exchanges, any subsequent messages received by the STS are also formatted as RSTRs. In addition, STSs may initiate exchanges by sending unsolicited RSTRs. STSs implement four SOAP actions and corresponding message elements for managing security tokens: for issuance, renewal, cancellation, and validation. Moreover, RSTs and RSTRs need not appear as envelope bodies; they may also be incorporated in the security headers of envelopes carrying some other primary payload. WS-Trust allows security token exchanges to be nested. For example, a client may need to contact several STSs in order to accumulate enough cryptographic evidence before accessing a service; similarly, an STS may contact other STSs in the process of gathering adequate security tokens. Finally, this traffic may itself be protected using tokens previously exchanged.

The goal of these exchanges is to reach an agreement on a *security context* (SC) shared between different parties. However, WS-Trust leaves the nature of the agreement unspecified. For instance, an STS may simply be a public repository for X.509 certificates that accepts anonymous requests and responds with matching certificates, with no particular trust relationship or agreement at the end of the exchange. On the other hand, an STS may establish a protected session between a client and a service, after authenticating the client and enforcing access control to the service, thereby ensuring a precise agreement between the client and the service.

Our formal model (in Section 4) focuses on the core security aspects of WS-Trust. The model omits some other aspects: renewal, cancellation, and validation actions; error handling; unsolicited RSTRs; and advanced algorithm negotiation and delegation mechanisms.

We believe these aspects are optional; experimentally, we did not encounter sample code that relied on them. We also checked that their syntax does not clash with the syntax of the elements verified in our model, so that at least our model safely applies to implementations that neither support them (when producing messages) nor accept them (when consuming messages), even when they interact with more advanced implementations.

WS-Trust also proposes an optional attribute RequestSecurityToken/@context for correlation between RST and RSTRs. As recommended in WS-Security, our model relies instead on the WS-Addressing message identifier of the enclosing envelope, which plays a similar role and makes this context attribute redundant.

3.2 Syntax for RST/RSTR Exchanges

In what follows, we focus on STSs that implement a simple, two-message RST/RSTR exchange for establishing a security context, as described in the specification [Kaler et al. 2004b, Section 6.1-2]. We begin by explaining the detail of the syntax of these messages and their intended semantics, which we reflect in our models.

—*Principals:* As described in WS-Trust, an envelope that carries an RST may contain a BaseToken element, typically an X.509 certificate or a username token, that explicitly identifies and authenticates the requesting principal. Alternatively, the RST may be anonymous. Besides, the RST may contain an `<AppliesTo>` element indicating the service with whom the client wishes to establish a security context.

—*Keying:* WS-Trust provides optional mechanisms for key establishment: both the client and the STS may include some (encrypted) fresh random value, referred to as *entropy*; the established security context key, if any, is either one of these values, or their joint hash. In the latter case, for instance, each party decrypts the other party's entropy, then computes sckey = psha1(clientEntropy,stsEntropy) and stores this key as part of the newly-established security context. A benefit of this computation is that the freshness and psendo-randomness of the key are guaranteed, irrespective of the other party's choice of entropy. Most modern session-establishment protocols offer similar guarantees, see for example SSL [Freier et al. 1996] and IKE [Harkins and Carrel 1998]. Conversely, if for instance the client accepts an STS-only key, an unsafe STS may supply an arbitrary key, possibly already used in another session.

—*Negotiation:* WS-Trust describes additional information that may be included in RSTs, used for instance to demand some choice of cryptographic algorithm or policy, or to provide authorization materials. We deal abstractly with such additional information: we record it in the security context, and thus authenticate it, but leave its interpretation to the application.

As a first concrete example of TulaFale code modelling WS-Trust, we give the predicates that verify the structure of RST and RSTR elements in our script. Anticipating WS-SecureConversation, we assume that `"SecurityContextToken"` (SCT) is the type of the requested security token: a basic token with an identity and a key, computed here from client and server entropies.

```
predicate EntropicRST(rst:item,srvURI:string,etok,ExtraInfo:item):–
  rst = <RequestSecurityToken>
          <TokenType>"SecurityContextToken"</>
          <RequestType>"Issue"</>
```

```
    <AppliesTo><EndpointReference>srvURI</></>
    <Entropy>etok </>
    ExtraInfo
  </>.
```

After extracting the RST from an incoming envelope, the STS uses this predicate to decompose this RST. The first argument of the predicate, presumed to be an rst element, is decomposed by pattern matching into a series of elements. The constant parts in the pattern ensure the RST is a well-formed request for SCT issuance; srvURI provides information on the intent of the SCT, here the URI of the web service; etok is the client entropy, encrypted for the service; finally, ExtraInfo collects elements not explicitly used in our model, but perhaps trusted by the protocol participants.

WS-Trust prescribes that RSTRs returned by the STS include a *requested security token* that indicates the identifier of the (newly created) SCT and a *requested proof token* that contains (typically encrypted) server entropy used to compute the SCT key. It may also include an <AppliesTo> (not necessarily matching the RST).

```
predicate EntropicRSTR(rstr:item,srvURI:string,BaseToken:item,
                       uriSTS:string,sctid:string,etok:item):−
  rstr = <RequestSecurityTokenResponse>
      <AppliesTo><EndpointReference>srvURI</></>
      <RequestedSecurityToken>
        <SecurityContextToken>
          <Identifier>sctid</></></>
      <Entropy>etok </>
      <RequestInfo>
        BaseToken
        uriSTS </>
    </>.
```

The client decomposes each incoming RSTR with this predicate; it extracts the srvURI (implicitly comparing it to the request srvURI if this parameter is bound when calling the predicate) and the STS's contributions to the SC (namely its identifier sctid and its encrypted entropy etok).

Departing from the WS-Trust specification, our model also requires that the RSTR include a non-standard element, <RequestInfo>, with additional information about the RST, namely the token used to sign the RST and the URI it was sent to. As discussed in more detail in Section 4.6, it is necessary to authenticate the content of <RequestInfo> in order to correlate securely the RSTR with its RST. Our syntactic extension provides a simple solution, but other mechanisms are available.

3.3 Towards an Explicit Agreement on the Exchange

Session establishment is a well-studied goal for cryptographic protocols. In contrast to specifications of fixed protocol, however, WS-Trust leaves open several important design decisions that should be carefully considered when assembling a protocol.

Crucially, RST/RSTR exchanges aim to establish shared security contexts, but the contents of these contexts (including the participants' intentions) is left implicit. This can be a source of confusion, inasmuch as the flexibility of web services enables many different levels of agreement between processors sharing a security context. Ideally, the specifications should help secure precise agreements on security contexts between clients, STSs,

and servers. Following well-established prudent practices, a simple way to achieve strong agreement would be to supplement the syntax of RSTs and RSTRs with (optional, well-defined) data on the exchange, such as selected modes for authentication and keying, and identities of the requester, issuer, and target service. It is also important that this syntax be specified, so that its presence and contents can be validated.

For a given system, one should explicitly state what is guaranteed, both when an STS accepts an RST and issues an RSTR, and then when a client accepts an RSTR. These guarantees depend both on the contents and processing of the RST and RSTR. Hence, one should carefully review:

(1) what needs to be agreed upon—typically not just the SCT key;

(2) what is passed in the RST/RSTR (notably the signed materials in these messages);

(3) whether the web service implementations actually provide an agreement based on their processing of the exchange.

In a given implementation, an effective agreement depends on details of envelope processing. Still, the safety of security contexts should not overly rely on implementation choices. At least, whenever an exchange succeeds, the protocol designer may expect that any piece of data recorded in the security context is authentic. In comparison, traditional session establishment protocols like SSL [Freier et al. 1996] or IKE [Harkins and Carrel 1998] have specific options and guarantees to reach precise agreements, typically covering at least any data exchanged by the protocol.

Following the scenario illustrated in Figure 1, we define a concrete agreement. The agreement should at least cover the actual contents of SCs observed in the WSE implementation: a shared SCT identifier, a key, and some identity information for the three involved principals. It should also cover security parameters used in the exchange, such as:

—Whether the RST client is authenticated or anonymous. If this is not covered, a client may be convinced it is authenticated because its signature was accepted, whereas the STS received an unsigned request with the same message identifier and assumed an anonymous token was requested.

—Whether both the client and server, or the server alone, provided entropy. A mismatch may lead to an apparently successful SC recording a bad key.

—Any data used to authorize the issuance of the RSTR, such as the security token providing client authentication, or credentials presented in the RST.

—The URI and action for the STS. This may matter if different STSs enforce distinct authorization policies for the same service.

We arrive at the following content for the security context in our model, expressed as a TulaFale predicate:

```
predicate sctSC(SC:item,sctid:string,sckey:bytes,mode:string,
                UserToken,StsInfo:item,appTo:string,extra:item) :−
  SC = <SecurityContext>
         <Identifier>sctid</>
         <Key>base64(sckey)</>
         <Base>UserToken</>
         <STSInfo>StsInfo</>
         <AppliesTo>appTo</>
```

```
<EntropyMode>mode</>
<ExtraInfo>extra</></>.
```

Crucially, the security context records information on the identity of the three principals involved: `<Base>` records the token for the client; `<STSInfo>` records the URI and token for the STS; `<AppliesTo>` records the URI for the service.

3.4 Other Security Considerations

WS-Trust leaves authorization checks unspecified. They may be performed both at the STS and the service. For a given system, it is important to document who performs the checks, and how to interpret the privileges associated with a valid security token.

WS-Trust does not prescribe a particular message flow once the security token has been established, but (apparently) often assumes a single client will initiate all traffic. Although each token may typically be associated with a single session between two endpoints, in principle it may be involved in parallel sessions by multiple processors, and may be accepted by multiple services. This flexibility can seriously complicate session management, and replay protection in particular. Also, the web service may in principle access and use a security context before the client completes the protocol, with weaker security guarantees. For example, if an attacker has tampered with the RST, the client will detect the attack once it receives the RSTR, but in the meantime the service may have proceeded on the basis of the modified RST.

Regarding privacy, although it is possible to use pseudonyms [Kaler et al. 2003a], an eavesdropper that monitors an exchange may extract detailed information on its participants from the explicit, semi-structured message format, including for instance their identities, and sometimes the purpose of the session. Moreover, this information may be signed using long-term certificates, and thereby provides non-repudiable evidence.

In general, to resist denial-of-service attacks, responders in session establishment protocols should avoid allocating resources or performing expensive computations until initiators are authenticated. In our scenario, the web service is reasonably protected but the STS is open to attack. For example, an attacker may replay messages with modified elements, leading to expensive (failing) signature verifications. This problem may be alleviated using several tiers of STSs, or two-round RST/RSTR protocols with weak authentication in the first round guarding public-key authentication in the second round.

4. MODELLING AND VERIFYING USES OF WS-TRUST

We present our model of a single RST/RSTR exchange, such as the first exchange of the protocol depicted in Figure 1. The goal of the exchange is to ensure agreement on a shared security context. We first describe the exchange, then we detail our implementation of principals and the processing of RST and RSTR envelopes, and finally we state our main theorems for the resulting script, verified using the TulaFale and ProVerif tools. For simplicity, we only provide a few exemplary excerpts of the script—we refer the reader to `ssws-trust.tf` for the complete definitions.

4.1 RST/RSTR Exchange and Security Goals

A basic RST/RSTR exchange consists of two messages exchanged between a client and an STS process, as depicted in Figure 3. The goal of the exchange is for the two processes to establish and agree on an SC with a fresh identifier sctid and shared key sckey.

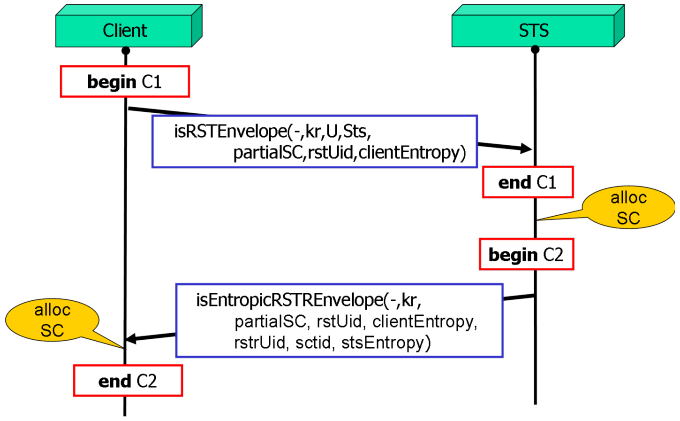

Fig. 3. Establishing a security context in entropic mode

This agreement is achieved in two steps. After the RST message has been accepted, the client and STS agree on a *partial* SC that consists of all the elements of the SC except `<Identifier>`, `<Key>`, and the STS token (within `<STSInfo>`), which are undetermined at this stage. After the RSTR message is accepted at the client, both processes agree on the full established SC. In the rest of this section, we describe how the two processes construct and check messages to achieve this agreement.

