Secure Sessions for Web Services

Karthikeyan Bhargavan
Microsoft Research

Ricardo Corin
University of Twente and
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

August 23, 2004 [*]

Abstract

WS-Security provides basic means to secure SOAP traffic, one envelope at a time. For typical web services, however, using WS-Security independently for each message is rather inefficient; besides, 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 introduces *security context*s, 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 limitations of these specifications.

[*]This is an extended version of a paper to appear at the ACM Workshop on Secure Web Services, October 29, 2004, Fairfax VA, USA.

Contents

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 [6]. We built our models by studying both the specifications and the WSE implementation [24]. Modelling reveals some potential vulnerabilities as well as allowing us to prove some formal properties.

Background: Web Services Security Web services are built on asynchronous communication of SOAP envelopes [29]. The mechanisms of WS-Security [26] provide means to secure these messages 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 [11, 27, 14, 17].) 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.

Background: WS-Trust and WS-SecureConversation Building on top of WS-Security, WS-Trust [21] describes how security tokens are requested and issued by SOAP processors; it relies on a dedicated *security token service* (STS) to evaluate requests and issue tokens. Moreover, WS-SecureConversation [20] 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 contents can be used to derive keys to protect traffic between these two parties.

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 [16]; we refer to this tutorial for additional implementation details.

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 secure contexts, SCs, with authenticated clients, to be used between clients and services. The first two steps are mechanisms covered by WS-Trust, while the latter exchanges (step 3)

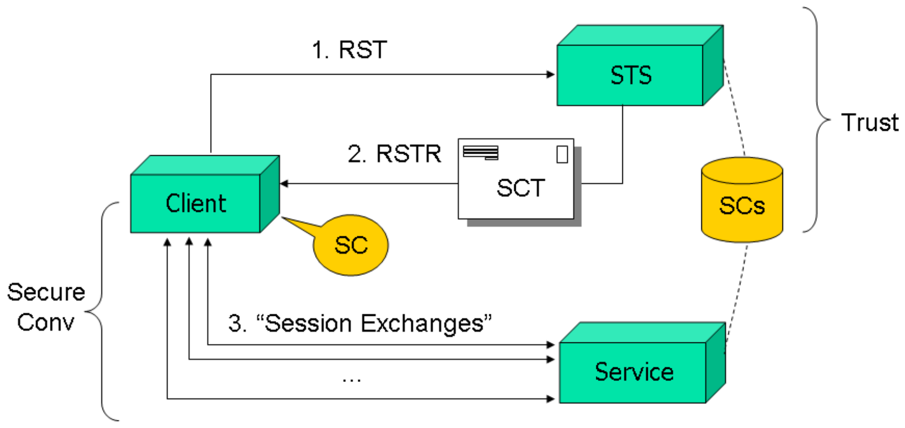

Figure 1: A typical protocol relying on WS-Trust and WS-SecureConversation

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 secure context SC, caches it and replies with a *Request Security Token Response* (RSTR) message, including a security context token (SCT) to indicate a new SC has been created (step 2). Crucially, 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 but partial threat model for web services, essentially an active attacker with some access to insider secrets. We develop simple, typical protocols relying on these specifications, and experiment with their implementation using WSE. We state and prove a series of core security properties for these protocols, thereby gaining confidence in our model of these specifications. We also informally discuss some limitations and potential vulnerabilities.

To the best of our knowledge, this is the first formal analysis of these two web services specifications, and these are the most complex SOAP-based security protocols yet formalized.

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 an 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. Finally, Section 7 discusses related work and Section 8 concludes.

This extended version of the paper includes the scripts used to obtain our results. Appendix A sketches the manual part of the proof of Theorem 4. Appendix B lists the scripts used for automated proofs. TulaFale itself is available from `http://Securing.WS`.

2 Modelling Systems and Threats

The security results in this paper are relative to the formal threat model of Dolev and Yao [12]. 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 [6], an XML version of the Dolev-Yao model embedded within the pi calculus. The pi calculus [25] 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 pure 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 to the protocol participants. 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 Blanchet's ProVerif analyzer [7, 8].

This section divides into two parts. The first part reviews the TulaFale language. The second part explains the particular threat model considered in this paper.

2.1 Systems Modelling in TulaFale

A TulaFale script defines an explicit system of multiple parallel processes, representing protocol participants. Processes interact by sending and receiving messages on a fixed set of channels. (In the pi calculus, processes may additionally generate channels at runtime.) Messages are terms in an algebraic model of XML, with signature and encryption primitives represented by idealized cryptographic functions [4]. The message formats of typical Dolev-Yao formalisms are rather abstract, and omit many details of the wire representation. In contrast, the TulaFale message format has the detailed structure of XML, and hence is sensitive to rewriting attacks that exploit this structure, such as for instance the compound structure of XML digital signatures. Moreover, we can directly transcribe the message formats of web services specifications into TulaFale, and check fidelity of the model with respect to messages generated by implementations.

We use logic programming to construct and check messages. For example, the following predicate asserts that the term tok is a username token [26, Section 6.2] for a principal with username u and password pwd, and that k is the symmetric key derived from this password using the nonce n and the timestamp t embedded in the token.

```
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)))).
```

All the predicates shown in the remainder of the paper are extracted from the code of our executable formal model [3]. 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 `</>`. Also, literal strings such as `"WS-Security"` are always quoted, whether they are within attributes or elements. Unquoted identifiers, such as u in `<Username>u</>`, are variables. Every variable has a sort: a **string** is an XML string, an **item** is an XML element or string, and a **bytes** is an array of bytes. Function symbols such as base64 and psha1 are abstract representations of operations on the data model; the function psha1 is an idealized hash function with no inverse. Given certain implementability constraints [4], predicates may be used in different modes, depending on whether each parameter is an input or an output. In our example, if tok and k are output parameters, and all the other parameters are inputs, TulaFale computes the two outputs, to yield a username token and its associated key.

We model web services and their clients as explicit processes that send and receive messages on a single soap channel, which models arbitrary transport layers for SOAP messaging. We annotate processes with events to indicate different phases of protocols. There are two kinds of events, **begin** and **end**, to mark initiation and apparently successful completion, respectively. Events carry data such as participant identities and the contents of messages exchanged. 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 [30] (also known as non-injective agreements [23]) between events. We formulate these as *robust safety* theorems: that in every run of the system, every occurrence of an **end** event has a corresponding **begin** event carrying the same data. For instance, authentication of an RST message is expressed as a correspondence between events marking the client sending and the server receiving the RST. Most security properties of TulaFale models stated in this paper are proved automatically by compiling to an intermediate pi calculus, and then running ProVerif.

2.2 Principals, Authentication Materials, Sessions, and Key Leakage

Our models assume the following participants and authentication materials:

- A single certification authority (CA), with public/private keypair kr/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 anyPrincipal) 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, of course, access to this database would be achieved by partial replication, but this is outside our model.

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, 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 [5].

3 Web Services Trust Language

WS-Trust "provides a framework for requesting and issuing security tokens, and to broker trust relationships" [21]. 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 their usage in context establishment protocols. 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 three SOAP actions and corresponding message elements for managing security tokens: for issuance, renewal, and validation. Moreover, these elements need not appear as envelope bodies; they may also be embedded 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 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. The nature of the agreement is left 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 and validation actions; error handling; unsolicited RSTRs; and advanced algorithm negotiation and delegation mechanisms. WS-Trust also proposes an optional attribute RequestSecurityToken/@context for correlation between RST and RSTRs. In our model, we rely instead on the message identifier of the enclosing envelope, which plays a similar role.

3.2 Syntax for RST/RSTR Exchanges

In the following, we focus on STSs implementing a simple, two-message RST/RSTR exchange for establishing a security context, as described in the specification [21, Section 6.1-2]. We begin by explaining the detail of the syntax of these messages and their intended semantics, which we accurately reflect in our models.

Principals: An RST may contain a BaseToken, typically an X.509 certificate or a username token, that identifies the requesting principal and that is used to authenticate the enclosing envelope. Alternatively, the RST may be anonymous. The RST may also contain an `<AppliesTo>` 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 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 of the key is guaranteed, irrespective of the other party's choice of entropy. (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: RSTs may include additional information, used for instance to demand some choice of cryptographic algorithm or policy, or to provide further authorization materials. We deal abstractly with such additional information, by recording it in the security context.

As a first concrete example of TulaFale code modelling WS-Trust, we give the predicates verifying 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,ref:string,etok,ExtraInfo:item):−
  rst = <RequestSecurityToken>
          <TokenType>"SecurityContextToken"</>
```

```
        <RequestType>"Issue"</>
        <AppliesTo><EndpointReference>srvURI</></>
        <Base><SecurityTokenReference>ref</></>
        <Entropy>etok </>
        ExtraInfo
      </>.
```

The STS decomposes each incoming RST with this predicate; it relies on pattern matching to decompose a (presumed) rst element passed as the first argument into a series of elements. The constant parts in the pattern ensure the RST is a 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; ref is a fragment URI to the security token identifying the requestor; finally, ExtraInfo collects elements not explicitly used in our model, but perhaps trusted by the protocol participants.

RSTRs returned by the STS include a *Requested Security Token*, indicating the identifier of the (newly created) SCT and a *Requested Proof Token*, containing (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).

Our model extends the specification to require 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 detailed in Section 4.5, without this extension, it is not possible to securely correlate the RSTR with its RST.

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 omits several important aspects 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 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 [14] or IKE [17] 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. For instance, 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 SC(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</></>.
```

Elements `<Base>`, `<STSInfo>`, and `<AppliesTo>` record information on the identity of the three principals involved: the client, the STS, and the service. We (arbitrarily) record the details of the security tokens identifying the client and the STS, and only record a 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 tampered RST.

Regarding privacy, although it is possible to use pseudonyms [19], 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 computation 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 can 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.

