TulaFale: A Security Tool for Web Services

Karthikeyan Bhargavan, Cédric Fournet,
Andrew D. Gordon, and Riccardo Pucella

Microsoft Research

Abstract. Web services security specifications are typically expressed as a mixture of XML schemas, example messages, and narrative explanations. We propose a new specification language for writing complementary machine-checkable descriptions of SOAP-based security protocols and their properties. Our TulaFale language is based on the pi calculus (for writing collections of SOAP processors running in parallel), plus XML syntax (to express SOAP messaging), logical predicates (to construct and filter SOAP messages), and correspondence assertions (to specify authentication goals of protocols). Our implementation compiles TulaFale into the applied pi calculus, and then runs Blanchet's resolution-based protocol verifier. Hence, we can automatically verify authentication properties of SOAP protocols.

1 Verifying Web Services Security

Web services are a wide-area distributed systems technology, based on asynchronous exchanges of XML messages conforming to the SOAP message format [BEK+00,W3C03]. The WS-Security standard [NKHBM04] describes how to sign and encrypt portions of SOAP messages, so as to achieve end-to-end security. This paper introduces TulaFale, a new language for defining and automatically verifying models of SOAP-based cryptographic protocols, and illustrates its usage for a typical request/response protocol: we sketch the protocol, describe potential attacks, and then give a detailed description of how to define and check the request and response messages in TulaFale.

1.1 Web Services

A basic motivation for web services is to support programmatic access to web data. The HTML returned by a typical website is a mixture of data and presentational markup, well suited for human browsing, but the presence of markup makes HTML a messy and brittle format for data processing. In contrast, the XML returned by a web service is just the data, with some clearly distinguished metadata, well suited for programmatic access. For example, search engines export web services for programmatic web search, and e-commerce sites export web services to allow affiliated websites direct access to their databases.

Generally, a broad range of applications for web services is emerging, from the well-established use of SOAP as a platform and vendor neutral middleware

within a single organisation, to the proposed use of SOAP for device-to-device interaction [S+04].

In the beginning, "SOAP" stood for "Simple Object Access Protocol", and was intended to implement "RPC using XML over HTTP" [Win98,Win99,Box01]. HTTP facilitates interoperability between geographically distant machines and between those in protection domains separated by corporate firewalls that block many other transports. XML facilitates interoperability between different suppliers' implementations, unlike various binary formats of previous RPC technologies. Still, web services technology should not be misconstrued as HTTP-specific RPC for distributed objects [Vog03]. HTTP is certainly at present the most common transport protocol, but the SOAP format is independent of HTTP, and some web services use other transports such as TCP or SMTP [SMWC03]. The design goals of SOAP/1.1 [BEK+00] explicitly preclude object-oriented features such as object activation and distributed garbage collection; by version 1.2 [W3C03], "SOAP" is a pure name, not an acronym. The primitive message pattern in SOAP is a single one-way message that may be processed by zero or more intermediaries between two end-points; RPC is a derived message pattern built from a request and a response. In brief, SOAP is not tied to objects, and web services are not tied to the web. Still, our running example is an RPC over HTTP, which still appears to be the common case.

1.2   Securing Web Services with Cryptographic Protocols

Web services specifications support SOAP-level security via a syntax for embedding cryptographic materials in SOAP messages. To meet their security goals, web services and their clients can construct and check *security headers* in messages, according to the WS-Security format [IM02,NKHBM04]. WS-Security can provide message confidentiality and authentication independently of the underlying transport, using, for instance, secure hash functions, shared-key encryption, or public-key cryptography. WS-Security has several advantages compared to using a secure transport such as SSL, including scalability, flexibility, transparency to intermediaries such as firewalls, and support for non-repudiation. Significantly, though, WS-Security does not itself prescribe a particular security protocol: each application must determine its security goals, and process security headers accordingly.

Web services may be vulnerable to many of the well-documented classes of attack on ordinary websites [SS02,HL03]. Moreover, unlike typical websites, web services relying on SOAP-based cryptographic protocols may additionally be vulnerable to a new class of *XML rewriting attacks*: a range of attacks in which an attacker may record, modify, replay, and redirect SOAP messages, but without breaking the underlying cryptographic algorithms. Flexibility comes at a price in terms of security, and it is surprisingly easy to misinterpret the guarantees actually obtained from processing security headers. XML is hence a new setting for an old problem going back to Needham and Schroeder's pioneering work on authentication protocols; SOAP security protocols should be judged safe, or not, with respect to an attacker who is able to "interpose a computer on

2

all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material" [NS78]. XML rewriting attacks are included in the WS–I threat model [DHK+04]. We have found a variety of replay and impersonation attacks in practice.