To begin with, both processes know kr, the public key of the CA used to check the validity of public key certificates, and share the trusted database of principal secrets. The client principal is represented by a user principal record U (see Section 2) that contains either a username and password or an X.509 certificate and its associated private signing key. The STS principal is represented by a server principal record Sts.

In our exchange, each envelope incorporates a globally unique identifier; its structure is represented by the predicate uid; it consists of a freshly generated message identifier id and a public timestamp t; it is typically used to protect against message replays.

The process Client represents an instance of a SOAP client sending RSTs and processing RSTRs on behalf of a user. Next, we describe its high level behaviour (in Section 4.2 we elaborate the Client process in full detail). Each run is in one of two operation modes: either *entropic mode*, where the client provides entropy for the security context, or *non-entropic* mode, where it does not. In both modes, the server provides entropy. (We do not model a third mode, allowed by the specification, where the client alone provides entropy.) Figure 3 depicts a typical run in entropic mode. In both modes, the client process is given a partialSC that provides the parameters for a new security context. The client then constructs an RST message corresponding to partialSC, from U to Sts, containing a unique identifier, rstUid, and in entropic mode, a fresh client-generated value, clientEntropy.

The process STS represents an STS server. When it receives an RST message, it checks and extracts the received security context parameters partialSC. The STS then generates a new full security context, SC, with a fresh identifier sctid. It generates its own stsEntropy and uses it to compute the shared sckey associated with this security context. It then returns to the client an RSTR, uniquely identified by rstrUid, containing the sctid and stsEntropy.

The server entropy in the RSTR, and, in entropic mode, the client entropy in the RST, are encrypted and then signed, as in the WSE implementation [Microsoft Corporation 2004].

The intentions of and the agreement between the client and STS are recorded using **begin** and **end** events, as follows. Before sending the RST, the client records its proposed security context as the event **begin** C1. After checking the RST, the STS indicates its acceptance of the proposed security context as the event **end** C1. Similarly, before sending the RSTR, the STS records the established security context as **begin** C2, and after checking the received RSTR, the client indicates acceptance of the security context with **end** C2.

The correspondence assertion C1 after the first message requires that the client and STS processes agree on the values of the proposed parameters partialSC, the rstUid, and the clientEntropy:

$$C1 = (partialSC,rstUid,clientEntropy)$$

Including rstUid in C1 provides support for replay detection: if the STS process were to further ensure that it never accepts two RSTs with the same Uid, then agreement on C1 implies that each message sent by the client is accepted at most once by the STS process.

The correspondence assertion C2 after the second message requires that the client and STS processes agree on the full established SC and the unique identifiers of both messages (again to enable replay detection).

$$C2 = (SC,rstUid,rstrUid)$$

We say that a principal is a client, STS, or server in C1 (or C2) if it is recorded under the corresponding role in the security context partialSC (or SC). For instance, we say that C1 has a *safe client* if the principal recorded in the Base field of partialSC is safe.

4.2 Mapping Principals to TulaFale Processes

In order to illustrate the coding of processes in TulaFale, we detail the Client process for the RST/RSTR exchange. The complete process, listed in Figure 4, is extracted verbatim from our script; it defines the series of actions performed by the Client (including calls to filtering predicates, communications, and event logging). Next, we focus on describing its actions. Sections 4.3 and 4.4 detail selected predicates involved in these actions. The Client process can be divided into two phases: it first computes the values needed to instantiate the RST/RSTR exchange, and then it performs the exchange either in entropic or in non-entropic mode.

The process is initialized with a bytestring kr, representing the public key of the CA. It then receives from the attacker, on the public channel init, a partial security context partialSC that sets the parameters for the requested security context, and a timestamp t1. (By receiving these values from the attacker, we allow all possible non-secret choices for these variables.) The client process retrieves the user record U from the trusted database channel anyPrincipal, generates a fresh message identifier id1, and uses the predicate uid to create a unique identifier rstUid for the RST message. The process calls the predicates PartialSctSC, hasPrincipal, and isSTSInfo, to decompose partialSC and extract entropy mode entropyMode, user principal name u, and server principal name subjSTS. (The other extracted variables are unused and can be ignored for now.)

—In case entropyMode = `"Both"`, indicating that both the client and server must generate entropy, the client continues in entropic mode. It generates a fresh bytestring clientEntropy and constructs the RST message muskrats by calling mkEntropicRSTEnvelope. The process then issues the event **begin** C1(u,partialSC,rstUid,clientEntropy),

```
process Client(kr:bytes) =
  in init (partialSC,t1);
  in anyPrincipal(U);
  new id1:string;
  filter uid(rstUid,id1,t1) → rstUid;
  filter PartialSctSC(partialSC,entropyMode,UserToken,StsInfo,x1,x2) →
                                    entropyMode,UserToken,StsInfo,x1,x2;
  filter hasPrincipal(UserToken,u) → u;
  filter isSTSInfo(StsInfo,uriSTS,subjSTS,ekSTS,certSTS) → uriSTS,subjSTS,ekSTS,certSTS;

if entropyMode="Both" then
(
  new clientEntropy:bytes;
  filter mkEntropicRSTEnvelope(msgrst,kr,U,partialSC,rstUid,clientEntropy) → msgrst;
  begin C1 (u,partialSC,rstUid,clientEntropy);
  out soap(msgrst);
  in soap(msgrstr);
  filter isEntropicRSTREnvelope(msgrstr,kr,partialSC,
              rstUid,clientEntropy,rstrUid,sctid,stsEntropy) → rstrUid,sctid,stsEntropy;
  filter fillSC(fullSC,partialSC,sctid,psha1(base64(clientEntropy),stsEntropy)) → fullSC;
  end C2 (u,subjSTS,fullSC,rstUid,rstrUid);
  !out anySCClient(fullSC))

else (if entropyMode="Server" then
   (
   filter mkNonEntropicRSTEnvelope(msgrst,kr,U,partialSC,rstUid) → msgrst;
   filter remCertPartialSctSC(partialSC,remCertSC) → remCertSC;
   begin C1 (u,remCertSC,rstUid,utf8("zero"));
   out soap(msgrst);
   in soap(msgrstr);
   filter isNonEntropicRSTREnvelope(msgrstr,kr,U,partialSC,rstUid,rstrUid,sctid,stsEntropy) →
                                                          rstrUid,sctid,stsEntropy;
   filter fillSC(fullSC,partialSC,sctid,stsEntropy) → fullSC;
   end C2 (u,subjSTS,fullSC,rstUid,rstrUid);
   !out anySCClient(fullSC))).
```

Fig. 4. Client process for WS-Trust

consisting of the issuing principal's name u, and the correspondence assertion C1. At this point the client process sends the RST message on the public channel soap, and receives the RSTR response msgrstr on the same channel. This response is processed by calling isEntropicRSTREnvelope, which extracts the message identifier rstrUid, the security context identifier sctid, and the STS's entropy stsEntropy. Then, the client process calls the predicate fillSC to construct the full security context fullSC from partialSC, sctid, and the new sckey associated with the security context, computed as psha1(base64(clientEntropy),stsEntropy). Finally, it issues an event **end** C2, with the user and STS principal names and correspondence assertion C2, and adds fullSC to the trusted security context database channel anySCClient.

—In case entropyMode = "Server", indicating non-entropic mode, the client process

calls the predicate mkNonEntropicRSTEnvelope that constructs an RST message msgrst with no client entropy. In non-entropic mode, the STS certificate is not authenticated as part of the RST message; it is only authenticated in the RSTR message. Hence, in this mode, the correspondence assertion is C1 = (remCertSC,rstUid,utf8("zero")), where remCertSC is computed from partialSC by erasing the STS certificate in StsInfo and the clientEntropy is set to "zero".) The rest of the process is similar to the one for the entropic case, performing the RST/RSTR exchange and processing the received msgrstr using isNonEntropicRSTREnvelope, then constructing a full security context fullSC, and issuing the **end** C2 event before adding fullSC to the trusted security context database.

The structure of the STS process is similar. It receives an RST message first and after processing it, it reconstructs partialSC, rstUid, and clientEntropy (if any). It then issues the appropriate **end** C1 event. Depending on the entropic mode in partialSC, it then constructs an RSTR message msgrstr in either entropic or non-entropic mode, issues the appropriate **begin** C2 event, and sends msgrstr back to the client.

4.3 Processing the RST Envelope

In our exchange, the SOAP envelope that carries the RST has a header consisting of a message identifier, <To> and <Action> elements that designate an STS for issuing an SCT, and a <Security> element that itself consists of a timestamp, a token identifying the client, and a digital signature. The structure of the envelope is expressed as a predicate:

```
predicate envRST(env,rst:item,uriSTS,id,t:string,Sig,BaseToken:item) :−
  env = <Envelope>
          <Header>
            <MessageId>id</>
            <To>uriSTS</>
            <Action>"RSTSCT"</>
            <Security>
              <Timestamp><Created>t</></>
              BaseToken
              Sig</></>
          <Body>rst</></>.
```

The client uses this predicate (and others) to assemble an RST envelope; conversely, as an early step in its processing, the STS uses this predicate to check that a received envelope complies with this structure. The full processing for the RST envelope at the receiving STS is coded by the predicate:

```
predicate isRSTEnvelope(msgrst:item,kr:bytes,U,Sts:item,
                        partialSC,rstUid:item,clientEntropy:bytes) :−
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken),
  EntropicRST(rst,srvURI,etok,ExtraInfo),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isAsymEncryptedKey(etok,clientEntropy,sx),
  isX509Token(BaseToken,kr,u,a,ek),
  isSignature(sig1,"rsasha1",ek,
        [<Body>rst</><To>uriSTS</><Action>"RSTSCT"</>
          <MessageId>id1</><Created>t1</>]),
  uid(rstUid,id1,t1),
```

PartialSctSC(partialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo).

Here, we give the predicate clause used to check an entropic RST signed using a user's X.509 public-key certificate. The script contains similar clauses for checking the other cases, and also defines a symmetric predicate for preparing RST envelopes on the client side.

The predicate takes as input the received envelope (msgrst) and checks it using the public key of the CA (kr) and the principal records (U,STS) for the user and the STS. It returns as output the proposed security context partialSC, the unique identifier rstUid, and the received clientEntropy.

To this end, the predicate first calls envRST to parse msgrst and extract the rst, the relevant header fields, the message signature sig1, and the user's authenticating BaseToken. It then calls isEntropicRST to parse the rst and retrieve etok, which contains the encrypted clientEntropy, checking that it contains a fragment URI BaseTokenId pointing to the user's BaseToken. The predicate isSTS checks that the STS record Sts has a uriSTS that matches the <To> header of the RST, and extracts the certificate certSTS and private key sx corresponding to the STS. This private key is used to decrypt etok and retrieve the clientEntropy. Then, the predicate isX509TokenPub extracts the user's public key from BaseToken and the predicate isSignature checks that sig1 is a valid user signature that covers the message body and all the parsed header elements. Finally, the predicates uid and PartialSctSC construct the outputs rstUid and partialSC (to be included in C1).

4.4 Processing the RSTR Envelope

The envelope carrying the RSTR has a similar structure to the RST envelope; it is expressed in the following predicate:

```
predicate envRSTR(msgrstr:item,rstr:item,id2:string,t2:string,
                  STSToken:item,sig2:item,rto:string) :−
  msgrstr = <Envelope>
              <Header>
                <MessageId>id2</><RelatesTo>rto</><Action>"RSTRSCT"</>
                <Security>
                  <Timestamp><Created>t2</></>
                  STSToken
                  sig2</></>
              <Body>rstr</></>.
```

The main difference is that the <To> header is replaced with a <RelatesTo> header that contains the message identifier of the RST being responded to.

The creation of RSTRs at the service and the corresponding checks at the client are also expressed as symmetric predicates. We detail the predicate used by the client to check entropic RSTRs issued for X.509 user principals:

```
predicate isEntropicRSTREnvelope(msgrstr:item,kr:bytes,partialSC:item,
                                 rstUid:item,clientEntropy:bytes,
                                 rstrUid:item,sctid:string,stsEntropy:bytes) :−
  PartialSctSC(partialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,rto,t1),
  isSTSPubInfo(StsInfo,uriSTS,certSTS),
  envRSTR(msgrstr,rstr,id2,t2,STSToken,sig2,rto),
```

```
EntropicRSTR(rstr,srvURI,BaseToken,uriSTS,sctid,etok),
isEncryptedKey(etok,stsEntropy,clientEntropy),
x509Token(STSToken,certSTS),
isX509Cert(certSTS,kr,subjSTS,"rsasha1",ek),
isSignature(sig2,"rsasha1",ek,
            [<Body>rstr</>
            <RelatesTo>rto</>
            <Action>"RSTRSCT"</>
            <MessageId>id2</>
            <Created>t2</>]),
uid(rstrUid,id2,t2).
```

This predicate takes as input a received RSTR envelope, msgrstr, along with kr and the partialSC, rstUid, and clientEntropy previously sent in the RST envelope. It returns the unique identifier rstrUid, and the new sctid and stsEntropy generated by the server.