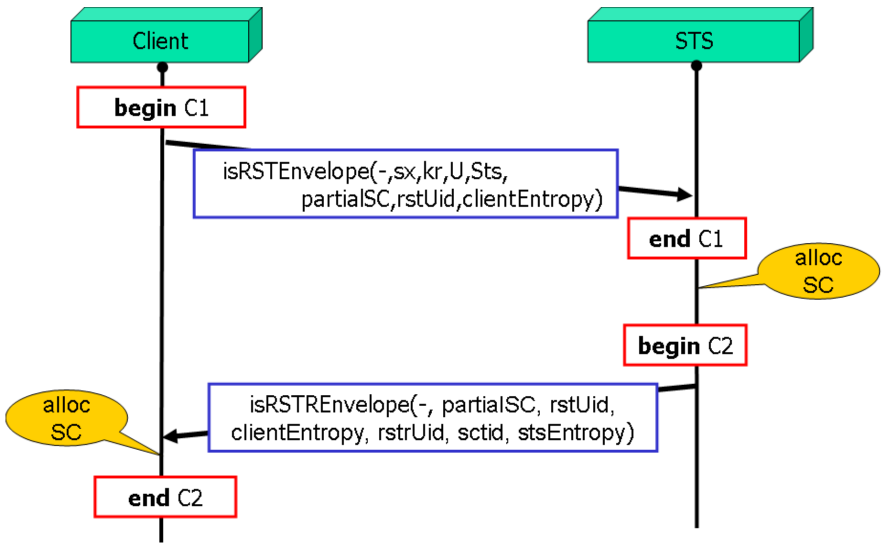
Client
STS
begin C1
isRSTEnvelope(-,sx,kr,U,Sts, partialSC,rstUid,clientEntropy)
end C1
alloc SC
begin C2
isRSTREnvelope(-, partialSC, rstUid, clientEntropy, rstrUid, sctid, stsEntropy)
alloc SC
end C2

Figure 2: Establishing a secure context in entropic mode

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 an agreement on a shared security context. We describe the exchange, detailing our implementation of principals and the processing of RST and RSTR envelopes, then state our main theorems for the resulting script, verified using the TulaFale and ProVerif tools.

4.1 Mapping Principals to TulaFale Processes

A normal RST/RSTR exchange consists of two messages exchanged between a client and an STS process, as depicted in Figure 2. 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. 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>` and `<Key>`, 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 needed to check the validity of public key certificates, and share the trusted database of principal secrets.

In our exchange, each envelope embeds a globally unique identifier; its structure is represented by the predicate uid, and it consists of a freshly generated message identifier, id, and a public timestamp, t.

The process Client in Figure 2 represents an instance of a SOAP client sending

RSTs and processing RSTRs on behalf of a user. 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 either mode, the server provides entropy. (We do not model a third mode, allowed by the specification, where the client alone provides entropy.) Figure 2 depicts a typical run in entropic mode. In both modes, the attacker initializes the client process by sending it a PartialSC that provides the parameters for a new security context, including the name of the user and the URIs for the STS and service. The client then retrieves the user record U from the trusted database (see Section 2) and constructs an RST message corresponding to PartialSC. A user record contains either a username and password or an X.509 certificate and its associated private signing key. The RST message has a unique identifier, rstUid, and in entropic mode, it also contains a fresh client-generated value, clientEntropy.

The process STS represents an STS server. It first retrieves a server principal record Sts, containing a URL address uriSTS, an X.509 public-key certificate certSTS, and the associated private signing key skSTS. When it receives an RST, it also retrieves the principal record U for the client. The trusted database thus represents all authorized clients of the STS. After checking the RST, the STS process generates a new security context with fresh identifier sctid and the received parameters PartialSC. It generates its own stsEntropy and uses it to compute the shared sckey associated with this context. It then returns an RSTR, uniquely identified by rstrUid, containing the sctid and stsEntropy to the client.

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 [24].

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 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 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:

$$\mathrm{C1} = (\mathrm{PartialSC},\mathrm{rstUid},\mathrm{clientEntropy})$$

Including the rstUid in C1 enables 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).

$$\mathrm{C2} = (\mathrm{SC},\mathrm{rstUid},\mathrm{rstrUid})$$

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 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 designating 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. This structure 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, the STS uses this predicate to check that a received envelope complies with this structure, as a first step of its processing. The full processing for the RST envelope 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,BaseTokenid,etok,ExtraInfo),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isEncryptedKey(etok,clientEntropy,sx,certSTS),
  isX509TokenPub(kr,BaseToken,u,BaseTokenid,ek,certU),
  isSignature(sig1,"rsasha1",ek,
        [<Body>rst</><To>uriSTS</><Action>"RSTSCT"</>
         <MessageId>id1</><Created>t1</>]),
  uid(rstUid,id1,t1),
  PartialSC(PartialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo).
```

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

The predicate first parses msgrst using envRST, extracting the rst, the relevant header fields, the message signature sig1, and the user's authenticating BaseToken. It then uses 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 then used to decrypt etok to retrieve the clientEntropy. Then, the predicate isX509TokenPub extracts the user's public key from BaseToken and the predicate isSignature checks that the corresponding private signing key has been used to generate the message signature sig1, and that sig1 covers the message body and all the parsed header elements. Finally, the predicates uid and PartialSC construct the outputs rstUid and PartialSC.

Here, we depict the 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 it defines a symmetric predicate for preparing RST envelopes on the client side.

4.3 Processing the RSTR Envelope

The SOAP envelope carrying RSTRs has a similar structure to the RST envelope, 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.

Both the creation of RSTRs at the service and the corresponding checks at the client are again 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) :–
  PartialSC(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),
  isEncryptedKeySym(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 the received RSTR envelope, msgrstr, along with kr and the PartialSC, rstUid, and clientEntropy received in the RST envelope. It then extracts as output the unique identifier rstrUid, and the new sctid and stsEntropy generated by the server.

This predicate uses many clauses similar to those of isRSTEnvelope. It first uses PartialSC and uid to parse PartialSC and rstUid into their components. and it uses isSTSPubInfo to extract the URI and certificate of the STS. Then, it parses the received envelope using envRSTR, checking that the `<RelatesTo>` header contains the message identifier of the RST, and extracting the rstr, STSToken and other headers. The predicate EntropicRSTR parses the rstr, checking that it was issued by the STS at uriSTS for the user principal identified in BaseToken and for the service at srvURI, and extracts the sctid and etok containing the encrypted stsEntropy. Then, isEncryptedDataSym decrypted etok using clientEntropy to extract stsEntropy. Next, the encryption key of the STS is extracted from the certSTS echoed in STSToken and isSignature checks that the message signature sig2 was generated with the corresponding private key, and that sig2 covers the rstr and the parsed header elements. Finally, uid puts together the output rstrUid from the message identifier and timestamp.

4.4 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. 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 the script in the presence of an active attacker (and hence all choices of operation modes):*

- *For each **end** C1 event with a safe client, there is a matching **begin** C1 event.*
- *For each **end** C2 event with a safe client and a safe STS, there is a matching **begin** C2 event.*

Hence, the exchange guarantees that any RST envelope from a safe client, accepted by a safe STS, and used to allocate a secure 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 the script in the presence of an active attacker (and hence all choices of operation mode), for each **begin** C2 with safe client and STS, the Key element recorded in SC remains secret.*

Hence, even if the service immediately uses the SC key to encrypt messages, only the client may decrypt those messages. Combining the two theorems, this also holds 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 known to the attacker instead of a fresh value as its entropy.

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

These results are automatically proved by running TulaFale on the script listed in Appendix B.2. 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.

4.5 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 the RSTR does not include the client identity (that is, consider omitting BaseToken from the predicate EntropicRSTR in Section 3.2) and suppose 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 an 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 [26] does not prescribe any general mechanism to correlate requests and responses securely. One solution—an IBM/Microsoft proposal to the WS-Security Technical Committee—involves echoing and signing (parts of) the signature of the request in the response message. Another solution, 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 [20]. We survey the specification, and in particular focus on the new security tokens it introduces.

5.1 Tokens for 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 embeds a *context identifier*, so that the recipient can access the relevant 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 embeds a fresh nonce and a reference to an SCT, and 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(nonce),key).
```

In general, the parent token need not be an SCT; instead one can use, for example, a Kerberos token, or 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 correct but ambiguous signatures, which may be misinterpreted by the recipient after an attacker has rewritten unsigned parts of the envelope.

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 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 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 now 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 3 (but disregard the dashed lines for now). 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).

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 [9] headers included in the envelope.

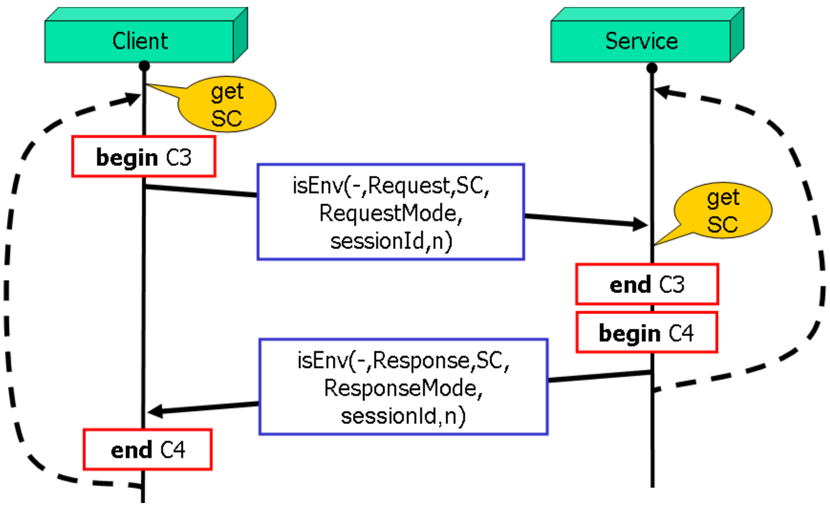
Client
Service
get SC
begin C3
isEnv(-,Request,SC, RequestMode, sessionId,n)
get SC
end C3
begin C4
isEnv(-,Response,SC, ResponseMode, sessionId,n)
end C4

Figure 3: Iterating exchanges

First, the client inputs from the attacker a Request envelope, which 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 matching sctid, if any, and completes Request by adding a fresh message identifier and a secret request 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 that record 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:

$$\mathrm{C3} = (\mathrm{SC},\mathrm{Request},\mathrm{RequestMode})$$

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 ResponseEnvelopes

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 env(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 is similar to those defined in Section 4.2, with three main differences: The envelope now includes a security context token (sct) and two derived key tokens (dksctEnc and dksctSig) used to indicate keys for encryption and signing. The envelope also includes a generic parameter DestInfo that provides headers specific to requests and responses. Finally, the envelope includes an encrypted body (ebody).

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

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

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

In the predicate, env 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.

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

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

Next, predicate SC 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:

Theorem 3 (Robust Safety of C3, C4 and Secrecy) *For all runs of the script in the presence of an active attacker (and hence all choices of operation modes):*

- *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 are kept secret.*

The proof is obtained by running TulaFale on the script listed in Appendix B.4.

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 3—thus modelling a more substantial conversation. For simplicity, we fix the operation mode and always use derived keys.