1.3   Formalisms and Tools for Cryptographic Protocols

The use of formal methods to analyze cryptographic protocols and their vulnerabilities begin with work by Dolev and Yao [DY83]. In the past few years there has been intense research on the Dolev-Yao model, leading to the development of numerous formalisms and tools.

TulaFale builds on the line of research using the pi calculus. The pi calculus [Mil99] is a general theory of interaction between concurrent processes. Several variants of the pi calculus, including spi [AG99], and a generalization, applied pi [AF01], have been used to formalize and prove properties of cryptographic protocols. A range of compositional reasoning techniques is available for proving protocol properties, but proofs typically require human skill and determination. Recently, however, Blanchet [Bla01,Bla02] has proposed a range of automatic techniques, embodied in his theorem prover ProVerif, for checking certain secrecy and authentication properties of the applied pi calculus. ProVerif works by compiling the pi calculus to Horn clauses and then running resolution-based algorithms.

1.4   TulaFale: A Security Tool for Web Services

TulaFale is a new scripting language for specifying SOAP security protocols, and verifying the absence of XML rewriting attacks:

TulaFale = processes + XML + predicates + assertions

The pi calculus is the core of TulaFale, and allows us to describe SOAP processors, such as clients and servers, as communicating processes. We extend the pi calculus with a syntax for XML plus symbolic cryptographic operations; hence, we can directly express SOAP messaging with WS-Security headers. We declaratively specify the construction and checking of SOAP messages using Prolog-style predicates; hence, we can describe the operational details of SOAP processing. Independently, we specify security goals using various assertions, such as correspondences for message authentication and correlation.

It is important that TulaFale can express the detailed structure of XML signatures and encryption so as to catch low-level attacks on this structure, such as copying part of an XML signature into another; more abstract representations of message formats, typical in the study of the Dolev-Yao model and used for instance in previous work on SOAP authentication protocols [GP03], are insensitive to such attacks.

Our methodology when developing TulaFale has been to study particular web services implementations, and to develop TulaFale scripts modelling their
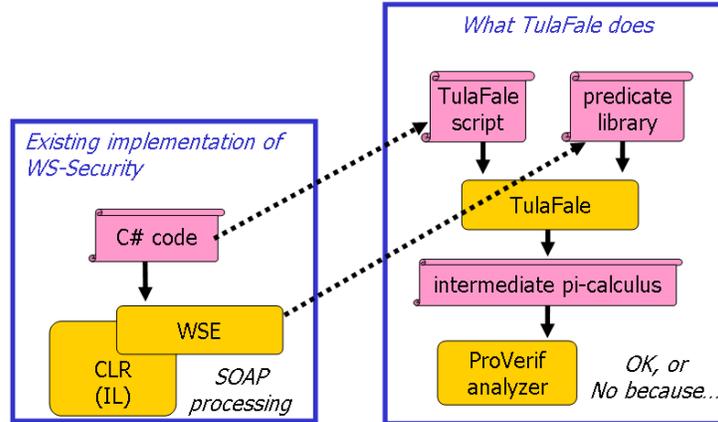
Fig. 1. Modelling WS-Security protocols with TulaFale

security aspects. Our experiments have been based on the WSE development kit [Mic02], a particular implementation of WS-Security and related specifications. We have implemented the running example protocol of this paper using WSE, and checked that the SOAP messages specified in our script faithfully reflect the SOAP messages observed in this implementation. For a discussion of the implementation of related protocols, including logs of SOAP messages, see the technical report version of an earlier paper [BFG04a].

Fig. 1 illustrates our methodology. On the left, we have the user-supplied code for implementing a web services protocol, such as the one of this paper, on top of the WSE library. On the right, we have the TulaFale script modelling the user-supplied code, together with some library predicates modelling operations performed by WSE. Also on the right, we have the TulaFale tool, which compiles its input scripts into the pure applied pi calculus, which is then analyzed via ProVerif.