To this end, the predicate uses clauses similar to those of isRSTEnvelope. First, PartialSctSC and uid parse partialSC and rstUid into their respective components, and isSTSPubInfo extracts the URI and certificate of the STS. Then, envRSTR parses the envelope, checks that the message identifier in its `<RelatesTo>` header matches the identifier rto used in the RST envelope, and extracts rstr, STSToken and other headers. EntropicRSTR parses rstr and checks that it was issued by the STS at uriSTS, for the user principal identified in BaseToken, and for the service at srvURI; it extracts the sctid and the stsEntropy encrypted within etok. Next, isEncryptedKey decrypts etok (using clientEntropy as decryption key) and yields stsEntropy. The following three clauses retrieve the STS verification key from the certSTS included in STSToken and check that sig2 is a valid STS signature that covers the rstr and the parsed header elements. Finally, uid constructs the output rstrUid from the message identifier and timestamp (to be included in C2).

4.5 Authentication and Secrecy Results

As described in Section 2, our full TulaFale script models our system as an explicit pi calculus process consisting of an unbounded number of clients and STSs running in parallel and willing to communicate over a public channel, under the control of the attacker, for any choice of operation modes.

Although the opponent is powerful, our theorems assert that the RST/RSTR exchange preserves our authentication, correlation, and secrecy goals. The authentication and correlation goals are stated in terms of the correspondence between **begin** and **end** events generated by client and STS processes; the attacker cannot generate events.

THEOREM 1 (ROBUST SAFETY OF C1, C2). *For all runs of script* `ssws-trust.tf` *in the presence of an active attacker, we have:*

—*For each* **end** C1 *with a safe client, there is a matching* **begin** C1.

—*For each* **end** C2 *with safe client and STS server, there is a matching* **begin** C2.

Hence, the exchange guarantees that any RST envelope from a safe client, accepted by a safe STS, and used to allocate a security context actually corresponds to a genuine request with matching parameters.

Secrecy is stated in terms of the attacker's knowledge of the established session key.

THEOREM 2 (SESSION-KEY SECRECY). *For all runs of script* `ssws-trust.tf` *in the presence of an active attacker, for each* **begin** C2 *with safe client and STS server, the* Key *element recorded in* SC *remains secret.*

Hence, even if the service immediately uses the SC key to encrypt messages, at most the client who signed the RST may decrypt those messages. Combining the two theorems, this is also the case once the client issues a matching **end** C2 and starts using the SC key. In addition, we have checked that the result holds even if, in entropic mode (that is, where both client and server provide entropy), one of the participants uses a value selected by the attacker instead of a fresh value as its entropy.

As a corollary of Theorems 1 and 2, if both the client and the STS are safe and the exchange completes, then the two parties agree on an SC containing a shared, secret key.

These results are automatically proved by running TulaFale on script `ssws-trust.tf`. In addition to security properties, we also check a series of basic functional properties, checking for instance that the protocol can successfully complete for each choice of mode and safe principals.

4.6 Cautionary Notes

As discussed in Section 3.3, properly checking (and correlating) the contents of the RST and RSTR is critical for reaching our expected agreements. To conclude this section, we illustrate a few vulnerabilities that occur in variants of our model featuring weaker enforcement mechanisms.

Suppose that the RSTR does not include the client identity (that is, consider omitting BaseToken from predicate EntropicRSTR in Section 3.2) and the attacker has obtained the private key of an unsafe user, say E. When another (safe) user C sends a signed RST, E can intercept the envelope, possibly modify its content (for instance ExtraInfo), and substitute E's certificate and valid signature for C's. The STS accepts the modified message as a valid RST from E, records E's identity and ExtraInfo in a new SC, and sends back a signed RSTR to E. The attacker forwards the RSTR envelope to C, who accepts it as a valid reply to its original request. Subsequently, messages sent by C to the service will be accepted and mis-attributed to E. Even if the client insists on using entropic mode, the attack persists as long as the entropy is encrypted for the STS before being signed as part of the RST, since the attacker can blindly resign the encrypted entropy.

Similarly, if the RSTR omits uriSTS, we lose agreement on the URI of the STS. This may become problematic in case several (safe) STSs sign RSTRs using the same certificate but enforce different authorization policies.

This difficulty in correlating RST and RSTR messages is a particular instance of a general problem: the current WS-Security standard [Nadalin et al. 2004] does not prescribe any general mechanism to correlate requests and responses securely. One solution, *signature confirmation* (described in a draft revision of the WS-Security standard) involves echoing and signing parts of the signature of the request in the response message. Another approach, specific to WS-Trust, involves returning alongside the RSTR an "authenticator" hash of the contents of the RST and RSTR.

5. WEB SERVICES SECURE CONVERSATION LANGUAGE

WS-SecureConversation "defines mechanisms for establishing and sharing security contexts, and deriving keys from established security contexts (or any shared secret)" in order

to secure series of messages [Kaler et al. 2004a]. We survey the specification, and in particular focus on the new security tokens it introduces.

5.1 Tokens for Security Contexts and Key Derivations

WS-SecureConversation introduces two new kinds of security token: SCTs and DKTs.

A *security context token* (SCT) in the security header of an envelope represents (an abstract pointer to) a shared security context (SC), typically established using an RST/RSTR exchange, as described in previous sections. The SCT simply incorporates a *context identifier*, so that the recipient can access the relevant security context, notably the authenticated identity of the sender. Local references to the SCT can appear in the envelope whenever a symmetric key is needed, to indicate that the recipient should read the key from the SC. In our scripts, we use the following structural predicate for SCTs:

```
predicate SCT(sct:item,sctid:string):−
  sct = <SecurityContextToken><Identifier>sctid</></>.
```

A *derived key token* (DKT) provides a reference to a master key, an algorithm, and additional parameters to compute a separate key. For instance, a typical DKT incorporates a fresh nonce and a reference to an SCT, and thereby indicates that the recipient should compute a derived key as the hash of that nonce keyed with the SC key. Such DKTs may be used to secure independent requests relying on the same SC, or to derive distinct keys for encryption and for authentication.

In our scripts, we use the structural predicate DKSCT to decompose DKTs that refer to SCTs and the predicate deriveKey to compute the associated key:

```
predicate DKSCT(dksct:item,sctid:string,nonce:bytes):−
  dksct = <DerivedKeyToken>
            <SecurityTokenReference>
              <Reference>sctid</>
              <valueType>"SCT"</></>
            <Nonce>base64(nonce)</></>.

predicate deriveKey(dk:bytes,key:bytes,nonce:bytes):−
  dk = psha1(base64(key),concat(utf8("WSSecureConversation"),nonce)).
```

In general, the parent token need not be an SCT; instead one can use, for example, a Kerberos token, or even another DKT. WS-SecureConversation also supports other variants of key derivation, a lightweight derived-key mechanism that provides the same functionality as a DKT within a key reference, and some SCT propagation and amendment mechanisms. We do not model these advanced mechanisms in this paper.

5.2 Security Considerations

As advocated in Section 3.3, one should carefully review the intent and usage of security contexts, especially when they are used to derive authentication materials. Otherwise, a weak (or unauthenticated) agreement may for instance lead to valid signatures that are interpreted differently by the sender and the receiver.

Unlike fixed protocols, key selection and derivation are dynamically driven by the tokens included in the envelope; these elements are often unauthenticated or used before authentication. Thus, even if the recipient can successfully decrypt or validate a signature using the derived key suggested in the envelope, it is equally important for this recipient to

check that the key is derived from an adequate security token. If the key is derived from an SCT, for instance, this may involve comparing the target URI and apparent sender of the envelope to the `<AppliesTo>` and BaseToken recorded in the SC.

Despite the terminology, the uniqueness (or freshness) of received SC identifiers and derived key nonces should not be taken for granted, especially when they are passed in the clear. For instance, a hostile service may eavesdrop an SCT and initiate its own sessions with the same identifier; this may invalidate the security context, or lead to confusion about its contents.

Finally, although WS-SecureConversation promotes the derivation of a separate key for each purpose, more efficient keying mechanisms are often available. In the absence of knowledge of the usage of the keys, it is prudent to generate a fresh key systematically, as this may prevent interference between cryptographic algorithms. Nonetheless, for signing a sequence of messages sent by a single client, for example, a key implicitly derived from a hash of the session identifier and sequence number is more efficient than a key derived from a random nonce that additionally signs these two elements. Besides, for common encryption algorithms, a random initialization vector can play a role similar to the nonce in a derived key, at a fraction of the cost.

6. MODELLING AND VERIFYING USES OF WS-SECURECONVERSATION

Continuing with the example of Figure 1, we consider exchanges between a client and a service following the completion of an RST/RSTR exchange, as modelled in Section 4. The security goals of these exchanges are to achieve mutual authentication between client and service, to ensure message correlation between requests and responses, and to preserve the secrecy of all message bodies. We first model a single request/response exchange, before generalizing our results to "open-ended conversations" comprising arbitrary sequences of exchanges.

A typical run of the protocol is depicted in Figure 5. It involves a process Client that sends a request to a web service using an existing security context and waits for a response, and a process Service that handles such requests, for some given address and SOAP action (srvURI,srvAC).

To reduce the complexity of automated verification, when relying on a security context to protect messages, we use an idealized STS that assumes agreement on the security context, rather than the one analyzed in Section 4. Similarly to the processes that distribute authentication materials discussed in Section 2.3, the idealized STS creates security contexts and communicates them to the client and the service using private channels. It may also leak some of these security contexts (including their keys), as a way to model shared-key compromises. Our security theorems are relative to *safe security contexts*, that is, security contexts whose keys have not been leaked to the attacker by the STS.

6.1 Mapping Principals to TulaFale Processes

When considering each envelope in this protocol, we use an abstract parameter, DestInfo, to represent the concatenation of some WS-Addressing [Box et al. 2004] headers included in the envelope.

First, the client inputs from the attacker a Request envelope that provides a security context identifier sctid, a timestamp t, and target service information srvURI and srvAC. The client then fetches from the SC database a security context that matches sctid, if any, and extends the Request envelope by adding a fresh message identifier and a secret request

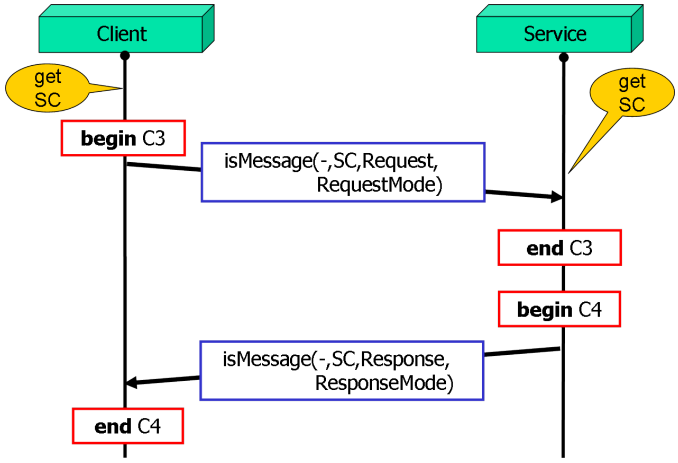

Fig. 5. A single request/response exchange

body. (Hence, Request is an envelope with some DestInfo that includes headers containing the request message identifier and target service information.)

In addition, the attacker can also choose between two operation modes: either securing the request with the shared SC key, or securing it with fresh keys derived from the SC key. These key-derivation details are recorded in an element, RequestMode, which contains either two nonces used to derive keys for encryption and signature, or a constant indicating that the SC key is directly used.

As in Section 4, our processes issue events to indicate their intent: before sending the request, the client emits **begin** C3; after receiving the request and checking its validity, the web service records the acceptance by emitting **end** C3. These events record the following data:

$$C3 = (SC,Request,RequestMode)$$

After accepting a request from a client, the service similarly prepares a Response containing a response body and some addressing headers: DestInfo now includes headers echoing the server address srvURI and the request identifier, plus a header containing a fresh response identifier. For simplicity, the service uses the same operation mode as the client: if the request used derived keys, so does the response. The corresponding key derivation details are recorded in ResponseMode.

Before sending its response, the service emits **begin** C4. After checking the validity of the response, the client emits **end** C4. These events record data for both the request and the response:

$$C4 = (C3,Response,ResponseMode)$$

where C3 includes data on the request, and Response and ResponseMode include data on the response.