Each session is identified by sessionId, freshly generated by the client before sending its first request. Each request within the session is indexed by a sequence number. To this end, we (mostly) comply with the syntax of WS-ReliableMessaging [13] and use its simple request acknowledgment 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:string,msgNumber:bytes):−
  Sequence = <Sequence>
               <Identifier>sessionId</>
               <MessageNumber>base64(msgNumber)</></>.
```

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 n-th request and response, respectively, $\mathrm{C3}_n$ and $\mathrm{C4}_n$ record not only the envelope just sent (for **begin** events) or accepted (for **end** events), but also the preceding sequence of all previously-processed envelopes for the session, and their shared session identifier sessionId and security context SC (which provides in particular client and service identification).

$$\begin{aligned} \mathrm{C3}_n &= (\mathrm{SC},\mathrm{sessionId},\mathrm{Req}_0, \mathrm{Resp}_0, \ldots, \mathrm{Req}_n) \\ \mathrm{C4}_n &= (\mathrm{SC},\mathrm{sessionId},\mathrm{Req}_0, \mathrm{Resp}_0, \ldots, \mathrm{Req}_n, \mathrm{Resp}_n) \end{aligned}$$

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 $\mathrm{C3}_n$, $\mathrm{C4}_n$ and Secrecy) *For all runs of the script in the presence of an active attacker (and hence all choices of operation modes), and for all $n \geq 0$, we have:*

- *For each **end** $\mathrm{C3}_n$ with a safe security context, there is a matching **begin** $\mathrm{C3}_n$.*
- *For each **end** $\mathrm{C4}_n$ with a safe security context, there is a matching **begin** $\mathrm{C4}_n$.*
- *All request and response bodies protected by a safe security context remain secret.*

The proof is shown in Appendix A, while the script for our iterated protocol is listed in Appendix B.5.

The theorem is obtained by running ProVerif on a similar, but slightly more abstract script, in which sequencing is also controlled by the environment. We then manually carry over the properties from one script to another, relying on standard equivalences in the pi calculus [2].

6.5 Cautionary Notes

As in Section 4.5, 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 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 effect on the server, and thus replays are considered harmless in the first exchange. Besides, replay protection is hard to implement for web server farms, where several servers share the same SOAP address and STS.

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

7 Related Work

There has been great 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. We mention two roughly comparable examples.

Paulson [28] 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, Blanchet, and Fournet [1] show various security properties of the JFK key establishment protocol using ProVerif [7, 8]. ProVerif needs little user interaction compared to Isabelle, and also supports unbounded models of the protocol.

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* [10] show access control properties for SOAP-based web services, relying on an underlying secure channel such as SSL/TLS. Gordon and Pucella [15] 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 [22] extract abstract descriptions of some simple WS-Security protocols from XML message sequences.

These descriptions are intended for finite-state analysis with the FDR model checker, although in principle many tools for the Dolev-Yao model may be applicable.

Other formal work on web services protocols includes a model of Atomic Transaction [18].

8 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 hours 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 [31].

Acknowledgment

We gladly acknowledge the help and encouragement of the Microsoft WSE and Indigo teams.

References

[1] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. In *Proceedings of the 13th European Symposium on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 340–354. Springer, 2004.

[2] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

[3] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. Long version of this paper, including TulaFale scripts, at `http://research.microsoft.com/projects/samoa/secure-sessions-with-scripts.pdf`, Oct. 2004.

[4] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 198–209, 2004. An extended version appears as Microsoft Research Technical Report MSR–TR–2003–83.

[5] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, 2004. To appear.

[6] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*. Springer, 2004.

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

[8] B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, 2002.

[9] D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, Aug. 2004. At `http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/`.

[10] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing SOAP e-services. *International Journal of Information Security*, 1(2):100–115, 2002.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[12] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[13] C. Ferris, D. Langworthy, et al. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*, Mar. 2004. At `http://msdn.microsoft.com/ws/2004/03/ws-reliablemessaging/`.

[14] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol: Version 3.0. `http://home.netscape.com/eng/ssl3/draft302.txt`, November 1996.

[15] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *ACM Workshop on XML Security 2002*, pages 18–29, 2003. An extended version appears as Microsoft Research Technical Report MSR–TR–2002–108.

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

[17] D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE). `http://www.ietf.org/rfc/rfc2409.txt`, Nov. 1998.

[18] J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. Formal specification of a web services protocol. In *1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, 2004. University of Pisa.

[19] C. Kaler, A. Nadalin, et al. *Web Services Federation Language (WS-Federation) Version 1.0*, July 2003. At `http://msdn.microsoft.com/ws/2003/07/ws-federation/`.

[20] C. Kaler, A. Nadalin, et al. *Web Services Secure Conversation Language (WS-SecureConversation) Version 1.1*, May 2004. At `http://msdn.microsoft.com/ws/2004/04/ws-secure-conversation/`.

[21] C. Kaler, A. Nadalin, et al. *Web Services Trust Language (WS-Trust) Version 1.1*, May 2004. At `http://msdn.microsoft.com/ws/2004/04/ws-trust/`.

[22] E. Kleiner and A. W. Roscoe. Web services security: A preliminary study using Casper and FDR. In *Proceedings of Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.

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

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

[25] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[26] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, Mar. 2004. At `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`.

[27] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[28] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

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

[30] T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

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

A Proof Sketch for Theorem 4

In this section we refer to the script reported in Appendix B.5 as $\mathcal{S}^N$. Abstractly, the client process in $\mathcal{S}^N$ behaves as follows:

```
new sessionDb;
C[
  new sessionId:string;
  out sessionDb(emptyHistory,sessionId,zero()))
][
  !in sessionDb(History,sessionId,msgNumber);
  ...(linear)...;
  begin C3 (History.Mreq,msgNumber,sessionId);
  ...(linear)...;
  end C4 (History.Mreq.Mresp,msgNumber,sessionId);
  out sessionDb(newHistory,sessionId,succ(msgNumber)).
]
```

C[ ] denotes an (arbitrary) context in which the script is being run. Initially, the client creates a sessionDb channel, that is going to be used by the session to keep state after every exchange. Then the client creates a fresh sessionId, and finally enters a loop, where in each iteration a subsequent exchange is performed. In the script, emptyHistory is the empty tuple, while History contains the previous exchanges. Next we show the behaviour of the service, in which as a simplification we input the new sessionId directly from the attacker (since, in any case, the sessionId is sent in the clear in the first message):

```
new sessionDbS;
C[
  in attacker(sessionId);
  out sessionDbS(emptyHistory,sessionId,zero()))
][
  !in sessionDbS(History,sessionId,msgNumber);
  ...(linear)...;
  end C3 (History.Mreq,msgNumber,sessionId);
  ...(linear)...;
  begin C4 (History.Mreq.Mresp,msgNumber,sessionId);
  out sessionDbS(newHistory,sessionId,succ(msgNumber)).
]
```

where attacker is a public channel in control of the attacker.

We code a similar but more abstract script $\mathcal{S}^A$. The script is presented in Appendix B.6. TulaFale terminates rapidly on $\mathcal{S}^A$ (within about twenty minutes) but appears to diverge on $\mathcal{S}^N$. In $\mathcal{S}^A$, the client behaves as follows:

```
new sessionDb;
C[
  new sessionId:string;
  out sessionDb(sessionId)
][
  !in sessionDb(sessionId)
  in attacker(msgNumber);
  ...(linear)...;
  begin C3 (Mreq,msgNumber,sessionId);
  ...(linear)...;
  end C4 (Mresp,msgNumber,sessionId);
  out sessionDb(sessionId).
]
```

The service is coded similarly:

```
new sessionDbS;
C[
  in attacker(sessionId);
  out sessionDbS(sessionId))
][
  !in sessionDbS(sessionId);
  in attacker(msgNumber);
  ...(linear)...;
  end C3 (Mreq,msgNumber,sessionId);
  ...(linear)...;
  begin C4 (Mresp,msgNumber,sessionId);
  out sessionDbS(sessionId).
]
```

Intuitively, $\mathcal{S}^N$ and $\mathcal{S}^A$ differ on the fact that $\mathcal{S}^A$ inputs msgNumber directly from the attacker, while $\mathcal{S}^N$ uses built-in integers. Also, $\mathcal{S}^A$ does not record the whole conversation but only the last request, whereas $\mathcal{S}^N$ records all the previously exchanged requests and responses. Besides these two differences, $\mathcal{S}^N$ and $\mathcal{S}^A$ are identical.

In fact, in script $\mathcal{S}^N$, C3 and C4 are very similar: C4 only differs from C3 in that it adds the last response to the recorded conversation. To simplify notation, we can then fuse C3 and C4 into a unique correspondence C, indexed by n, and unifying requests and responses simply as messages M:

$$\mathrm{C}_n = (\mathrm{sessionId}, \mathrm{M}_0, ..., \mathrm{M}_n)$$

So, for example, the first event **begin** C3 (Mreq0,0,sessionId) corresponds to event **begin** C_0 =(sessionId,M_0), while the first **begin** C4 (Mreq0.Mresp0,0,sessionId) corresponds to **begin** C_1 =(sessionId,M_0,M_1).

Also notice that, for $n > 0$, we can decompose C_n as $\mathrm{C}_n = \mathrm{C}_{n-1}.\mathrm{M}_n$.

For our script $\mathcal{S}^A$, we have a weaker correspondence, which records only the last message:

$$\mathrm{C}_n^{\mathcal{A}} = (\mathrm{sessionId}, \mathrm{M}_n)$$

As a consequence of the fact that $\mathcal{S}^N$ and $\mathcal{S}^A$ are identical, we obtain that whenever an event (**begin** or **end**) C_n is recorded in an execution of $\mathcal{S}^N$, a corresponding event (**begin** or **end**) $C_n^{\mathcal{A}}$ is recorded in a corresponding execution of $\mathcal{S}^A$. This is established in the following Lemma:

Lemma 1 (Same events) *For the scripts $\mathcal{S}^N$ and $\mathcal{S}^A$ it holds:*

1. *For every reduction of $\mathcal{S}^N$ into T^N, there exists T^A and a corresponding reduction of $\mathcal{S}^A$ into T^A s.t. every recorded event* ***begin*** *or* ***end*** *C_n =(sessionId,$M_0, \ldots,$$M_n$) has a corresponding recorded event* ***begin*** *or* ***end*** *$C_n^{\mathcal{A}}$ =(sessionId,M_n).*

2. *For every reduction of $\mathcal{S}^A$ into T^A, there exists T^N and a corresponding reduction of $\mathcal{S}^N$ into T^N s.t. every recorded event* ***begin*** *or* ***end*** *$C_n^{\mathcal{A}}$ =(sessionId,M_n) has a corresponding recorded event* ***begin*** *or* ***end*** *C_n(sessionId,M'$_0, \ldots,$M'$_n$,M_n) for some M'$_i$, $i \in [0..n-1]$.*

This follows from inspection of the scripts $\mathcal{S}^N$ and $\mathcal{S}^A$, where every action and event action is identical, except the (above mentioned) difference, which does not influence the behaviour of the scripts ($\mathcal{S}^A$ has only an extra input from the environment).

By looking on the scripts, we also obtain the following useful lemma:

Lemma 2 (Script Structure) *In $\mathcal{S}^N$, for $n \geq 1$:*

1. *each time an* ***end*** *C_n (**end** $C_n^{\mathcal{A}}$) is recorded, then* ***begin*** *C_{n-1} (**begin** $C_{n-1}^{\mathcal{A}}$) is also recorded.*

2. *each time an* ***begin*** *C_n (**begin** $C_n^{\mathcal{A}}$) is recorded, then* ***end*** *C_{n-1} (**end** $C_{n-1}^{\mathcal{A}}$) is also recorded.*

The next lemma establishes the correspondence in $\mathcal{S}^{\mathcal{A}}$:

Lemma 3 (TF) *In $\mathcal{S}^A$,* ***end*** *$C_n^{\mathcal{A}}$ implies* ***begin*** *$C_n^{\mathcal{A}}$.*

This is established by running ProVerif on script $\mathcal{S}^A$.

Inspecting the script $\mathcal{S}^N$ we deduce that, for every $n \geq 1$, each **begin** and **end** event C is recorded at most once:

Lemma 4 (Linearity By Shape) *In $\mathcal{S}^N$, for $n \geq 1$, pairs (sessionId$, n)$ index all C events.*

For $n = 0$, we know that there is only one **begin** C_0 (since sessionId is always generated fresh), but there may be more than one **end** C_0 (since the first request may be replayed several times, see Section 6.4, paragraph before statement of Theorem 4). To establish Theorem 4, we need to require the uniqueness of **end** C_0 by coding an anti-replay cache on the service code in $\mathcal{S}^N$.

We are now ready to prove the following theorem, which establishes Theorem 4.

Theorem 5 (Robust Safety for C_n) *For all runs of the script $\mathcal{S}^N$ with the service having an anti-replay cache for the initial request, in the presence of an active attacker, and for all $n \geq 0$, we have that for each* ***end*** *C_n with a safe security context, there is a matching* ***begin*** *C_n. Furthermore, all request and response bodies protected by a safe security context remain secret.*

Proof. The last part of the theorem, regarding secrecy of the request and response bodies, can be established by running ProVerif over $\mathcal{S}^A$ (as listed in Appendix B.6), checking secrecy. Since both scripts are identical except an input of the msgNumber, we conclude that secrecy of the request and response bodies in $\mathcal{S}^A$ implies also secrecy of the request and response bodies in $\mathcal{S}^N$.

Now we consider the first part of the theorem, regarding correspondences. We proceed by induction on n. For $n = 0$, we have that $C_n = C_n^{\mathcal{A}}$, and then by Lemma 3 we obtain the result. Now take $n > 0$. Applying Lemma 2(1) to **end** C_n, we obtain that event **begin** C_{n-1} was recorded, with $C_n = C_{n-1}.M_n$.

Now consider again **end** C_n. Applying Lemma 1(1), we know that there is a corresponding event **end** $C_n^{\mathcal{A}}$ in $\mathcal{S}^A$. By Lemma 3, we obtain the recorded event **begin** $C_n^{\mathcal{A}}$ in $\mathcal{S}^A$. Applying Lemma 1 (2), we know that there must be a recorded event **begin** $C'_{n-1}M_n$ for some C'_{n-1}.

By Lemma 2(2), we obtain **end** C'_{n-1}. By inductive hypothesis, we obtain **begin** C'_{n-1}. Now, note that C'_{n-1} shares the same sessionId with C_{n-1}. If $n = 1$, then by hypothesis there is only one event **end** C_0 with the same sessionId, thus $C_0 = C'_0$. If $n > 1$, then by Lemma 4 we again obtain that $C_{n-1} = C'_{n-1}$. This implies that **begin** $C'_{n-1}M_n$ =**begin** $C_{n-1}M_n$ = **begin** C_n, establishing the theorem.

B Scripts

B.1 Common Predicates and Processes

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// common.tf — Model of Trust/SecureConversation

predicate mkRefs(ts:items,rs:items) :−
  ts = [t1 t2 t3],
  mkRef(t1,r1),
  mkRef(t2,r2),
  mkRef(t3,r3),
  rs = [r1 r2 r3].


predicate isRefs(ts:items,rs:items) :−
  ts = [t1 t2 t3],
  rs = [r1 r2 r3],
  ref(t1,r1),
  ref(t2,r2),
  ref(t3,r3).

// ============================
// Data structure to represent a service identity

predicate isService(S:item,uri:item,ac:item) :−
  S = <Service><To>uri</> <Action>ac</></>.

// same as in library but with Identifier

predicate isUsernameToken (tok:item,U:item,id:string,n:bytes,t:string,k:bytes,u:string) :−
  isUser(U,u,pwd),
  tok = <UsernameToken wsuId=id>
          <Username> u </>
          <Nonce> base64(n) </>
          <Created> t </> </>,
  k = psha1(pwd,concat(n,utf8(t))).
```

```
// symmetric encryption
predicate isEncryptedDataSym (encrypted:item,plain:item,k:bytes) :−
   encrypted = <EncryptedData><CipherData><CipherValue>base64(cipher)</></></>,
   c14n(plain) = decaes(k,cipher).

predicate mkEncryptedDataSym (encrypted:item,plain:item,k:bytes) :−
   cipher = aes(k,c14n(plain)),
   encrypted = <EncryptedData><CipherData><CipherValue>base64(cipher)</></></>.

// symmetric key encryption
predicate isEncryptedKeySym (encrypted:item,plain:bytes,k:bytes) :−
   encrypted = <EncryptedKey><CipherData><CipherValue>base64(cipher)</></></>,
   plain = decaes(k,cipher).

predicate mkEncryptedKeySym (encrypted:item,plain:bytes,k:bytes) :−
   cipher = aes(k,plain),
   encrypted = <EncryptedKey><CipherData><CipherValue>base64(cipher)</></></>.


predicate isEncryptedKey (encrypted:item,plain:bytes,dk:bytes,cert:bytes) :−
   encrypted = <EncryptedKey>
                    <CipherData><CipherValue>base64(cipher)</></>
                    <Key><BinarySecurityToken>base64(cert)</></></>,
   plain = decrsa(dk,cipher).

predicate mkEncryptedKey (encrypted:item,plain:bytes,ek:bytes,cert:bytes) :−
   cipher = rsa(ek,plain),
   encrypted = <EncryptedKey>
                    <CipherData><CipherValue>base64(cipher)</></>
                    <Key><BinarySecurityToken>base64(cert)</></></>.

// ====================================
// <SecurityContextToken> − reference to a SecurityContext
// <SecurityContext> − actual shared context −−
// WS−specific, reflecting class


predicate PartialSC(sd:item,mode:string,UserToken:item,StsInfo:item,appTo:string,extra:item) :−
   sd = <SecurityContext>
            <Base>UserToken</>
            <STSInfo>StsInfo</>
            <AppliesTo>appTo</>
            <EntropyMode>mode</>
            <ExtraInfo>extra</>
          </>.

predicate remCertPartialSC(partialSC:item,remCertSC:item) :−
   PartialSC(partialSC,mode,UserToken,StsInfo,appTo,extra),
   isSTSPubInfo(StsInfo,uriSTS,certSTS),
   remCertSC = <SecurityContext>
            <Base>UserToken</>
            <STSUri>uriSTS</> // in WSE impl, URI only, here includes a binarysecurity token containing STS cert
            <AppliesTo>appTo</>
            <EntropyMode>mode</>
            <ExtraInfo>extra</>
          </>.

predicate SC(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</></>.
```

```
predicate fillSC(sc:item,partialsc:item,sctid:string,sctkey:bytes) :−
  PartialSC(partialsc,mode,UserToken,StsInfo,appTo,extra),
  SC(sc,sctid,sctkey,mode,UserToken,StsInfo,appTo,extra).

predicate X509Token (tok:item,cert:bytes) :−
  tok = <BinarySecurityToken ValueType="X509v3"> base64(cert) </>.

predicate isX509TokenId (kr:bytes,tok:item,U:item,u:string,id:string,sk:bytes,certU:bytes) :−
  isX509User(U,u,sk,certU,kr),
  tok = <X509Token wsuId=id>
          <BinarySecurityToken>
            base64(certU)
          </>
        </>.

predicate isPrincipal(tok:item, u:string) :−
  tok = <X509Token wsuId=id>
          <BinarySecurityToken>
            base64(certU)
          </>
        </>,
  u = x509user(certU).

predicate isPrincipal(tok:item, u:string) :−
  tok = <UsernameToken wsuId=id>
          <Username> u </>
          <Nonce> base64(n) </>
          <Created> t </> </>.

predicate isX509TokenPub (kr:bytes,tok:item,u:string,id:string,ek:bytes,certU:bytes) :−
  tok = <X509Token wsuId=id>
          <BinarySecurityToken>
            base64(certU)
          </>
        </>,
  isX509Cert(certU,kr,u,"rsasha1",ek).


predicate mkX509User (U:item,u:string,sk:bytes,xcert:bytes,sr:bytes) :−
  xcert = x509(sr,u,"rsasha1",pk(sk)),
  U = <User><Username>u</><Cert>base64(xcert)</><Sk>base64(sk)</></>.


predicate isX509User (U:item,u:string,sk:bytes,xcert:bytes,kr:bytes) :−
  U = <User><Username>u</><Cert>base64(xcert)</><Sk>base64(sk)</></>,
  isX509Cert(xcert,kr,u,"rsasha1",pk(sk)).


// ==================================
// Generate fresh username/password combination
channel genUser(string).
channel leakUser(string).
private channel anyUser(item).
event LeakUser(string).

process MkUser() =
  in genUser(u);
  new pwd:string;
  filter isUser(U,u,pwd) → U;
  !out anyUser (U).

process MkUserLeak(kr:bytes) =
  in leakUser(u);
  in anyUser(U);
  event LeakUser(u);
  ((filter isUser(U,u,pwd) → pwd;
    out publish(pwd)) |
   (filter isX509User(U,u,sk,xcert,kr) ⟶ sk,xcert;
    out publish(base64(sk)))).

channel genUserCert(string).
process MkUserCert(sr:bytes) =
  in genUserCert(u);
  new sk:bytes;
  filter mkX509User(U,u,sk,xcert,sr) ⟶ U,xcert;
  out publish (base64(xcert));
  !out anyUser (U).

// ================================
// Relates STS with a specific cert; the attacker can choose the details
// of a STS, but not their certs
// ==============================
// Data structure to represent a token server
predicate isSTSPubInfo(S:item,uri:string,cert:bytes) :−
  S = <STS><To>uri</><BinarySecurityToken>base64(cert)</></>.

predicate isSTSInfo(S:item,uri:string,subj:string,ek:bytes,cert:bytes) :−
  S = <STS><To>uri</><BinarySecurityToken>base64(cert)</></>,
  subj = x509user(cert),
  ek = x509key(cert).

predicate isSTSPrivInfo(S:item,StsInfo:item,sx:bytes) :−
  S = <STSPriv>
        StsInfo
        <PrivateSigningKey>base64(sx)</>
        </>,
  isSTSPubInfo(StsInfo,uri,cert),
  pk(sx) = x509key(cert).

predicate isSTS(S:item,StsInfo:item,uri:string,subj:string,sk:bytes,cert:bytes) :−
  isSTSPrivInfo(S,StsInfo,sk),
  isSTSInfo(StsInfo,uri,subj,pk(sk),cert).

predicate uid(uid:item, id:string, t:string) :−