Next, we describe the structure and processing of envelopes that effectively protect these requests and responses.

6.2 Processing Request and Response Envelopes

Since the request and response envelopes are processed similarly, we use generic predicates for both purposes. When using derived keys, the structure of these SOAP envelopes is given by the predicate:

```
predicate isEnv(Env:item,DestInfo:items,t:string,sig,ebody:item,
                sctid:string,mode:item) :−
  Env = <Envelope>
          <Header>
            <Security>
              <Timestamp><Created>t</></>
              sct dksctEnc dksctSig
              sig</> @
            DestInfo </>
          <Body>ebody</></>,
  SCT(sct,sctid),
  DKSCT(dksctEnc,sctid,EncNonce), DKSCT(dksctSig,sctid,SigNonce),
  derivedKeyMode(mode,EncNonce,SigNonce).
```

The structure of the envelope differs from those of Section 4.3 in three ways:

—the envelope includes a security context token (sct) and two derived key tokens (dksctEnc and dksctSig) used to indicate keys for encryption and signing;

—the envelope includes a generic parameter (DestInfo) that provides headers specific to requests and responses;

—the envelope includes an encrypted body (ebody).

After setting the structure of the envelope, the isEnv predicate inspects the SCT and DKTs, in order to return an SC identifier (sctid) and a mode descriptor including the two nonces used for key derivation (EncNonce and SigNonce).

We now give a predicate used for validating incoming envelopes: both requests for the service and responses for the client.

```
predicate isMessage(Env,SC,EnvelopeInfo,mode:item) :−
  isEnv(Env,DestInfo,t,sig,ebody,sctid,mode),
  sctSC(SC,sctid,sckey,entropyMode,UserToken,StsInfo,appTo,extra),
  computeKeys(mode,sckey,EncKey,SigKey),
  isEncryptedData(ebody,body,EncKey),
  body = <Body>b</>,
  EnvInfo(EnvelopeInfo,t,sctid,DestInfo,b),
  isSignature(sig,"hmacsha1",SigKey,
              [<Body>ebody</> <Created>t</> @ DestInfo ]).
```

In the predicate, isEnv parses the envelope, extracting DestInfo and other sub-elements.

When processing a request, DestInfo contains information about the web service URI and desired action, along with a message identifier. This is checked by the predicate:

```
predicate destInfoReq(DestInfo:items,srvURI:item,srvAC:item,id1:string):−
  DestInfo = [<To>srvURI</> <Action>srvAC</> <MessageId>id1</>].
```

When processing a response, DestInfo contains the web service URI as source information, plus a message identifier and the original request message identifier, used for correlation.

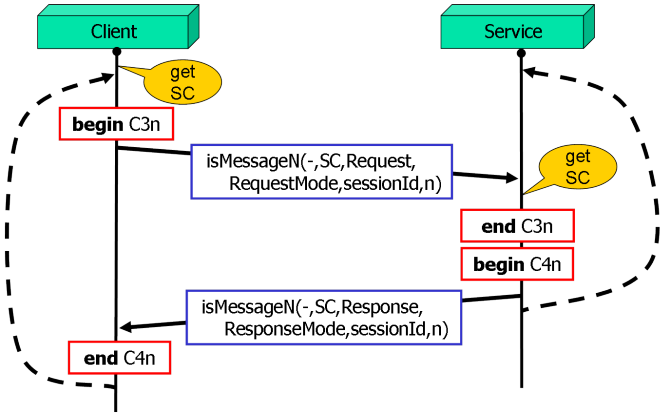

Fig. 6. Iterating exchanges

predicate destInfoResp(DestInfoResponse:**items**,srvURI:**item**,id1:**string**,id2:**string**):−
DestInfoResponse = [<From>srvURI</> <RelatesTo>id1</> <MessageId>id2</>].

The call to isEnv also extracts timestamp t, signature sig an encrypted body ebody, security context identifier sctid and operation mode.

Continuing with the explanation of isMessage, the predicate sctSC checks sctid against the security context SC and then retrieves the security context key sckey. Predicate computeKeys uses that key and the two nonces passed in mode to compute the keys EncKey and SigKey protecting the envelope, as explained in Section 5.1. EncKey is then used to decrypt the message body, whereas SignKey is used to verify a signature binding the encrypted message body, a timestamp, and the addressing headers. Finally, information extracted from the envelope is returned in EnvelopeInfo.

6.3 Authentication and Secrecy Results

The following theorem establishes the agreement, message correlation, and secrecy properties for the exchange described above. Its proof is obtained by running TulaFale.

THEOREM 3 (ROBUST SAFETY OF C3, C4 AND SECRECY). *For all runs of script* `ssws-secconv.tf` *in the presence of an active attacker, we have:*

—*For each* **end** C3 *with a safe security context, there is a matching* **begin** C3.

—*For each* **end** C4 *with a safe security context, there is a matching* **begin** C4.

—*For each exchange with a safe security context, the request and response bodies remain secret.*

6.4 Open-Ended Conversations

We now extend our protocol to allow clients and services to iterate their exchanges—as suggested by the dashed lines of Figure 6—thus modelling a more substantial conversation. For simplicity, we fix the operation mode and always use derived keys.

Each session is identified by a sessionId string, freshly generated by the client before sending its first request. Within the session, each request is indexed by a sequence number. To this end, we (mostly) comply with the syntax of WS-ReliableMessaging [Ferris

et al. 2004] and use its simple request acknowledgement mechanism: requests carry a `<Sequence>` header, including the sessionId and a message number n, set to zero by the client in the first request, and incremented by one in every subsequent request. The structure of this header is given by the predicate:

predicate sequence(Sequence:**item**,sessionId,n:**string**):−
  Sequence = `<Sequence>`
    `<Identifier>`sessionId`</>`
    `<MessageNumber>`n`</></>`.

Similarly, responses carry a `<SequenceAcknowledgement>` header echoing the received sessionId and message number.

To specify an agreement on the conversation as a whole, our client and service collect detailed information in events, as follows. For the nth request and response, respectively, C3n and C4n record as history H not only the envelope just sent (Req_n for **begin** events) or accepted (Resp_n for **end** events), but also the preceding sequence S=[Resp_{n-1} $\text{Req}_{n-1} \cdots \text{Resp}_0$ Req_0] of all previously processed envelopes for the session. Thus, for C3n, the recorded history is H=[Req_n@ S] while for C4n the recorded history is H'=[Resp_n Req_n@ S]. C3n and C4n also record the shared session identifier sessionId, the message number n of the last exchange, and the security context SC (which provides client and service identification in particular).

$$\text{C3n} = (\text{SC,sessionId,H,n})$$
$$\text{C4n} = (\text{SC,sessionId,H',n})$$

To establish the correspondences, we use a script that protects the service from replays of initial requests with identical session identifiers. (This is necessary because the server does not contribute to the generation of the session identifier, and thus could be lead to run several sessions for a single client session.)

THEOREM 4 (ROBUST SAFETY OF C3N, C4N AND SECRECY). *For all runs of script* `ssws-secrm.tf` *in the presence of an active attacker, we have:*

—*For each* **end** C3n *with a safe security context, there is a matching* **begin** C3n.

—*For each* **end** C4n *with a safe security context, there is a matching* **begin** C4n.

—*All request and response bodies protected by a safe security context remain secret.*

The proof is shown in Appendix A; it uses the script `ssws-secrm.tf` for the iterated protocol. It also relies on a similar, but slightly more abstract script, `ssws-secrm-a.tf`, in which sequencing is also controlled by the environment. We use ProVerif on both scripts to establish a series of correspondences. We then manually combine these properties by reasoning on the structure of these scripts, relying on standard proof techniques for the pi calculus [Abadi and Fournet 2001; Bhargavan et al. 2005].

6.5 Cautionary Notes

As in Section 4.6, we mention some attacks observed as we modelled weaker variants of the protocol.

Upon receiving a message secured using SCTs, it is important to attribute the message to the correct sending principal, typically by verifying that the message is signed and encrypted using keys derived from the same SC. Conversely, envelopes with multiple SCTs

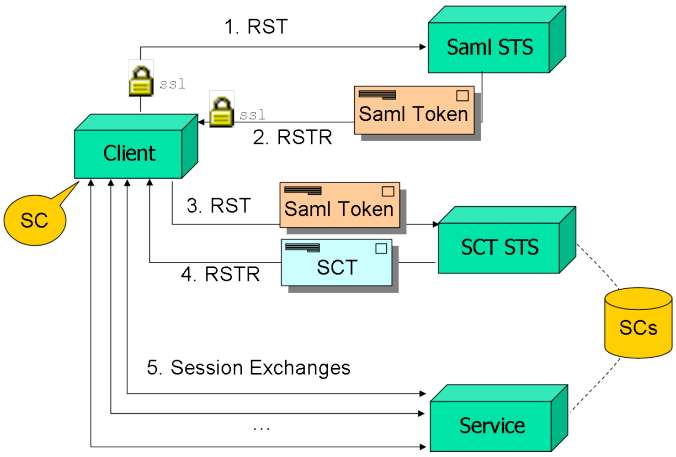

Fig. 7. The WS-Trust/WS-SecureConversation Interoperability Scenario

are often problematic. Consider, for instance, a web server that attributes the message to the principal associated with the first SCT in the security header. Then, an attacker can intercept a message protected (that is, signed or encrypted) using an SCT, and can rewrite it by inserting a different SCT at the beginning of the security header. Assuming both SCTs are associated with valid SCs for the same service but for different clients, the service will accept the rewritten request and attribute it to the wrong client. Consider now a web server that attributes the message to the principal associated with the encrypting SCT, but accepts messages signed with a different SCT. Then, if the attacker knows the secret stored in an (unsafe) SC, it can impersonate any sender using a (safe) SC, by intercepting, modifying and re-signing its messages.

Concerning open-ended conversations using WS-ReliableMessaging, as illustrated in Section 6.4, the current specification makes it necessary to enforce replay protection for initial client requests. This contrasts with a common pattern in protocols, whereby the first request has no persistent effect on the server, and thus replays are considered harmless in the first exchange (see for instance Abadi et al. [2004]). Besides, replay protection is hard to implement for web server farms, where several servers share the same SOAP address and STS. (For an example of a web server farm managing several sessions with WS-SecureConversation, see Chris Keyser's work [2004].)

Without replay protection, a subtle "session replication" attack may appear when the same initial client request can be accepted several times. Starting from the first request, an attacker can systematically replicate each request towards multiple servers, forward one selected response to the client, and discard the other responses. As soon as some of the responses differ, some server will accept requests that do not correspond to its previous responses, and the client will receive inconsistent responses.

7. APPLICATION: INTEROPERABILITY SCENARIOS

We finally consider the working scenario described in the WS-Trust/WS-SecureConversation interoperability workshop [WS 2004]. In contrast to the specifications, this scenario provides a complete protocol, used as a common test case for comparing implementations

from six different vendors.

The scenario is outlined in Figure 7; it involves four roles—a client, a SAML server [OASIS Security Services TC 2005], an STS server, and a web service—and a series of exchanges initiated by the client:

—A first RST/RSTR exchange with the SAML server, to obtain a token. This exchange is protected at the transport layer, using SSL.

—A second RST/RSTR exchange with the STS server. This exchange is protected using WS-Security, relying on the SAML token. The outcome of the exchange is a security context shared between the client and the target service.

—One or several exchanges with the service, protected using WS-Security and WS-SecureConversation, and relying on the shared SC.

As a formal counterpart to testing, we verify a series of authentication properties in this scenario by applying the TulaFale models developed in the previous sections. We reuse the processes and predicates of Section 3 to represent the two RST/RSTR exchanges, and those of Section 5 to represent the subsequent session exchanges. As in Section 6, we rely on an idealized SCT token server for analyzing the session exchanges.

We spent only a few days to model and verify this scenario (including extensions of our library to support SSL and SAML as required in the scenario). We believe this verification effort is comparable to the coding and testing of the scenario for an existing target implementation of web services. We found several flaws in earlier versions of the scenario [WS 2002]—although not in its latest version. Moreover, our specification of target security properties for the scenario may also be of interest for testing.

7.1 Simple Models for SSL and SAML

The first RST/RSTR exchange of the scenario is similar to the exchange presented in Section 4, except that a SAML token is exchanged instead of an SCT and that the two messages are protected using SSL. We model SAML tokens and SSL only to the extent required by this exchange. These models are thus simpler and more abstract than those of Sections 4 and 6.

The SAML token provides client authentication and carries a shared key. The following predicate builds a token s that carries a key k to be shared between a client with subject name u and an STS server with public key ka; the token is signed using the private key sT of the SAML server. (The computation of the shared key k from secret values provided by the client and the SAML server is coded as in Section 4.)

```
predicate Saml(s:item,sT:bytes,u:string,ka,k:bytes) :−
  mkAsymEncryptedKey(ekey,k,ka),
  auth = <AuthenticationStatement>
           <Subject><NameIdentifier>u</></>
           <SubjectConfirmation><KeyInfo>ekey</></></>,
  mkSignature(sig,"rsasha1",sT,[<SamlAssertion>auth</>]),
  s = <SamlAssertion>auth sig</>.
```

First, mkAsymEncryptedKey encrypts k under the public key of the target STS server ka. Then, an authentication statement auth is built from the encrypted key ekey and the client name u; the statement is signed using the private key of the SAML server. Finally, the SAML token s consists of the auth item and the corresponding signature.

SSL [Freier et al. 1996] can protect SOAP envelopes as a whole, irrespective of their contents (including any SOAP-level principal identifier or password mentioned within an envelope). In the interoperability scenario, SSL connects an anonymous client to a server identified by its public-key certificate. We model this connection simply by having the client generate a symmetric key symk, pass it encrypted under the SAML-server public key, and use that key for encrypting the first RST and RSTR envelopes. (SSL provides additional security guarantees, which are irrelevant in our case.) We detail the processing of the first message by the SAML server. The server retrieves the RST from the encrypted envelope msg and encrypted key ek, using the predicate below.

predicate isSSLRequest(msg,kr,ek,symk:**bytes**,env,S,serverToken:**item**) :−
  isX509(S,_,cert,sT,kr),
  x509Token(serverToken,cert),
  c14n(<SSLKey>base64(symk)</>) = decrsa(sT,ek),
  c14n(<SSLRequest>env</>) = decaes(symk,msg).

The SAML server has an X.509 certificate cert with a corresponding private key sT. Using sT, the server decrypts first ek to obtain symk, then msg to obtain env.

7.2 Authentication Results

As in the previous sections, any number of principals may be involved in parallel runs of the scenario, and they systematically record their view of the run using correspondences. For a given run of the protocol, the client records **begin** I1 before sending the first RST; the SAML server records **end** I1 after receiving and accepting a first RST, then records **begin** I2 before sending back the first RSTR; and so on until, finally, the client records **end** I6 after receiving and accepting the service response. The content of the six correspondences records the intended authentication properties of this scenario, as outlined below:

I1 = (partialSamlSC)	I2 = (SamlSC)
I3 = (partialSctSC)	I4 = (SctSC)
I5 = (SctSC,Request)	I6 = (SctSC,Request,Response)

(We refer to the scripts for a complete specification.) In these correspondences, the various SC elements record agreements at each stage of the protocol. The data agreed in these correspondences extends that described in Section 3.3; it includes for instance a tag that indicates whether SSL or WS-Security is used to secure the agreement, and additional details on the keys and the issued tokens. In particular, the name of the client principal is recorded as part of the Base token in all these SCs: first in a username token in partialSamlSC and SamlSC; then in the SAML token associated with SamlSC in partialSctSC and SctSC. The URI of the STS server is also recorded: first as the target AppliesTo field in partialSamlSC and SamlSC, then in STSInfo in partialSctSC and SctSC. Finally, Request and Response record authenticated envelope contents and their correlation, as described in Section 6.

For this section only, we adapt our notion of client safety as follows: a client principal is unsafe when (1) any of its secrets has been leaked to the attacker, or (2) it has contacted any unsafe SAML server. Hence, client safety now also depends on SAML server safety. (Using TulaFale, we easily check that our properties do not hold if, instead, we request only that the SAML server contacted for the session be safe, as the client may have sent its password to another, unsafe SAML server during a prior session. This is a well-known shortcoming of password-based authentication on top of SSL.)

The script modelling the two WS-Trust RST/RSTR exchanges is `wstrust-secconv-interop1-4.tf`; the script modelling the subsequent WS-SecureConversation session exchanges is `wstrust-secconv-interop5-6.tf`. By running TulaFale, we establish the following properties:

THEOREM 5 (ROBUST SAFETY OF I1–I6). *For all runs of the scripts in the presence of an active attacker, we have:*

—*For each* **end** I1 *with a safe client, there is a matching* **begin** I1.

—*For each* **end** I2 *with a safe client, there is a matching* **begin** I2.

—*For each* **end** I3 *with safe client and STS server, there is a matching* **begin** I3.

—*For each* **end** I4 *with safe client and STS server, there is a matching* **begin** I4.

—*For each* **end** I5 *with a safe security context* SctSC*, there is a matching* **begin** I5.

—*For each* **end** I6 *with a safe security context* SctSC*, there is a matching* **begin** I6.

8. RELATED WORK

Our work on SOAP-based cryptographic protocols relies on treating cryptographic transforms as operators in an abstract algebra of messages. This symbolic model of cryptography was first explored by Dolev and Yao [1983]. A substantial body of research has subsequently developed proof techniques for this model, in the context of many formalisms, including various logics and theorem provers [Burrows et al. 1989; Kemmerer et al. 1994; Paulson 1998; Cohen 2000; Durgin et al. 2003], process algebras and calculi [Ryan et al. 2001; Abadi and Gordon 1999; Abadi and Fournet 2001], and strand spaces [Thayer Fábrega et al. 1999].

There has been a trade off between accuracy and automation during much of the development of proof techniques for this model, For example, the inductive method of Paulson [1998] supports proofs of correctness for a realistic threat model, with unbounded numbers of participants and parallel sessions, and compromise of keys, but at the cost of lengthy sessions with the interactive theorem prover Isabelle. Slightly earlier, Lowe [1996] uses the FDR model checker to find attacks automatically on protocol models with low bounds on the numbers or participants and sessions; still, the absence of attacks on bounded models does not in general imply the strong results provable with Isabelle.

More recently, several tools, such as TAPS [Cohen 2000] and ProVerif [Blanchet 2001; 2002], have been developed to prove security properties automatically with respect to a realistic, unbounded threat model. Both TAPS and ProVerif model protocols within first-order logic, and rely upon a resolution theorem prover. Additionally, ProVerif can prove properties of protocols expressed in a process calculus, via a logical semantics of processes.

Tools such as these have allowed studies of substantial key exchange protocols. We mention two examples. Paulson [1999] shows authentication, secrecy, and integrity properties of a model of the SSL/TLS protocol with the interactive prover Isabelle; Paulson's was the first formal study of SSL/TLS to put no finite bounds on the numbers of principals or concurrent sessions—an assumption made in earlier approaches using model-checking. More recently, Abadi et al. [2004] show various security properties of the JFK key establishment protocol using ProVerif. ProVerif needs little user interaction compared to Isabelle, and also supports unbounded models of the protocol.

TulaFale uses ProVerif as a basis for the automated analysis of XML based protocols. It seems likely that TulaFale could be adapted to rely on other tools, such as TAPS, but this

has not been attempted.

To summarize, there has been substantial recent progress in formalisms and tools for the Dolev-Yao model, but protocols of the level of complexity considered here have only recently come within the reach of formal methods. Hundreds, at least, of symbolic protocol models have now been analyzed automatically by tools such as TAPS and ProVerif. Still, at the time our work was first published [Bhargavan et al. 2004a] the models generated by the TulaFale tool from the scripts described in this paper were the largest, to the best of our knowledge, on the basis of numbers of lines of code or logical clauses, and set a new high-water mark in terms of complexity.

Aside from the symbolic view of cryptography, as pioneered by Dolev and Yao, there has been much research on a computational view, based on computational complexity and probability theory rather than abstract algebra. Proofs in symbolic models are easier to automate than in computational models, while computational models make fewer assumptions. There has been substantial recent progress in developing precise relationships between the two views, such as the work by Abadi and Rogaway [2002] and Backes et al. [2003]. Research in this area is now leading to automated tools for checking properties within computational models [Blanchet 2006]. Applying such tools to web services protocols is an intriguing direction for future research.

There are by now many implementations of SOAP and XML security, but there is comparatively little work on formalizing the resulting security properties. Damiani et al. [2002] show access control properties for SOAP-based web services, relying on an underlying secure channel such as SSL/TLS. Gordon and Pucella [2005] prove authentication properties of SOAP-based security protocols, but do not consider key establishment and do not model XML syntax in detail. Kleiner and Roscoe [2004] construct abstract descriptions of some simple WS-Security protocols from XML message sequences, so as to analyze the abstractions with the FDR model checker. In a recent study, Kleiner and Roscoe [2005] also apply FDR to analyze the WSE implementation of the WS-Trust/WS-SecureConversation protocol, and independently find some of the vulnerabilities described in this paper. Their use of simplifying transformations [Hui and Lowe 2001] leads to succinct descriptions of the protocols and potential attacks.

Groß and Pfitzmann [2004] analyze a protocol [Kaler et al. 2003b] in which a web browser contacts a federated identity provider to sign into a website. The protocol relies on the SAML [OASIS Security Services TC 2005] and WS-Federation [Kaler et al. 2003a] specifications. Groß and Pfitzmann's analysis consists of rigorous, informal proofs.

Recent work by Backes et al. [2006] analyzes a protocol [Davis et al. 2005] used to test interoperability of implementations of the WS-ReliableMessaging specification [Ferris et al. 2004]. The protocol is similar to the one we study in Section 7 and consists of an initial RST/RSTR exchange using WS-Trust and WS-SecureConversation, and then a series of messages exchanged according to WS-ReliableMessaging. Two separate analyzes are presented: an automated, Dolev-Yao based verification using the OFMC and CL-AtSE frontends of the AVISPA tool [Armando et al. 2005]; and a manual analysis based on a computational model of cryptography. Neither analysis uncovers any vulnerabilities in their models of the protocols, nor any inconsistencies, ambiguities, or omissions in the specifications. Unlike their manual analysis, our work is not founded on a computational model of cryptography; we have not considered how to recast and prove properties of our protocol descriptions within such a model. Their formal model contains no detail of the

precise XML encoding of data; a strength of our approach is that we do represent the XML encoding in full detail, and hence can detect attacks that rely on the details of the structure of XML signatures, for example. Moreover, in contrast to our model, which allows executions with an unbounded number of sessions and exchanges, their formal model imposes several bounds. At most three parallel protocol sessions can be instantiated. A client can start at most two message-sending sequences. There are at most three payload messages per message sequence.

Finally, Johnson et al. [2004] use a model checker to establish properties of a formal model of WS-AtomicTransaction, another specification relating to SOAP-based web services. They do not consider security properties.

9. CONCLUSIONS

Our study complements the ongoing work to author and refine the WS-Trust and WS-SecureConversation specifications, to develop implementations, and to test conformance at interoperability workshops. Our positive results concerning secrecy and authenticity within a formal threat model increase confidence in particular usages of the specifications. On the other hand, our negative results for other usages reveal potential vulnerabilities and suggest corrections or guidance. We believe our formal approach can be an asset to the community during the standardization process, as we can swiftly verify security properties of particular proposals. For example, it took only a few days to adapt the scripts of this paper to model and check properties of the protocol used at the WS-Trust and WS-SecureConversation interoperability workshop [WS 2004].

Finally, this study is a substantial example of automated reasoning about detailed formal descriptions of security protocols, demonstrating the power of the underlying theorem prover, ProVerif, in combination with manual proof techniques. Still, although we endeavoured to ensure agreement between our formal descriptions of message formats and the actual messages exchanged by implementations, our formal guarantees do not directly apply to those implementations. This potential for discrepancy between model and implementation is a familiar problem; we are working to address it in the setting of SOAP and other security protocols by extracting pi calculus models automatically from implementation code [Bhargavan et al. 2006].

Acknowledgement. We acknowledge the help and encouragement of the Web Services Enhancements and Windows Communication Foundation teams at Microsoft. We thank the anonymous referees for their detailed comments.

REFERENCES

ABADI, M., BLANCHET, B., AND FOURNET, C. 2004. Just fast keying in the pi calculus. In *13th European Symposium on Programming (ESOP'04)*. LNCS, vol. 2986. Springer, 340–354.

ABADI, M. AND FOURNET, C. 2001. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. 104–115.

ABADI, M. AND GORDON, A. D. 1999. A calculus for cryptographic protocols: The spi calculus. *Information and Computation 148*, 1–70.

ABADI, M. AND ROGAWAY, P. 2002. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology 15,* 2, 103–127.

ARMANDO, A., BASIN, D., BOICHUT, Y., CHEVALIER, Y., COMPAGNA, L., CUELLAR, J., HANKES DRIELSMA, P., HEÁM, P.-C., MANTOVANI, J., MÖDERSHEIM, S., VON OHEIMB, D., RUSINOWITCH, M., SANTIAGO, J., TURUANI, M., VIGANÒ, L., AND VIGNERON, L. 2005. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *17th International Conference on*

Computer Aided Verification (CAV'05), K. Etessami and S. K. Rajamani, Eds. LNCS, vol. 3576. Springer. Available at http://www.avispa-project.org/publications.html.