  uid = <Uid>id t</>.

// ==================================
// Generate fresh principals, with username/password or X.509 certs
channel genPrincipal(string).
channel leakPrincipal(string).
private channel anyPrincipal(item).
event LeakPrincipal(string).

process MkPrincipal() =
  in genPrincipal(u);
  new pwd:string;
  filter isUser(U,u,pwd) ⟶ U;
  !out anyPrincipal (U).

channel genPrincipalCert(string).
process MkPrincipalCert(sr:bytes) =
  in genPrincipalCert(u);
  new sk:bytes;
  filter mkX509User(U,u,sk,xcert,sr) ⟶ U,xcert;
  out publish (base64(xcert));
  !out anyPrincipal (U).

process MkPrincipalLeak(kr:bytes) =
  in leakPrincipal(u);
  in anyPrincipal(U);
  event LeakPrincipal(u);
  ((filter isUser(U,u,pwd) ⟶ pwd;
    out publish(pwd)) |
   (filter isX509User(U,u,sk,xcert,kr) ⟶ sk,xcert;
    out publish(base64(sk)))).
```

```
channel genSTS(item).
private channel anySTS(item).
process MkSTS(kr:bytes) =
  in genSTS(S);
  filter isSTSPubInfo(S,uri,cert)  → uri,cert;
  in anyPrincipal(U);
  filter isX509User(U,u,sx,cert,kr) → u,sx;
  filter isSTSPrivInfo(StsPriv,S,sx)  → StsPriv;
  !out anySTS(StsPriv).

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

B.2 Authentication and Secrecy for WS-Trust (Theorems 1 and 2)

B.2.1 WS-Trust: Main Processes

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wstrust.tf − Model of Trust/SecureConversation

// Part 1: SC establishment, RST/RSTR exchange
// see WS−Trust spec 1.1 May 2004

// This file contains predicates to create envelopes for messages RST/RSTR.

constructor undef():bytes.

// ==============================
// RST template
// Client provides entropy Case
// with ExtraInfo for arbitrary extensions (Lifetime...) added by Ricardo

predicate EntropicRST(rst:item,srvURI,ref:string,etok,ExtraInfo:item):−
  rst = <RequestSecurityToken>
          <TokenType>"SecurityContextToken"</>
          <RequestType>"Issue"</>
          <AppliesTo><EndpointReference>srvURI</></>
          <Base><SecurityTokenReference>ref</></>
          <Entropy>etok </>
          ExtraInfo
        </>.

// Client does NOT provide entropy Case
predicate NonEntropicRST(rst:item,srvURI:string,ref:string,ExtraInfo:item):−
  rst = <RequestSecurityToken>
          <TokenType>"SecurityContextToken"</>
          <RequestType>"Issue"</>
          <AppliesTo><EndpointReference>srvURI</></>
          <Base><SecurityTokenReference>ref</></>
          ExtraInfo
        </>.

// ================================
// Envelope template for RST SCT
// (omitted ReplyTo)

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</></>.

// ==================================
// Make Message RST

// Four cases arising from entropyMode={Both,Server}, and algorithm={hmacsha1,rsasha1} (client using Username token or X509)
predicate mkEntropicRSTEnvelope(msgrst:item,kr:bytes,U:item,partialSC:item,
                                rstUid:item,clientEntropy:bytes) :-
  PartialSC(partialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,id1,t1),
  isSTSPubInfo(StsInfo,uriSTS,certSTS),
  isX509Cert(certSTS,kr,subjSTS,"rsasha1",ek),
  mkEncryptedKey(etok,clientEntropy,ek,certSTS),
  isUsernameToken(BaseToken,U,BaseTokenid,utokN,utokTS,sk,u),
  EntropicRST(rst,srvURI,BaseTokenid,etok,ExtraInfo),
  mkSignature(sig1,"hmacsha1",sk,
           [<Body>rst</>
            <To>uriSTS</>
            <Action>"RSTSCT"</>
            <MessageId>id1</>
            <Created>t1</>]),
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken).
predicate mkEntropicRSTEnvelope(msgrst:item,kr:bytes,U:item,
                                partialSC:item,rstUid:item,clientEntropy:bytes) :-
  PartialSC(partialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,id1,t1),
  isSTSPubInfo(StsInfo,uriSTS,certSTS),
  isX509Cert(certSTS,kr,subjSTS,"rsasha1",ek),
  mkEncryptedKey(etok,clientEntropy,ek,certSTS),
  isX509TokenId(kr,BaseToken,U,u,BaseTokenid,sk,certU),
  EntropicRST(rst,srvURI,BaseTokenid,etok,ExtraInfo),
  mkSignature(sig1,"rsasha1",sk,
            [<Body>rst</>
            <To>uriSTS</>
            <Action>"RSTSCT"</>
            <MessageId>id1</>
            <Created>t1</>]),
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken).


predicate mkNonEntropicRSTEnvelope(msgrst:item,kr:bytes,U:item,partialSC:item,rstUid:item) :-
  PartialSC(partialSC,"Server",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,id1,t1),
  isSTSPubInfo(StsInfo,uriSTS,certSTS),
  isX509Cert(certSTS,kr,subjSTS,"rsasha1",ek),
  isX509TokenId(kr,BaseToken,U,u,BaseTokenid,sk,certU),
  NonEntropicRST(rst,srvURI,BaseTokenid,ExtraInfo),
  mkSignature(sig1,"rsasha1",sk,
    [ <Body>rst</>
      <To>uriSTS</>
      <Action>"RSTSCT"</>
      <MessageId>id1</>
      <Created>t1</>]),
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken).

// ================================
// Check Message RST

predicate isRSTEnvelope(msgrst:item,kr:bytes,U:item,Sts:item,
                        PartialSC:item,rstUid:item,clientEntropy:bytes) :-
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken),
  EntropicRST(rst,srvURI,BaseTokenid,etok,ExtraInfo),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isEncryptedKey(etok,clientEntropy,sx,certSTS),
  isUsernameToken(BaseToken,U,BaseTokenid,n,t,sk,u),
  isSignature(sig1,"hmacsha1",sk,[<Body>rst</><To>uriSTS</><Action>"RSTSCT"</>
                                  <MessageId>id1</><Created>t1</>]),
  uid(rstUid,id1,t1),
  PartialSC(PartialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo).

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,BaseTokenid,etok,ExtraInfo),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isEncryptedKey(etok,clientEntropy,sx,certSTS),
  isX509TokenPub(kr,BaseToken,u,BaseTokenid,ek,certU),
  isSignature(sig1,"rsasha1",ek,
         [<Body>rst</><To>uriSTS</><Action>"RSTSCT"</>
          <MessageId>id1</><Created>t1</>]),
  uid(rstUid,id1,t1),
  PartialSC(PartialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo).

predicate isRSTEnvelope(msgrst:item,kr:bytes,U:item,Sts:item,
                                    partialSC:item,rstUid:item,clientEntropy:bytes) :−
  envRST(msgrst,rst,uriSTS,id1,t1,sig1,BaseToken),
  uid(rstUid,id1,t1),
  NonEntropicRST(rst,srvURI,BaseTokenid,ExtraInfo),
  clientEntropy=utf8("zero"),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isX509TokenPub(kr,BaseToken,u,BaseTokenid,ek,certU),
  isSignature(sig1,"rsasha1",ek,
                [<Body>rst</>
                <To>uriSTS</>
                <Action>"RSTSCT"</>
                <MessageId>id1</>
                <Created>t1</>]),
  PartialSC(partialSC,"Server",BaseToken,StsInfo,srvURI,ExtraInfo).

// ===============================
// RSTR request response template

// Entropy case

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 </>
       </>.

// No Entropy case
predicate NonEntropicRSTR(rstr:item,srvURI:string,BaseToken:item,uriSTS:string,sctid:string,etok:item):−
  rstr = <RequestSecurityTokenResponse>
            <AppliesTo><EndpointReference>srvURI</></>
            <RequestedSecurityToken>
              <SecurityContextToken>
                <Identifier>sctid</></></>
            <RequestedProofToken>
              etok </>
            <RequestInfo>
              BaseToken
              uriSTS </>
          </>.

// ===============================
// Envelope template for RSTR SCT
//
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</></>.

// ==============================
// Make Message RSTR

// Entropy case
// Ricardo: <AddedInfo> added to get agreement. NOT present in the wse−samples
predicate mkEntropicRSTREnvelope(msgrstr:item,kr:bytes,Sts:item,
                                 partialSC:item,rstUid:item,clientEntropy:bytes,
                                 rstrUid:item,sctid:string,stsEntropy:bytes) :−
  PartialSC(partialSC,"Both",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,rto,t1),
  uid(rstrUid,id2,t2),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  mkEncryptedKeySym(etok,stsEntropy,clientEntropy),
  EntropicRSTR(rstr,srvURI,BaseToken,uriSTS,sctid,etok),
  X509Token(STSToken,certSTS),
  mkSignature(sig2,"rsasha1",sx,
              [<Body>rstr</>
              <RelatesTo>rto</>
              <Action>"RSTRSCT"</>
              <MessageId>id2</>
              <Created>t2</>]),
  envRSTR(msgrstr,rstr,id2,t2,STSToken,sig2,rto).



// No entropy case
// Ricardo's choice: entropyMode="no" only allowed when requestor is using x509 cert
predicate mkNonEntropicRSTREnvelope(msgrstr:item,kr:bytes,Sts:item,partialSC:item,
                                    rstUid:item,rstrUid:item,sctid:string,stsEntropy:bytes) :−
  PartialSC(partialSC,"Server",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,rto,t1),
  uid(rstrUid,id2,t2),
  isSTS(Sts,StsInfo,uriSTS,subjSTS,sx,certSTS),
  isX509TokenPub(kr,BaseToken,u,Basetokenid,ku,certU),
  mkEncryptedKey(etok,stsEntropy,ku,certU),
  NonEntropicRSTR(rstr,srvURI,BaseToken,uriSTS,sctid,etok),
  X509Token(STSToken,certSTS),
  mkSignature(sig2,"rsasha1",sx,
              [<Body>rstr</>
              <RelatesTo>rto</>
              <Action>"RSTRSCT"</>
              <MessageId>id2</>
              <Created>t2</>]),
  envRSTR(msgrstr,rstr,id2,t2,STSToken,sig2,rto).

// ==============================
// Check Message RSTR

// Entropy case

predicate isEntropicRSTREnvelope(msgrstr:item,kr:bytes,partialSC:item,
                                 rstUid:item,clientEntropy:bytes,
                                 rstrUid:item,sctid:string,stsEntropy:bytes) :−
  PartialSC(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),
  isEncryptedKeySym(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).

// No Entropy case
predicate isNonEntropicRSTREnvelope(msgrstr:item,kr:bytes,U:item,partialSC:item,
                                                    rstUid:item,rstrUid:item,sctid:string,stsEntropy:bytes) :−
  PartialSC(partialSC,"Server",BaseToken,StsInfo,srvURI,ExtraInfo),
  uid(rstUid,rto,t1),
  envRSTR(msgrstr,rstr,id2,t2,STSToken,sig2,rto),
  X509Token(STSToken,certSTS),
  uid(rstrUid,id2,t2),
  NonEntropicRSTR(rstr,srvURI,BaseToken,uriSTS,sctid,etok),
  isX509User(U,u,sk,certU,kr),
  isEncryptedKey(etok,stsEntropy,sk,certU),
  isSTSPubInfo(StsInfo,uriSTS,certSTS),
  isX509Cert(certSTS,kr,subjSTS,"rsasha1",ek),
  isSignature(sig2,"rsasha1",ek,
                [<Body>rstr</>
                <RelatesTo>rto</>
                <Action>"RSTRSCT"</>
                <MessageId>id2</>
                <Created>t2</>]).

// ============================
// Act as a client, on behalf of U

//channel init(string,item,item,string,item,item).
channel init(item,string).
private channel anySCClient(item).
event C1(string,item,item,bytes).
event C2(string,string,item,item,item).

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

// depending on attacker's choice, branch
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 remCertPartialSC(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))).

event LeakSC(item).
channel leakSC(string).

process ClientSCLeak() =
   in leakSC(sctid);
   in anySCClient(sc);
   filter SC(sc,sctid,sctkey,x1,x2,x3,x4,x5) → sctkey,x1,x2,x3,x4,x5;
   event LeakSC(sc);
   out publish(base64(sctkey)).

// ================================
// STS Token Server
   private channel anySCService(item).
   //channel accept(string,string).
   channel accept(string).
process STS(kr:bytes) =
   in soap(msgrst);
   in anyPrincipal(U);
   in anySTS(Sts);
   filter isRSTEnvelope(msgrst,kr,U,Sts,partialSC,rstUid,clientEntropy) → partialSC,rstUid,clientEntropy;
   filter PartialSC(partialSC,entropyMode,UserToken,StsInfo,x1,x2)  → entropyMode,UserToken,StsInfo,x1,x2;
   filter isPrincipal(UserToken,u)  → u;

// in accept (id2,t2);
   in accept (t2);
   new sctid:string;
   new stsEntropy:bytes;
   new id2:string;
   filter uid(rstrUid,id2,t2)  → rstrUid;
   filter isSTSInfo(StsInfo,uriSTS,subjSTS,ekSTS,certSTS)  → uriSTS,subjSTS,ekSTS,certSTS;

   if entropyMode="Both" then
   (
     end C1 (u,partialSC,rstUid,clientEntropy);

     filter mkEntropicRSTREnvelope(msgrstr,kr,Sts,partialSC,rstUid,clientEntropy,rstrUid,sctid,stsEntropy)  → msgrstr;
     filter fillSC(fullSC,partialSC,sctid,psha1(base64(clientEntropy),stsEntropy)) → fullSC;

     begin C2 (u,subjSTS,fullSC,rstUid,rstrUid);
     out soap(msgrstr);
     !out anySCService(fullSC))

   else
   (
     if entropyMode="Server" then
     (

     filter remCertPartialSC(partialSC,remCertSC)  → remCertSC;
     end C1 (u,remCertSC,rstUid,utf8("zero"));

     filter mkNonEntropicRSTREnvelope(msgrstr,kr,Sts,partialSC,rstUid,rstrUid,sctid,stsEntropy)  → msgrstr;
     filter fillSC(fullSC,partialSC,sctid,stsEntropy) → fullSC;

     begin C2 (u,subjSTS,fullSC,rstUid,rstrUid);
     out soap(msgrstr);

     !out anySCService(fullSC))).
```

B.2.2 WS-Trust: Correspondences and Secrecy Assertions

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wstrust_corresp.tf — Model of Trust/SecureConversation

// Part 1: SCT establishment, RST/RSTR exchange

// This file contains a top−level process for modelling exchanges of RST/RSTR,
// plus queries for correspondences of C1 and C2.

import "library.tf".
import "common.tf".
import "wstrust.tf".

channel soap(item), publish(string).

// Basic functional properties.
//query end:C1(C10,C11,"Server",C13,C14,C15).
query begin:C1(C11,C12,C13,C14).
query end:C1(C11,C12,C13,C14).
// query begin:C1(C1e1,C1e2,C1e3,C1e4,C1e5,C1e6,C1e7,"Server",C1e9).
// query end:C1(C11,C12,C13,C14,C15,C16,C17,"Both",C19).
// query begin:C1(C1e1,C1e2,C1e3,C1e4,C1e5,C1e6,C1e7,"Both",C1e9).
// query end:C1(C1u,C1sc,C1uid,C1clent) ==> LeakPrincipal(C1u).
// query end:C1(C1u,C1sc,C1uid,C1clent,"Server") ==> begin:C1(C1u,C1sc,C1uid,C1clent,"Server") |LeakPrincipal(C1u).
// query end:C1(C1u,C1sc,C1uid,C1clent,"Server") ==> LeakPrincipal(C1u).


// Theorem 1 part 1
 query end:C1(C1u,C1sc,C1uid,C1clent) ==> begin:C1(C1u,C1sc,C1uid,C1clent) |LeakPrincipal(C1u).

// // C2: RSTR correspondence + correlation with RST between client and STS
  query end:C2(C20,C21,C22,C23,C24).
  query begin:C2(C2e0,C2e1,C2e2,C2e3,C2e4).
// query end:C2(C21,C22,C23,C24,C25,"Both").
// query begin:C2(C2e1,C2e2,C2e3,C2e4,C2e5,"Server").
// query end:C2(C2u,C2sts,C2sc,C2uid1,C2uid2) ==>
// LeakPrincipal(C2u) |LeakPrincipal(C2sts).


// Theorem 1 part 2
  query end:C2(C2u,C2sts,C2sc,C2uid1,C2uid2) ==>
      begin:C2(C2u,C2sts,C2sc,C2uid1,C2uid2) |
      LeakPrincipal(C2u) |LeakPrincipal(C2sts).


// Theorem 2
//private name clientEntropy:bytes.
//private name stsEntropy:bytes.
//secret clientEntropy.
//secret stsEntropy.

// ================================
// Simulation

simulate with 0.

new sr:bytes; let kr = pk(sr);
out publish(base64(kr));