BACKES, M., MÖDERSHEIM, S., PFITZMANN, B., AND VIGANÒ, L. 2006. Symbolic and cryptographic analysis of the secure WS-ReliableMessaging scenario. In *Foundations of Software Science and Computation Structures (FOSSACS'06)*. 428–445.

BACKES, M., PFITZMANN, B., , AND WAIDNER, M. 2003. A composable cryptographic library with nested operations. In *10th ACM Conference on Computer and Communications Security (CCS'03)*. 220–230.

BARBIR, A., GOODNER, M., GUDGIN, M., GRANQVIST, H., NADALIN, A., ET AL. 2006. *Web Services Secure Conversation Language (WS-SecureConversation) Oasis Committee Draft Version 0.1.* At http://www.oasis-open.org/committees/download.php/20158/ws-secureconversation-1.3-spec-cd-01.pdf.

BHARGAVAN, K., CORIN, R., FOURNET, C., AND GORDON, A. D. 2004a. Secure sessions for web services. In *2004 ACM Workshop on Secure Web Services (SWS'04)*. 56–66.

BHARGAVAN, K., CORIN, R., FOURNET, C., AND GORDON, A. D. 2004b. Secure sessions for web services. Tech. Rep. MSR-TR-2004-114, Microsoft Research.

BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. 2004. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*. 268–277.

BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. 2005. A semantics for web services authentication. *Theor. Comput. Sci. 340,* 1 (June), 102–153.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND PUCELLA, R. 2004. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*. LNCS, vol. 3188. Springer, 197–222.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2006. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 139–152.

BLANCHET, B. 2001. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, 82–96.

BLANCHET, B. 2002. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*. LNCS, vol. 2477. Springer, 342–359.

BLANCHET, B. 2006. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*. 140–154.

BOX, D., CURBERA, F., ET AL. 2004. *Web Services Addressing (WS-Addressing).* At http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/.

BURROWS, M., ABADI, M., AND NEEDHAM, R. 1989. A logic of authentication. *Proceedings of the Royal Society of London A 426*, 233–271.

COHEN, E. 2000. TAPS: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 144–158.

DAMIANI, E., DE CAPITANI DI VIMERCATI, S., PARABOSCHI, S., AND SAMARATI, P. 2002. Securing SOAP e-services. *International Journal of Information Security 1,* 2, 100–115.

DAVIS, D., FERRIS, C., GAJJALA, V., GAVRYLYUK, K., GUDGIN, M., KALER, C., LANGWORTHY, D., MORONEY, M., NADALIN, A., ROOTS, J., STOREY, T., VISHWANATH, T., , AND WALTER, D. 2005. WS-ReliableMessaging Interop Workshop. At ftp://www6.software.ibm.com/software/developer/library/ws-rmseconscenario.doc.

DIFFIE, W. AND HELLMAN, M. 1976. New directions in cryptography. *IEEE Transactions on Information Theory IT-22,* 6 (Nov.), 644–654.

DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory IT–29,* 2, 198–208.

DURGIN, N. A., MITCHELL, J. C., AND PAVLOVIC, D. 2003. A compositional logic for proving security properties of protocols. *Journal of Computer Security 11,* 4, 677–721.

FERRIS, C., LANGWORTHY, D., ET AL. 2004. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging).* At http://msdn.microsoft.com/ws/2004/03/ws-reliablemessaging/.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. November 1996. The SSL protocol: Version 3.0. http://home.netscape.com/eng/ssl3/draft302.txt.

GOLLMANN, D. 2003. Authentication by correspondence. *IEEE Journal on Selected Areas in Communications 21,* 1, 88–95.

GORDON, A. D. AND PUCELLA, R. 2005. Validating a web service security abstraction by typing. *Formal Aspects of Computing 17*, 277–318.

GROSS, T. AND PFITZMANN, B. 2004. Proving a WS-Federation Passive Requestor profile. In *2004 ACM Workshop on Secure Web Services (SWS)*. ACM Press.

GUDGIN, M. 2004. Using WS-Trust and WS-SecureConversation. *MSDN*. At `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/ws-trustandsecureconv.asp`.

GUDGIN, M., NADALIN, A., ET AL. 2005a. *Web Services Secure Conversation Language (WS-SecureConversation) Version 1.2*. At `http://www.oasis-open.org/committees/download.php/17364/lists.oasis-open.orgarchivesws-sx200512zip00000.zip`.

GUDGIN, M., NADALIN, A., ET AL. 2005b. *Web Services Trust Language (WS-Trust) Version 1.2*. At `http://www.oasis-open.org/committees/download.php/17364/lists.oasis-open.orgarchivesws-sx200512zip00000.zip`.

GUDGIN, M., NADALIN, A., ET AL. 2005c. *Web Services Trust Language (WS-Trust) Version 1.2*. At `http://www.oasis-open.org/committees/download.php/20160/ws-trust-1.3-spec-cd-01.pdf`.

HARKINS, D. AND CARREL, D. 1998. RFC 2409: The Internet Key Exchange (IKE). `http://www.ietf.org/rfc/rfc2409.txt`.

HUI, M. L. AND LOWE, G. 2001. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security 9,* 1/2, 3–46.

JOHNSON, J. E., LANGWORTHY, D. E., LAMPORT, L., AND VOGT, F. H. 2004. Formal specification of a web services protocol. In *1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*. University of Pisa.

KALER, C., NADALIN, A., ET AL. 2003a. *Web Services Federation Language (WS-Federation) Version 1.0*. At `http://msdn.microsoft.com/ws/2003/07/ws-federation/`.

KALER, C., NADALIN, A., ET AL. 2003b. *WS-Federation: Passive Requestor Profile Version 1.0*. At `ftp://www6.software.ibm.com/software/developer/library/ws-fedpass.pdf`.

KALER, C., NADALIN, A., ET AL. 2004a. *Web Services Secure Conversation Language (WS-SecureConversation) Version 1.1*. At `http://msdn.microsoft.com/ws/2004/04/ws-secure-conversation/`.

KALER, C., NADALIN, A., ET AL. 2004b. *Web Services Trust Language (WS-Trust) Version 1.1*. At `http://msdn.microsoft.com/ws/2004/04/ws-trust/`.

KEMMERER, R., MEADOWS, C., AND MILLEN, J. 1994. Three systems for cryptographic protocol analysis. *Journal of Cryptology 7,* 2, 79–130.

KEYSER, C. 2004. Source code for SCT tokens in a farm source code. `http://blogs.msdn.com/chriskeyser/archive/2004/11/30/272678.aspx`.

KLEINER, E. AND ROSCOE, A. W. 2004. Web services security: A preliminary study using Casper and FDR. In *Automated Reasoning for Security Protocol Analysis (ARSPA'04)*.

KLEINER, E. AND ROSCOE, A. W. 2005. On the relationship between web services security and traditional protocols. In *Mathematical Foundations of Programming Semantics (MFPS XXI)*.

LOWE, G. 1996. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, vol. 1055. Springer, 147–166.

LOWE, G. 1997. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop, 1997*. IEEE Computer Society Press, 31–44.

Microsoft Corporation 2004. *Web Services Enhancements (WSE) 2.0 SP1*. Microsoft Corporation. At `http://msdn.microsoft.com/webservices/building/wse/default.aspx`.

MILNER, R. 1999. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

NADALIN, A., KALER, C., HALLAM-BAKER, P., AND MONZILLO, R. 2004. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004). OASIS Standard 200401*. At `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`.

NEEDHAM, R. AND SCHROEDER, M. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM 21,* 12, 993–999.

OASIS Security Services TC 2005. *Security Assertion Markup Language FAQ*. OASIS Security Services TC. At `http://www.oasis-open.org/committees/security/faq.php`.

PAULSON, L. 1998. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security 6*, 85–128.

PAULSON, L. C. 1999. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur. 2,* 3, 332–351.

RYAN, P. Y. A., SCHNEIDER, S. A., GOLDSMITH, M. H., LOWE, G., AND ROSCOE, A. W. 2001. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Pearson Education.

THAYER FÁBREGA, F., HERZOG, J., AND GUTTMAN, J. 1999. Strand spaces: Proving security protocols correct. *Journal of Computer Security 7*, 191–230.

VOGELS, W. 2003. Web services are not distributed objects. *IEEE Internet Computing 7,* 6, 59–66.

W3C 2003. *SOAP Version 1.2*. W3C. W3C Recommendation, at `http://www.w3.org/TR/soap12`.

WOO, T. AND LAM, S. 1993. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*. 178–194.

WS 2002. WS-Trust/WS-SecureConversation Interop Workshop. At `http://msdn.microsoft.com/webservices/community/workshops/trustworkshop112003.aspx`.

WS 2004. WS-SecureConversation/WS-Trust Interop Workshop. At `http://msdn.microsoft.com/webservices/community/workshops/TrustWorkshopOct2004.aspx`.

A. AUXILIARY DEFINITIONS AND PROOF OF THEOREM 4

Before detailing the proof of Theorem 4, we set up auxiliary notations for processes and present an intermediate script that models our protocol more abstractly.

A.1 Semantics for Processes and Events

We reason about TulaFale scripts as processes in the applied pi calculus [Abadi and Fournet 2001; Bhargavan et al. 2005]. We use the concrete syntax of TulaFale, detailed in Section 2.1, for all processes. We let P, Q, T, C, S, $\mathcal{S}$ range over processes.

We write $P \equiv Q$ when P and Q are structurally equivalent: they are equal by any capture-avoiding rearrangements of parallel compositions and restrictions. In particular, structural equivalence treats parallel composition as commutative and associative, with 0 as its neutral element. We write $P \rightarrow Q$ when P reduces to Q in a single reduction step: a communication on a channel, a predicate evaluation, or branching on a conditional test. Reductions are closed by structural equivalence. We write $P \rightarrow^* Q$ when P reduces to Q in $n \geq 0$ steps.

A context is a process with holes ($[\cdot]$) instead of some of its subprocesses. We consider several kinds of contexts: an *evaluation context* is a context with a single hole occurring at top level (that is, under restrictions and parallel compositions, but not under guards); for instance, an arbitrary active attacker is modeled as an evaluation context containing no events. A *guarded context* is a context where all occurrences of the holes are guarded. A *linear context* is either the hole, or an evaluation context applied to a (simpler) linear context under a linear guard: an input, an output, a filter, or an event. (In the following proofs, we use these contexts to decompose processes and express that their structure is invariant by reduction.) We always assume that restricted names are distinct from free names in their enclosing contexts—this can be systematically enforced using local renaming.

A process P *records the events* **begin** C, ..., **end** C′, ... when these events occur in evaluation contexts in P, that is, $P \equiv E[$ **begin** C $\mid \ldots \mid$**end** C′ $\mid \ldots]$ for some evaluation context E.

In order to illustrate these auxiliary definitions, we now give an equivalent, more explicit statement for Theorem 1 with a detailed agreement. We let $\mathcal{S}$ be the script presented in Section 4 and defined in `ssws-trust.tf`.

RESTATEMENT OF THEOREM 1. *Let E be an evaluation context with no events. If $E[\mathcal{S}] \rightarrow^* P$, then*

(*1*) *If P records event* **end** C1(partialSC,rstUid,clientEntropy)*, then P records either event* **begin** C1(partialSC,rstUid,clientEntropy) *or event* **begin** Leak(u)*, where* u *is the username identifier appearing in the* BaseToken *of* SC.

(*2*) *If P records* **end** C2(SC,rstUid,rstrUid)*, then P records either* **begin** C2(SC,rstUid, rstrUid) *or* **begin** Leak(sts)*, where* sts *is the URI identifier appearing in* STSInfo *in* SC.

Similarly, let $\mathcal{S}^{\mathcal{N}}$ be the script presented in Section 6.4 and defined in `ssws-secrm.tf`. We arrive at the following restatement of Theorem 4:

RESTATEMENT OF THEOREM 4. *Let E be an evaluation context with no events and where the name* secret *does not occur. If $E[\mathcal{S}^{\mathcal{N}}] \rightarrow^* P$, then*

(*1*) *If* P *records* **end** C3n(sc,sessionId,H,n)*, then* P *records either* **begin** C3n(sc,sessionId, H,n) *or* **begin** Leak(sc).

(*2*) *If* P *records* **end** C4n(sc,sessionId,H,n)*, then* P *records either* **begin** C4n(sc,sessionId, H,n) *or* **begin** Leak(sc).

(*3*) *If* P *outputs* secret *on a free channel, then* P *records* **begin** Leak(sc) *and either* **begin** C3n(sc,sessionId,H,n) *or* **begin** C4n(sc,sessionId,H,n) *where* secret *is the body of the last envelope of* H.