  ( !MkPrincipal()
    |!MkPrincipalCert(sr)
    |!MkPrincipalLeak(kr)
    |!MkSTS(kr)
    |!ClientSC(kr)
    |!STS(kr)
  )
```

B.3 Predicates and Processes for WS-SecureConversation

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wssecconv_common.tf — Model of Trust/SecureConversation

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

// ============================
// Envelope template for Messages

predicate env(Env:item,DestInfo:items,t:string,
              sig:item,ebody:item,sctid:string,
              mode:item) :—
  Env = <Envelope>
          <Header>
            <Security>
              <Timestamp><Created>t</></>
              sct
              sig</> @ DestInfo</>
          <Body>ebody</></>,
  SCT(sct,sctid),
  nonDerivedKeyMode(mode).

predicate env(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).


predicate envb(Env:item,DestInfo:items,t:string,
              sig:item,ebody:item,sctid:string,
              mode:item) :—
  nonDerivedKeyMode(mode),
  SCT(sct,sctid),
  Env =
    <Envelope>
      <Header>
        <Security>
          <Timestamp><Created>t</></>
          sct
          sig</> @ DestInfo</>
      <Body>ebody</></>.


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

predicate deriveKey(dk:bytes,key:bytes,nonce:bytes):−
  dk = psha1(base64(nonce),key).

predicate computeKeys(mode:item,key:bytes,EncKey:bytes,SigKey:bytes):−
  derivedKeyMode(mode,EncNonce,SigNonce),
  deriveKey(EncKey,key,EncNonce),
  deriveKey(SigKey,key,SigNonce).

predicate computeKeys(mode:item,key:bytes,EncKey:bytes,SigKey:bytes):−
  nonDerivedKeyMode(mode),
  EncKey = key,
  SigKey = key.

// ==============================
// Make Message
predicate mkEnv(msg:item,EnvelopeInfo:item,sc:item,mode:item) :−
  EnvInfo(EnvelopeInfo,t,sctid,DestInfo,b),
  SC(sc,sctid,sckey,entropyMode,UserToken,StsInfo,appTo,extra),
  computeKeys(mode,sckey,EncKey,SigKey),
  mkEncryptedDataSym(ebody,b,EncKey),
  mkSignature(sig,"hmacsha1",SigKey,
              [ <Body>ebody</>
                <Created>t</>
                @ DestInfo ]),
  envb(msg,DestInfo,t,sig,ebody,sctid,mode).

// ==============================
// Check Message

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

predicate sequence(Sequence:item,sessionId:string,msgNumber:bytes):−
  Sequence = <Sequence>
               <Identifier>sessionId</>
               <MessageNumber>base64(msgNumber)</></>.

predicate destInfo(DestInfo:items,srvURI:item,srvAC:item,sessionId:string,msgNumber:bytes):−
  DestInfo = [<To>srvURI</> <Action>srvAC</>
              <Sequence>
                <Identifier>sessionId</>
                <MessageNumber>base64(msgNumber)</></>].

predicate destInfoResponse(DestInfoResponse:items,srvURI:item,sessionId:string,msgNumber:bytes):−
  DestInfoResponse = [<From>srvURI</>
                      <SequenceAcknowledgement>
                        <Identifier>sessionId</>
                        <MessageNumber>base64(msgNumber)</></>].

// ==========================
channel genService(item).
private channel anyService(item).
process MkService() =
  in genService(SrvInfo);
  filter isService(SrvInfo,srvURI,srvAC)  → srvURI,srvAC;
  !out anyService(SrvInfo).

private channel anySCClient(item).
private channel anySCService(item).



channel initMagic(string,item,item,string,item).
process magicSC() =
  in initMagic(mode,UserToken,StsInfo,appTo,extra);
  new sckey:bytes;
  new sctid:string;
  filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra) → sc;
  out publish(sctid);
  (!out anySCService(sc) |!out anySCClient(sc)).


event LeakSC(item).
channel initLeak(string).
process SCLeak() =
  in initLeak(sctid);
  in anySCClient(sc);

  event LeakSC(sc);
  out publish(base64(c14n(sc))).

// removed srvURI:string,srvAC:string,id:string
predicate EnvInfo(msg:item,t:string,sctid:string,DestInfo:items,b:item) :−
  msg =
    <Envelope>
      <Header>
        <Security>
          <Timestamp><Created>t</></>
          <SecurityContextToken>
           <Identifier>sctid</>
         </>
        </> @
        DestInfo
      </>
      <Body>b</></>.

// predicate partialEnvResp(msg:item,t:string,sctid:string,srvURI:string,id1:string,id2:string,b:item) :−
// msg =
// <Envelope>
// <Header>
// <Security>
// <Timestamp><Created>t</></>
//  sct
// </>
//  <From>srvURI</>
//  <RelatesTo>id1</>
//  <MessageId>id2</>
// </>
// <Body>b</></>,
// SCT(sct,sctid).

predicate derivedKeyMode(mode:item,EncNonce:bytes,SigNonce:bytes) :−
  mode = <DKMode>
           <EncryptionNonce>base64(EncNonce)</>
           <SignatureNonce>base64(SigNonce)</>
         </>.

predicate nonDerivedKeyMode(mode:item) :−
  mode = <NonDerivedKeyMode></>.

predicate isDKMode(mode:item,modestr:string) :−
   mode = <DKMode>
           <EncryptionNonce>base64(EncNonce)</>
           <SignatureNonce>base64(SigNonce)</>
         </>,
   modestr = "yes".
```

```
predicate isDKMode(mode:item, modestr:string) :−
   modestr = "no".
```

B.4 Authentication and Secrecy for a Two Message Conversation (Theorem 3)

B.4.1 Two Message Conversation: Main Processes

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wssecconv.tf − Model of Trust/SecureConversation
// Part 2: using the SC

//This file contains predicates to create envelopes for messages 1 and 2, exchanged using the SCT.

// This file contains parts included by draft.tex

event C3(item,item,item).
event C4(item,item,item,item,item).

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

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

// Service
channel acceptService(string,string).
process Service(SrvInfo:item) =
  in soap(msg1);
  in anySCService(sc);
  filter isService(SrvInfo,srvURI,srvAC) → srvAC,srvURI;

  filter isEnv(msg1,Request,sc,RequestMode)  → Request, RequestMode;
  end C3 (Request,sc,RequestMode);

  filter EnvInfo(Request,t1,sctid,DestInfo,b)  → t1,sctid,DestInfo,b;
  filter destInfoReq(DestInfo,srvURI,srvAC,id1) → id1;

  new EncNonce2:bytes;
  new SigNonce2:bytes;
  in acceptService (id2,t2);

  filter destInfoResp(DestInfoResponse,srvURI,id1,id2) → DestInfoResponse;
  filter isDKMode(RequestMode,DerivedKeyMode)  → DerivedKeyMode;
  if DerivedKeyMode = "yes" then
  (
   filter derivedKeyMode(ResponseMode,EncNonce2,SigNonce2)  → ResponseMode;
   filter EnvInfo(Response,t2,sctid,DestInfoResponse,b2)  → Response;
   begin C4 (Request,sc,RequestMode,Response,ResponseMode);

   filter mkEnv(msg2,Response,sc,ResponseMode)  → msg2;
   out soap(msg2))
  else
  (
   if DerivedKeyMode = "no" then
   (

   filter EnvInfo(Response,t2,sctid,DestInfoResponse,b2)  → Response;
   begin C4 (Request,sc,RequestMode,Response,RequestMode);

   filter mkEnv(msg2,Response,sc,RequestMode)  → msg2;
   out soap(msg2)
  )).
```

```
// ============================
// Act as a client

channel initClient(item,string).
process Client() =
  in initClient (Request,DerivedKeyMode);

  filter EnvInfo(Request,t,sctid,DestInfoGiven,givenB) → t,sctid,DestInfoGiven,givenB;
   new EncNonce1:bytes;
   new SigNonce1:bytes;
   new id1:string;
  filter destInfoReq(DestInfoGiven,srvURI,srvAC,givenId) → srvURI,srvAC,givenId;
  filter destInfoReq(DestInfo,srvURI,srvAC,id1) → DestInfo;
  filter EnvInfo(RequestIdB,t,sctid,DestInfo,b1) → RequestIdB;

  in anySCClient(sc);

  if DerivedKeyMode="yes" then
  (
   filter derivedKeyMode(RequestMode,EncNonce1,SigNonce1) → RequestMode;
   begin C3 (RequestIdB,sc,RequestMode);

   filter mkEnv(msg1,RequestIdB,sc,RequestMode) → msg1;

   out soap(msg1);
   in soap(msg2);
   filter isEnv(msg2,Response,sc,ResponseMode) → Response,ResponseMode;

   filter EnvInfo(Response,t2,sctid,DestInfoResponse,b) → t2,DestInfoResponse,b;
   filter destInfoResp(DestInfoResponse,srvURI,id1,id2) → id2;
   filter derivedKeyMode(ResponseMode,EncNonce2,SigNonce2) → EncNonce2,SigNonce2;

   end C4 (RequestIdB,sc,RequestMode,Response,ResponseMode)
  )
  else
  (
   filter nonDerivedKeyMode(RequestMode) → RequestMode;
   begin C3 (RequestIdB,sc,RequestMode);
   filter mkEnv(msg1,RequestIdB,sc,RequestMode) → msg1;

   out soap(msg1);
   in soap(msg2);
   filter isEnv(msg2,Response,sc,RequestMode) → Response;

   filter EnvInfo(Response,t2,sctid,DestInfoResponse,b) → t2,DestInfoResponse,b;
   filter destInfoResp(DestInfoResponse,srvURI,id1,id2) → id2;
   end C4 (RequestIdB,sc,RequestMode,Response,RequestMode)
  ).
```

B.4.2 Two Message Conversation: Correspondences and Secrecy Assertions

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wssecconv_corresp.tf — Model of Trust/SecureConversation

// Part 2: using the SCT

import "library.tf".
import "common.tf".
import "wssecconv_common.tf".
import "wssecconv.tf".

channel soap(item), publish(string).

// secret bodies
private name b1:item.
private name b2:item.

// Basic functional properties
```

```
//query begin:C3(C3partialEnv,C3sc,C3mode).
//query end:C3(C3partialEnv,C3sc,C3mode).
// query end:C3(C3partialEnv,C3sc,C3mode) ==> LeakSC(C3sc).

// query begin:C4(C3partialEnv,C3sc,C3mode,C4partialEnvResp,C4mode2).
// query end:C4(C3partialEnv,C3sc,C3mode,C4partialEnvResp,C4mode2).
// query end:C4(C3partialEnv,C3sc,C3mode,C4partialEnvResp,C4mode) ==> LeakSC(C4sc).

// Theorem 3 part 1
query end:C3(C3partialEnv,C3sc,C3mode) ==> begin:C3(C3partialEnv,C3sc,C3mode) |LeakSC(C3sc).

// Theorem 3, part 2
  query end:C4(C3partialEnv,C3sc,C3mode1,C4partialEnvResp,C4mode2) ==>
  begin:C4(C3partialEnv,C3sc,C3mode1,C4partialEnvResp,C4mode2) |LeakSC(C4sc).


// Theorem 3, part 3
secret b1.
secret b2.


// ================================
// Simulation
simulate with 0.

  ( !magicSC() |
   !SCLeak() |
   !MkService() |
   (!in anyService(SrvInfo); Service(SrvInfo) ) |
   (!Client()))
```

B.5 Authentication and Secrecy for an Open-Ended Conversation (Theorem 4)

B.5.1 Open-Ended Conversation: Main Processes

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wssecconvn.tf — Model of Trust/SecureConversation

//This file contains predicates to create envelopes for messages 1 and 2, exchanged using the SCT.

predicate collect(newbody:item,oldbodies:items,newbodies:items) :−
  newbodies = [newbody @ oldbodies].
// newbodies = [newbody].

event C3(item,item,bytes,bytes,bytes,string,string,items).
event C4(item,item,bytes,bytes,bytes,string,string,bytes,bytes,string,items).

channel acceptService(string,item).

process Service() =
  in soap(msg1);
  in sessionDbS(SrvInfo,sc,sessionId,lastbodies,msgNumber);
  filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
   → sctid,sckey,mode,UserToken,StsInfo,appTo,extra;
  filter isEnv(msg1,DestInfo,"yes",EncNonce1,SigNonce1,sctid,sckey,t1,b1)
   → DestInfo,EncNonce1,SigNonce1,t1,b1;
  filter isService(SrvInfo,srvURI,srvAC) → srvAC,srvURI;

  filter destInfo(DestInfo,srvURI,srvAC,sessionId,msgNumber) → ;

  filter collect(b1,lastbodies,bodies)  → bodies;
  end C3 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,bodies);

  in acceptService (t2,b2);
  new EncNonce2:bytes;
  new SigNonce2:bytes;

  filter collect(b2,bodies,newbodies)  → newbodies;
  begin C4 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,EncNonce2,SigNonce2,t2,newbodies);

  filter destInfoResponse(DestInfoResponse, srvURI,sessionId,msgNumber) → DestInfoResponse;

  filter mkEnv(msg2,DestInfoResponse,"yes",EncNonce2,SigNonce2,sctid,sckey,t2,b2)
    → msg2;
  out soap(msg2);
  out sessionDbS(SrvInfo,sc,sessionId,newbodies,succ(msgNumber)).

// ============================
// Act as a client

channel initClient(string,item).

process Client() =
  in initClient (t1,b1);
  in sessionDb(SrvInfo,sc,sessionId,lastbodies,msgNumber);
  filter isService(SrvInfo,srvURI,srvAC) → srvAC,srvURI;
  filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
   → sckey,mode,UserToken,StsInfo,appTo,extra,sctid;

  new EncNonce1:bytes;
  new SigNonce1:bytes;

  filter collect(b1,lastbodies,bodies)  → bodies;
  begin C3 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,bodies);

  filter destInfo(DestInfo,srvURI,srvAC,sessionId,msgNumber) → DestInfo;

  filter mkEnv(msg1,DestInfo,"yes",EncNonce1,SigNonce1,sctid,sckey,t1,b1)  → msg1;
  out soap(msg1);

  in soap(msg2);
  filter isEnv(msg2,DestInfoResponse,"yes",EncNonce2,SigNonce2,sctid,sckey,t2,b2)
    → DestInfoResponse,EncNonce2,SigNonce2,t2,b2;

   filter destInfoResponse(DestInfoResponse,srvURI,sessionId,msgNumber) → ;

  filter collect(b2,bodies,newbodies)  → newbodies;
  end C4 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,EncNonce2,SigNonce2,t2,newbodies);
  out sessionDb(SrvInfo,sc,sessionId,newbodies,succ(msgNumber)).

// Abstractly...
//
// new sessionDb;
// C[
// new sessionId:string;
// out sessionDb(...,sessionId,zero()))
// ][
// !in sessionDb(...,sessionId,msgNumber);
// ...(linear)...;
// begin C3 (...,msgNumber,sessionId);
// ...(linear)...;
// out sessionDb(...,sessionId,succ(msgNumber)).
// ]
```

B.5.2 Open-Ended Conversation: Correspondences and Secrecy Assertions

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// wssecconvn_corresp.tf — Model of Trust/SecureConversation

// Part 2: using the SCT

import "library.tf".
import "common.tf".
import "wssecconvn_common.tf".
import "wssecconvn.tf".

constructor zero():bytes.
constructor succ(bytes):bytes.
//destructor succ(bytes):bytes with succ(b)=b.

channel soap(item), publish(string).
// // private
// name b1:item.
// //private
// name b2:item.

// Functional properties
// query end:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies).
// query begin:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies).
query begin:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies).
// query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies).
// query LeakSC(SSC).
// // query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies) ==> LeakSC(C3sc).

// query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies)
// ==> begin:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies) |LeakSC(C3sc).

// query end:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies)
// ==> begin:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies) |
LeakSC(C4sc).

// secret b1.
// secret b2.

// ================================
// Simulation

simulate with 0.


channel ContextProvideSessionId(string).
channel newSession(string,item).
private channel sessionDbS(item,item,string,items,bytes).
private channel sessionDb(item,item,string,items,bytes).

( !MkService() |!magicSC() |!SCLeak() |

  // services accept new session on any SC
  (!in anyService(SrvInfo);
   in anySCService(sc);
  // as a simplification, we ask the context to provide the sessionId
   in ContextProvideSessionId(sessionId);// with anti−replay cache
   out sessionDbS(SrvInfo,sc,sessionId,[],zero()))

  // services receive further messages
  |!Service()

  // clients can start new sessions on any SC
  |(!in newSession(sctid,SrvInfo);
     in anySCClient(sc);
     filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
    → sckey,mode,UserToken,StsInfo,appTo,extra;
     new sessionId:string;
     out sessionDb(SrvInfo,sc,sessionId,[],zero()))

  // clients send messages using current sessions
  |!Client()

)
```

B.6 Authentication and Secrecy for an Abstract Open-Ended Conversation

B.6.1 Open-Ended Conversation Abstract Script: Main Processes

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// Ricardo: Model of Trust/SecureConversation

//This file contains predicates to create envelopes for messages 1 and 2, exchanged using the SCT.

predicate collect(newbody:item,oldbodies:items,newbodies:items) :−
// newbodies = [newbody @ oldbodies].
    newbodies = [newbody].

event C3(item,item,bytes,bytes,bytes,string,string,items).
event C4(item,item,bytes,bytes,bytes,string,string,bytes,bytes,string,items).

channel acceptService(string,item).
channel acceptExtra(bytes,items).

process Service() =
  in soap(msg1);

  in sessionDbS(SrvInfo,sc,sessionId);
  in acceptExtra(msgNumber,lastbodies);
  filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
   → sctid,sckey,mode,UserToken,StsInfo,appTo,extra;
  filter isEnv(msg1,DestInfo,"yes",EncNonce1,SigNonce1,sctid,sckey,t1,b1)
   → DestInfo,EncNonce1,SigNonce1,t1,b1;
  filter isService(SrvInfo,srvURI,srvAC) → srvAC,srvURI;

  filter destInfo(DestInfo,srvURI,srvAC,sessionId,msgNumber) → ;

  filter collect(b1,lastbodies,bodies)  → bodies;
  end C3 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,bodies);

  in acceptService (t2,b2);

  new EncNonce2:bytes;
  new SigNonce2:bytes;

  filter collect(b2,bodies,newbodies)  → newbodies;
  begin C4 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,EncNonce2,SigNonce2,t2,newbodies);

  filter destInfoResponse(DestInfoResponse, srvURI,sessionId,msgNumber) → DestInfoResponse;

  filter mkEnv(msg2,DestInfoResponse,"yes",EncNonce2,SigNonce2,sctid,sckey,t2,b2)
    → msg2;
  out soap(msg2);
  out sessionDbS(SrvInfo,sc,sessionId).

// ============================
// Act as a client

channel initClient(string,item,items,bytes). // two extra args

process Client() =
  in initClient (t1,b1,lastbodies,msgNumber);

  in sessionDb(SrvInfo,sc,sessionId);
  filter isService(SrvInfo,srvURI,srvAC) → srvAC,srvURI;
  filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
   → sckey,mode,UserToken,StsInfo,appTo,extra,sctid;

  new EncNonce1:bytes;
  new SigNonce1:bytes;
  filter collect(b1,lastbodies,bodies)  → bodies;
  begin C3 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,bodies);

  filter destInfo(DestInfo,srvURI,srvAC,sessionId,msgNumber) → DestInfo;

  filter mkEnv(msg1,DestInfo,"yes",EncNonce1,SigNonce1,sctid,sckey,t1,b1)  → msg1;
  out soap(msg1);

  in soap(msg2);
  filter isEnv(msg2,DestInfoResponse,"yes",EncNonce2,SigNonce2,sctid,sckey,t2,b2)
    → DestInfoResponse,EncNonce2,SigNonce2,t2,b2;

   filter destInfoResponse(DestInfoResponse,srvURI,sessionId,msgNumber) → ;

  filter collect(b2,bodies,newbodies)  → newbodies;
  end C4 (SrvInfo,sc,EncNonce1,SigNonce1,msgNumber,sessionId,t1,EncNonce2,SigNonce2,t2,newbodies);
  out sessionDb(SrvInfo,sc,sessionId).

// Abstractly...
//
// new sessionDb;
// C[
// new sessionId:string;
// out sessionDb(...,sessionId,zero()))
// ][
// !in sessionDb(...,sessionId,msgNumber);
// ...(linear)...;
// begin C3 (...,msgNumber,sessionId);
// ...(linear)...;
// out sessionDb(...,sessionId,succ(msgNumber)).
// ]
```

B.6.2 Open-Ended Conversation Abstract Script: Correspondences and Secrecy Assertions

```
// Copyright (C) Microsoft Corporation. All rights reserved.
// Ricardo: Model of Trust/SecureConversation

//

// Part 2: using the SCT

import "library.tf".
import "common.tf".
import "wssecconvn_common.tf".
import "wssecconva.tf".

constructor zero():bytes.
constructor succ(bytes):bytes.
//destructor succ(bytes):bytes with succ(b)=b.

  channel soap(item), publish(string).
// private
// name b1:item.
// private
// name b2:item.

// Checking reachability of events...
// query end:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies).
// query begin:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,lC4astbodies).
// query begin:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies).
// query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies).
// query LeakSC(SSC).
// query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies) ==> LeakSC(C3sc).

// Lemma 3
query end:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies) ==>
```

```
  begin:C3(C3SrvInfo,C3sc,C3EncNonce1,C3SigNonce1,C3msgNumber,C3sessionId,C3t1,C3lastbodies) |LeakSC(C3sc).

// Lemma 3
// query end:C4(C4SrvInfo,C4sc,C4EncNonce1,C4SigNonce1,C4msgNumber,C4sessionId,C4t1,C4EncNonce2,C4SigNonce2,C4t2,C4lastbodies)
// ==>begin:C4(C4SrvInfo,C4sc,C4EncNonce1x,C4SigNonce1x,C4msgNumberx,C4sessionId,C4t1x,C4EncNonce2,C4SigNonce2,C4t2,C4lastbodiesx) |
LeakSC(C4sc).

// Used in Theorem 5, second part
// secret b1.
// secret b2.

// ================================
// Simulation

simulate with 0.


channel ContextProvideSessionId(string).
channel newSession(string,item).
private channel sessionDbS(item,item,string).
private channel sessionDb(item,item,string).

( !MkService() |!magicSC() |!SCLeak() |

 // services accept new session on any SC
 (!in anyService(SrvInfo);
  in anySCService(sc);
 // as a simplification, we ask the context to provide the sessionId
  in ContextProvideSessionId(sessionId);
  out sessionDbS(SrvInfo,sc,sessionId))

 // services receive further messages
 |!Service()

 // clients can start new sessions on any SC
 |(!in newSession(sctid,SrvInfo);
   in anySCClient(sc);
   filter SC(sc,sctid,sckey,mode,UserToken,StsInfo,appTo,extra)
  → sckey,mode,UserToken,StsInfo,appTo,extra;
   new sessionId:string;
   out sessionDb(SrvInfo,sc,sessionId))

 // clients send messages using current sessions
 |!Client()
)
```