A.2 Proof of Theorem 4

The target properties of the theorem involve events that record the whole session history. Since the number of exchanges is unbounded, the detailed recorded history is unbounded as well, and the corresponding events cannot be automatically related via TulaFale and ProVerif.

Instead, we automatically establish a series of intermediate properties on similar, but more abstract scripts, and we compose these properties using standard, manual proof techniques for the pi calculus. In addition to $\mathcal{S}^{\mathcal{N}}$, used in the statement of Theorem 4, our proof relies on $\mathcal{S}^{\mathcal{A}}$, a more abstract script defined in `ssws-secrm-a.tf`. TulaFale terminates rapidly on $\mathcal{S}^{\mathcal{A}}$ (within about twenty minutes) but apparently diverges on $\mathcal{S}^{\mathcal{N}}$. To complete the proof of Theorem 4, we manually relate the behaviors of $\mathcal{S}^{\mathcal{N}}$ and $\mathcal{S}^{\mathcal{A}}$. Next, we outline the differences between $\mathcal{S}^{\mathcal{N}}$ (on the left) and $\mathcal{S}^{\mathcal{A}}$ (on the right). The client processes have the following structures:

```
private channel dc(item,item,item,item).
C[
  new sessionId:string;
  out dc(sc,sessionId,[],zero))
][
  !in dc(sc,sessionId,H,n);
  L3C[
    begin C3n (sc,sessionId,[Req @ H],n);
    L4C[
      end C4n (sc,sessionId,[Resp Req @ H],n);
      out dc(sc,sessionId,
             [Resp Req @ H],succ(n)). ]]]
```

```
private channel dc(item,item).
C[
  new sessionId:string;
  out dc(sc,sessionId)
][
  !in dc(sc,sessionId); in env(n);
  L3C[
    begin C3a (sc,sessionId,[Req],n);
    L4C[
      end C4a (sc,sessionId,[Resp Req],n);
      out dc(sc,sessionId).
                          ]]]
```

Channel dc—named sessionDbC in the scripts—appears in $\mathcal{S}^{\mathcal{N}}$ and $\mathcal{S}^{\mathcal{A}}$ only as shown above. The rest of the scripts is abstracted as a binary context $C[\cdot][\cdot]$ and two linear contexts, $L_{3C}[\cdot]$ and $L_{4C}[\cdot]$.

The client uses a message on private channel dc to record the state for every running session. To initiate a session, the client creates a fresh identifier sessionId and sends a message on dc representing the session state: its identifier and its security context sc (bound by the context $C[\cdot][\cdot]$). To extend an existing session, the client reads the session state on dc, performs the exchange, then replaces an updated state on dc. In $\mathcal{S}^{\mathcal{N}}$, the message also records the session history and the number for the next SOAP message, starting with an empty history and zero; the history is included in all events. In $\mathcal{S}^{\mathcal{A}}$, only sc and sessionId are recorded, the history is not used, and the next-message number is an arbitrary value provided by the environment on channel env; the simplified events are called C3a and C4a. (In the scripts, the message number n is named msgNumber.)

Similarly, the server processes in $\mathcal{S}^{\mathcal{N}}$ and $\mathcal{S}^{\mathcal{A}}$ have the following structures:

```
private channel ds(item,item,item,item).
S[
  in public(sessionId);
  L_ARC[out ds(sc,sessionId,[],zero)]
][
  !in ds(sc,sessionId,H,n);
  L_3S[
    end C3n (sc,sessionId,[Req @ H],n);
    L_4S[
      begin C4n (sc,sessionId,[Resp Req @ H],n);
      out ds(sc,sessionId,
             [Resp Req @ H],succ(n)). ]]]
```

```
private channel ds(item,item).
S[
  in public(sessionId);
  L_ARC[out ds(sc,sessionId)]
][
  !in ds(sc,sessionId); in env(n);
  L_3S[
    end C3a (sc,sessionId,[Req],n);
    L_4S[
      begin C4a (sc,sessionId,[Resp Req],n);
      out ds(sc,sessionId).
                                  ]]]
```

Channel ds—named sessionDbS in the scripts—appears in $\mathcal{S}^{\mathcal{N}}$ and $\mathcal{S}^{\mathcal{A}}$ only as explicited above, and the rest of the scripts is abstracted using a binary context $S[\cdot][\cdot]$ and three linear contexts, L_{ARC}, $L_{3S}[\cdot]$, and $L_{4S}[\cdot]$. The context L_{ARC} is part of the implementation of an anti-replay cache; before sending a message a message on ds, it checks that this is the first session with this identifier, and otherwise does nothing. (See process $S'_{y,sc}$ in Lemma 1 for our pi calculus implementation of the anti-replay cache.)

As a safe simplification, our servers input the session identifier sessionId for each new session directly from the environment, on channel public—named ContextProvideSessionId in the scripts—instead of reading this identifier from the first message of the session.

Reachable states of $\mathcal{S}^{\mathcal{N}}$. The following lemma gives an explicit representation for the reachable states of the protocol $\mathcal{S}^{\mathcal{N}}$. This representation is used later to relate the recorded events of $\mathcal{S}^{\mathcal{N}}$ to those of $\mathcal{S}^{\mathcal{A}}$. Crucially, the lemma concerns only the management of state on three channels; its correctness does not depend on the rest of the protocol.

LEMMA 1 (REACHABLE STATES FOR $\mathcal{S}^{\mathcal{N}}$). *We let*

$$
\begin{aligned}
C'_{sc} &= \textbf{new}\ \text{x:}\textbf{string};\ \textbf{out}\ \text{dc}(sc,\text{x},[],\text{zero}) \\
S'_{y,sc} &= \textbf{in}\ \text{cache}(\text{y}^\circ);(\textbf{out}\ \text{cache}([y@\text{y}^\circ])\ |\textbf{if}\ (y\ \textbf{in}\ \text{y}^\circ)\ \textbf{then}\ 0\ \textbf{else out}\ \text{ds}(sc,y,[],\text{zero})) \\
C_{succ} &= \textbf{!in}\ \text{dc}(\text{sc,x,H,n});LC[\textbf{out}\ \text{dc}(\text{sc,x,H'},\text{succ(n)})] \\
S_{succ} &= \textbf{!in}\ \text{ds}(\text{sc,y,H,n});LS[\textbf{out}\ \text{ds}(\text{sc,y,H'},\text{succ(n)})] \\
T^{\mathcal{N}} &= \textbf{private channel}\ \text{dc}(\textbf{item},\textbf{item},\textbf{item},\textbf{item}). \\
&\quad \textbf{private channel}\ \text{ds}(\textbf{item},\textbf{item},\textbf{item},\textbf{item}). \\
&\quad \textbf{private channel}\ \text{cache}(\textbf{items}). \\
&\quad E\big[C_{succ}\ |S_{succ}\ | \\
&\qquad C_{X'}[C'_{sc}]_{sc\in X'}\ |S_{Y'}[S'_{y,sc}]_{(y,sc)\in Y'}\ |\textbf{out}\ \text{cache}(Y^\circ)\ | \\
&\qquad \textstyle\prod_{(x,n)\in X} C_{x,n}[\ \textbf{out}\ \text{dc}(sc,x,H_{x,n},n)\ ]\ | \\
&\qquad \textstyle\prod_{(y,n)\in Y} S_{y,n}[\ \textbf{out}\ \text{ds}(sc,y,H_{y,n},n)]\big]
\end{aligned}
$$

where X' range over finite multisets of terms; X, Y, and Y' range over finite sets of pairs of terms and (term-coded) integers; Y° is a list collecting all first terms of pairs in Y; E ranges over evaluation contexts; $C_{X'}$ range over contexts with a hole for each element of X'; $S_{Y'}$ range over contexts with a hole for each element of Y'; $C_{x,n}$ and $S_{y,n}$ range over linear contexts indexed by X and Y; LC and LS are linear contexts appearing in $\mathcal{S}^{\mathcal{N}}$

such that $\mathcal{S}^{\mathcal{N}} \equiv T^{\mathcal{N}}_{\emptyset,\emptyset}$. *Let* E' *be an evaluation context, and assume that* dc*,* ds*, and* cache *do not occur in any of those contexts.*

If $E'[\mathcal{S}^{\mathcal{N}}] \rightarrow^* P$*, then*

(*1*) *there exists* $T^{\mathcal{N}}$ *such that* $P \equiv T^{\mathcal{N}}$*;*

(*2*) *reduction steps on* dc *communicate messages* $(sc, x, H_{x,i}, i)$ *for pairwise-distinct* (x, i) *such that* $(x, n) \in X$ *and* $i < n$*.*

(*3*) *reduction steps on* ds *communicate messages* $(sc, y, H_{y,i}, i)$ *for pairwise-distinct* (y, i) *such that* $(y, n) \in Y$ *and* $i < n$*.*

In this lemma, the process $T^{\mathcal{N}}$ represents a reachable state of $\mathcal{S}^{\mathcal{N}}$, parameterized by four index sets: the sets X and Y keep track of ongoing sessions for the client and the server, respectively, with initially $X = Y = \emptyset$; the sets X' and Y' keep track of sessions that have not started yet.

PROOF. The proof is by induction on the number of steps in $E'[\mathcal{S}^{\mathcal{N}}] \rightarrow^k P$. The base case holds by construction for $T^{\mathcal{N}}_{\emptyset,\emptyset}$. For the inductive case, the inductive hypothesis yields $T^{\mathcal{N}}$ such that $E'[\mathcal{S}^{\mathcal{N}}] \rightarrow^k \equiv T^{\mathcal{N}} \rightarrow P$. We perform a case analysis on the final reduction step using the structure of $T^{\mathcal{N}}$, with the following cases:

—The step is a communication on private channel dc. By construction, the input is the replicated input C_{succ}, and there are two subcases for the output:
  —The output is in C'_{sc} for some $sc \in X'$. Using structural equivalence, we rename the restricted name sessionId to some globally-fresh name x and lift the restriction on x to E. We remove sc from X', add (x, zero) to X, and define $H_{x,0}$ and $C_{x,0}[\_]$ to match the triggered process $LC[\dots]$.
  —The output is in $C_{x,n}$ for some $(x, n) \in X$. Thus, $C_{x,n}$ is an evaluation context. Using structural equivalence, we merge this context with E, we replace (x, n) by $(x, n + 1)$ in X, and define $H_{x,n+1}$ and $C_{x,n+1}$ to match the triggered process $LC[\dots]$.

—The step is a communication on private channel ds. By construction, the input is the replicated input S_{succ} and the output is in $S_{y,n}$ for some $(y, n) \in Y$. Thus, $S_{y,n}$ is an evaluation context. As in the case above, we merge $S_{y,n}$ with E, replace (y, n) by $(y, n + 1)$ in Y, and define $H_{y,n+1}$ and $S_{y,n+1}$ to match the triggered process $LS[\dots]$.

—The step is a communication on private channel cache. By construction, the only output on this channel is **out** cache(Y°); the input is $S'_{y,sc}$ for some $(y, sc) \in Y'$. We distinguish two subcases:
  —If $y \in Y^\circ$, then the triggered process is an output on cache of a list with the same elements as Y° in parallel with an inert process (since the inclusion test succeeds). To conclude, we let Y' become Y' minus (y, sc).
  —Otherwise, we let Y' become Y' minus (y, sc), let Y become Y plus (y, zero), let $H_{y,0}$ be [], and let $S_{y,0}$ be $[\cdot]$, so that the triggered process is an updated message on cache in parallel with the new element in the product on Y.

—All other communication steps preserve the structure of $T^{\mathcal{N}}$, for some updated sets X', Y' and contexts E, $C_{X'}$, $C_{x,n}$, $S_{Y'}$, $S_{y,n}$. □

Relating events of $\mathcal{S}^{\mathcal{N}}$ *and* $\mathcal{S}^{\mathcal{A}}$. In $\mathcal{S}^{\mathcal{N}}$, the correspondences C3n and C4n record similar session information, namely a tuple of the form (sc,sessionId, [M @ H],n). We define their

projection to events that may occur in $\mathcal{S}^{\mathcal{A}}$ as follows:

$$\begin{array}{rcl} \textbf{event}\ \text{C3n (sc,sessionId, [Req @ H],n)}^{\#} & = & \textbf{event}\ \text{C3a (sc,sessionId,[Req],n)} \\ \textbf{event}\ \text{C4n (sc,sessionId, [Resp Req @ H],n)}^{\#} & = & \textbf{event}\ \text{C4a (sc,sessionId,[Resp Req],n)} \end{array}$$

Other events are left unchanged: **event** Leak(sc)$^{\#}$ = **event** Leak(sc). We lift this projection to processes: $P^{\#}$ is obtained from P by projecting any event occurring in P. Our next lemma shows that, for any run of $\mathcal{S}^{\mathcal{N}}$, there is a corresponding run of $\mathcal{S}^{\mathcal{A}}$ that records the same Leak events, and an event **event** C$^{\#}$ for each event **event** C. (On the other hand, $\mathcal{S}^{\mathcal{A}}$ has runs that cannot be simulated by $\mathcal{S}^{\mathcal{N}}$.)

LEMMA 2 ($\mathcal{S}^{\mathcal{A}}$ SIMULATES $\mathcal{S}^{\mathcal{N}}$). *For any evaluation context $E^{\mathcal{N}}$ with no events and where the name* secret *does not occur, for any reductions $E^{\mathcal{N}}[\mathcal{S}^{\mathcal{N}}] \rightarrow^{*} P$, there exist an evaluation context $E^{\mathcal{A}}$ and reductions $E^{\mathcal{A}}[\mathcal{S}^{\mathcal{A}}] \rightarrow^{*} T^{\mathcal{A}}$ such that*

(*1*) *if P records e, then $T^{\mathcal{A}}$ records $e^{\#}$;*

(*2*) *if $T^{\mathcal{A}}$ records f, then P records some e such that $f = e^{\#}$;*

(*3*) *If P outputs* secret *on a free channel, then $T^{\mathcal{A}}$ also outputs* secret *on a free channel.*

PROOF. By Lemma 1(1), we have a process of the form $T^{\mathcal{N}}$ with $P \equiv T^{\mathcal{N}}$ (and in particular with the same events and outputs of secret on free channels). We obtain $T^{\mathcal{A}}$ from $T^{\mathcal{N}}$ by applying $.^{\#}$ to every event; erasing the message number and the history from every input and output on dc and ds; inserting an input **in** env(n) after every input on dc and ds; and adding outputs $\prod_{i=0}^{n}$**!out** env($succ^i$(zero)) at top-level, where $n = max\{i \mid (z,i) \in Y\}$. Thus, we define $T^{\mathcal{A}}$ and $E^{\mathcal{A}}[\_]$ as follows:

$$\begin{array}{rcl} C'^{\mathcal{A}}_{sc} & = & \textbf{new}\ \text{x:}\textbf{string};\textbf{out}\ \text{dc}(sc,\text{x}) \\ S'^{\mathcal{A}}_{y,sc}(Y) & = & \textbf{in}\ \text{cache}(\text{y}^{\circ});(\textbf{out}\ \text{cache}([y@\ \text{y}^{\circ}])\ |\textbf{if}\ (y\ \textbf{in}\ \text{y}^{\circ})\ \textbf{then}\ 0\ \textbf{else out}\ \text{ds}(sc,y))] \\ C^{\mathcal{A}}_{succ} & = & \textbf{!in}\ \text{dc(sc,x)};\textbf{in}\ \text{env(n)};LC^{\#}[\textbf{out}\ \text{dc(sc,x)}] \\ S^{\mathcal{A}}_{succ} & = & \textbf{!in}\ \text{ds(sc,y)};LS^{\#}[\textbf{out}\ \text{ds(sc,y)}] \\ T^{\mathcal{A}} & = & \prod_{i=0}^{n}\ \textbf{!out}\ \text{env}(succ^i(\text{zero}))\ \mid \\ & & \textbf{private channel}\ \text{dc}(\textbf{item},\textbf{item}). \\ & & \textbf{private channel}\ \text{ds}(\textbf{item},\textbf{item}). \\ & & \textbf{private channel}\ \text{cache}(\textbf{items}). \\ & & E^{\#}\big[C^{\mathcal{A}}_{succ}\ |S^{\mathcal{A}}_{succ}\ \mid \\ & & \quad C^{\#}_{X'}[\ C'^{\mathcal{A}}_{sc}\ ]_{sc\in X'}\ |S^{\#}_{Y'}[\ S'^{\mathcal{A}}_{y,sc}\ ]_{(y,sc)\in Y'}\ |\textbf{out}\ \text{cache}(Y^{\circ})\ \mid \\ & & \quad \textstyle\prod_{(x,n)\in X} C^{\#}_{x,n}[\ \textbf{out}\ \text{dc}(sc,x)\ ]\ \mid \\ & & \quad \textstyle\prod_{(y,n)\in Y} S^{\#}_{y,n}[\ \textbf{out}\ \text{ds}(sc,y)]\big] \end{array}$$

$$E^{\mathcal{A}}[\_] = \prod_{i=0}^{n}\textbf{!out}\ \text{env}(succ^i(\text{zero}))\ |\ E^{\mathcal{N}\#}[\_]$$

We show that, with these definitions, if $E^{\mathcal{N}}[\mathcal{S}^{\mathcal{N}}] \rightarrow^{*} P \equiv T^{\mathcal{N}}$, then $E^{\mathcal{A}}[\mathcal{S}^{\mathcal{A}}] \rightarrow^{*} T^{\mathcal{A}}$. The processes $T^{\mathcal{N}}$ and $T^{\mathcal{A}}$ differ only in the content of their events (but this content does not affect reductions) and on the messages communicated on dc and ds. Each communication step on dc is simulated by a matching communication step on dc immediately followed by a communication step on env to input a matching integer. Similarly, each communication step on ds is simulated by a matching communication step on ds and a

communication step on env. Any other reduction step in $T^{\mathcal{N}}$ immediately carries over to $T^{\mathcal{A}}$, with matching effects. □

The following lemmas state properties verified by TulaFale and ProVerif: some structural properties for $\mathcal{S}^{\mathcal{N}}$, and the main correspondence for $\mathcal{S}^{\mathcal{A}}$. They are established by running TulaFale on scripts `ssws-secrm.tf` and `ssws-secrm-a.tf`, respectively.

LEMMA 3. *For any evaluation context E with no events, if $E[\mathcal{S}^{\mathcal{N}}] \rightarrow^* P$, then:*

(*1*) *If P records* **end** C4n (sc,sessionId,[Resp Req @ H],n)*, then P also records* **begin** C3n (sc,sessionId,[Req @ H],n).

(*2*) *If P records* **begin** C4n (sc,sessionId,[Resp Req @ H],n)*, then P also records* **end** C3n (sc,sessionId,[Req @ H],n).

(*3*) *If P records* **begin** C3n(sc,sessionId,[Req @ H],succ(n))*, then P also records* **end** C4n (sc,sessionId,H,n).

(*4*) *If P records* **end** C3n(sc,sessionId,[Req @ H],succ(n))*, then P also records* **begin** C4n (sc,sessionId,H,n).

LEMMA 4 (AGREEMENT AND SECRECY IN $\mathcal{S}^{\mathcal{A}}$). *For any evaluation context E with no events and where the name* secret *does not occur, if $E[\mathcal{S}^{\mathcal{A}}] \rightarrow^* P$, then:*

(*1*) *If P records* **end** C3a(sc,sessionId,[Req],n)*, then P also records either* **begin** C3a(sc, sessionId,[Req],n) *or* Leak(sc).

(*2*) *If P records* **end** C4a (sc,sessionId,[Resp Req],n)*, then P also records either* **begin** C4a (sc,sessionId,[Resp Req],n) *or* Leak(sc).

(*3*) *If P outputs* secret *on a free channel, then P records* **begin** Leak(sc) *and either* **begin** C3a(sc,sessionId,[Req],n) *or* **begin** C4a(sc,sessionId,[Resp Req],n) *where* secret *is the body of* Req *or* Resp *respectively.*

We are now ready to prove the main result.

PROOF OF THEOREM 4. Property (3) of the theorem follows from Lemma 2(3) and Lemma 4(3). For Properties (1) and (2), we first show that x, n and y, n uniquely index each kind of events **begin** C3n, **end** C3n, **begin** C4n, and **end** C4n recorded in $\mathcal{S}^{\mathcal{N}}$, with n always representing an integer; this follows from the construction of X and Y and the linear communications on channels dc and ds, described in Lemma 1(2,3), that ensure that each event records sessionId and n at most once.

Let E be any evaluation context, with $E[\mathcal{S}^{\mathcal{N}}] \rightarrow^* P$. We define the family of properties $(\mathcal{P}_i)_{i>0}$ as follows:

—$\mathcal{P}_{2n+1}$: if P records **end** C3n(sc,sessionId,H,n), then P also records either **begin** C3n(sc, sessionId,H,n) or **begin** Leak(sc).

—$\mathcal{P}_{2n+2}$: if P records **end** C4n(sc,sessionId,H,n), then P also records either **begin** C4n(sc, sessionId,H,n) or **begin** Leak(sc).

We prove the whole family of properties by induction on i. For the base case $\mathcal{P}_1$, the process P records **end** C3n(sc,sessionId,[Req],0). Applying Lemmas 2(1), 4(1), and 2(2), we obtain that P also records either **begin** C3n(sc,sessionId,[Req],0) or Leak(sc), establishing the claim.

Suppose now that $\mathcal{P}_m$ holds for $m > 0$, and consider $\mathcal{P}_{m+1}$. We distinguish two cases, with a similar argument—the second case is illustrated in Figure 8. We use a structural

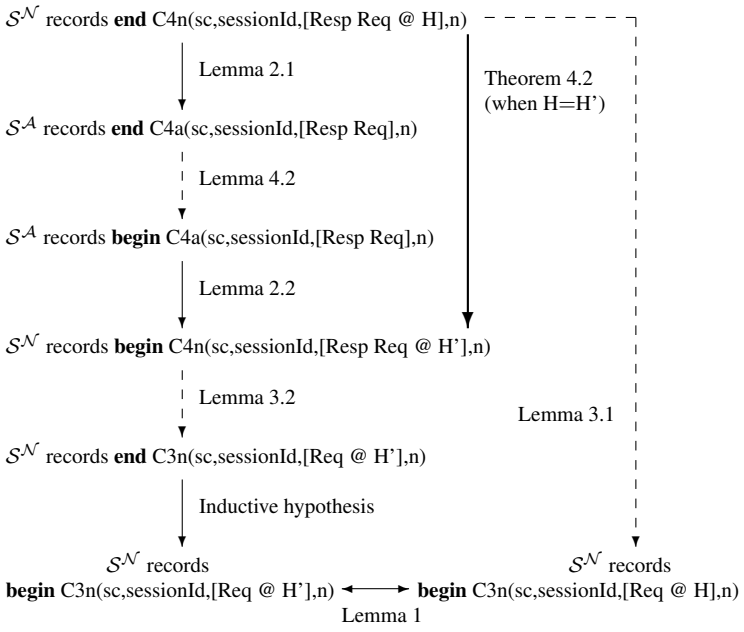

Fig. 8. Outline for the proof of Theorem 4(2) (bold arrow), using properties proved manually (solid arrows) and automatically (dashed arrows).

correspondence on $\mathcal{S}^{\mathcal{N}}$ towards the previous begin to match the old parts of the event, a correspondence on $\mathcal{S}^{\mathcal{A}}$ towards the latest begin to match the new parts, the inductive hypothesis to go from the previous end to the previous begin, and finally uniqueness of events at a given index.

—Case $m+1 = 2n+1$: the process P records **end** C3n(sc,sessionId,[Req @ H],succ(n)). By Lemma 2(1), Lemma 4(1), and Lemma 2(2), then either **begin** C3n(sc,sessionId, [Req @ H'],succ(n)) (for some H') or Leak(sc) is also recorded in P. If Leak(sc) is recorded then we are done, so consider the case in which **begin** C3n(sc,sessionId, [Req @ H'],succ(n)) is recorded. By Lemma 3(3), we know that **end** C4n(sessionId, H',n) is also recorded. By inductive hypothesis, we obtain **begin** C4n(sc,sessionId,H',n). Applying Lemma 3(4) to **end** C3n(sc,sessionId, [Req @ H],succ(n)), we obtain that **begin** C4n(sc,sessionId,H,n) is also recorded in P. Since events are indexed by sessionId and n, we obtain that H = H', establishing the claim.

—Case $m+1 = 2n+2$: the process P records **end** C4n(sc,sessionId,[Resp Req @ H],n). By Lemma 2(1), Lemma 4(2), and Lemma 2(2), either **begin** C4n(sc,sessionId, [Resp Req @ H'],n), for some H' or Leak(sc) is recorded in P. Consider the former case. By Lemma 3(2), **end** C3n(sc,sessionId,[Req @ H'],n) is also recorded. By inductive hypothesis, we obtain **begin** C3n(sc,sessionId,[Req @ H'],n). Applying Lemma 3(1) to **end** C4n(sc,sessionId,[Resp Req @ H],n), we obtain that **begin** C3n(sc,sessionId,[Req @ H],n) is also recorded in P. Since events are indexed by sessionId and n, we obtain that H = H', hence the property—this case is illustrated in Figure 8. □