TulaFale is a direct implementation of the pi calculus described in a previous formal semantics of web services authentication [BFG04a]. The original contribution of this paper is to present a concrete language design, to report an implementation of automatic verification of assertions in TulaFale scripts using Blanchet's ProVerif, and to develop a substantial example.

Section 2 informally introduces a simple request/response protocol and its security goals: authentication and correlation of the two messages. Section 3 presents TulaFale syntax for XML with symbolic cryptography and for predicates, and as a source of examples, explains a library of TulaFale predicates for constructing and checking SOAP messages. Section 4 describes predicates specific to the messages of our request/response protocol. Section 5 introduces processes and security assertions in TulaFale, and outlines their implementation

via ProVerif. Section 6 describes processes and predicates specific to our protocol, and shows how to verify its security goals. Finally, Section 7 concludes.

2 A Simple Request/Response Protocol

We consider a simple SOAP-based request/response protocol, of the kind easily implemented using WSE to make an RPC to a web service. Our security goals are simply message authentication and correlation. To achieve these goals, the request includes a *username token* identifying a particular user and a *signature token* signed by a key derived from user's password; conversely, the response includes a signature token signed by the server's public key. Moreover, to preserve the confidentiality of the user's password from dictionary attacks, the username token in the request message is encrypted with the server's public key. (For simplicity, we are not concerned here with any secrecy properties, such as confidentiality of the actual message bodies, and we do not model any authorization policies.)

In the remainder of this section, we present a detailed but informal specification of our intended protocol, and consider some variations subject to XML rewriting attacks. Our protocol involves the following principals:

- A single certification authority (CA) issuing X.509 public-key certificates for services, signed with the CA's private key.
- Two servers, each equipped with a public key certified by the CA and exporting an arbitrary number of web services.
- Multiple clients, acting on behalf of human users.

Trust between principals is modelled as a database associating passwords to authorized user names, accessible from clients and servers. Our threat model features an active attacker in control of the network, in possession of all public keys and user names, but not in possession of any of the following:

(1) The private key of the CA.
(2) The private key of any public key certified by the CA.
(3) The password of any user in the database.

The second and third points essentially rule out "insider attacks"; we are assuming that the clients, servers, and CA belong to a single close-knit institution. It is easy to extend our model to study the impact of insider attacks, and indeed to allow more than two servers, but we omit the details in this expository example.

Fig. 2 shows an intended run of the protocol between a client and server.

- The principal Client(kr,U) acts on behalf of a user identified by U (an XML encoding of the username and password). The parameter kr is the public key of the CA, needed by the client to check the validity of public key certificates.
- The principal Server(sx,cert,S) implements a service identified by S (an XML encoding of a URL address, a SOAP action, and the subject name appearing on the service's certificate). The parameter sx is the server's private signing key, while cert is its public certificate.
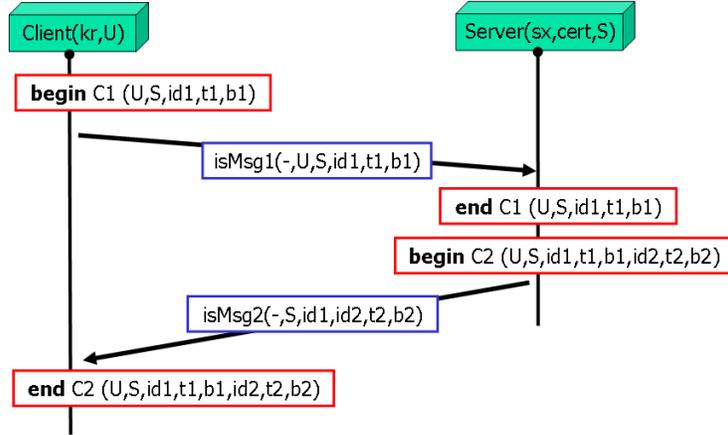
Fig. 2. An intended run of a client and server

- The client sends a request message satisfying isMsg1($-$,U,S,id1,t1,b1), which we define later to mean the message has body b1, timestamp t1, and message identifier id1, is addressed to a web service S, and has a `<Security>` header containing a token identifying U and encrypted with S's public key, and a signature of S, id1, t1, and b1 by U.
- The server sends a response message satisfying isMsg2($-$,S,id1,id2,t2,b2), which we define later to mean the message has body b2, timestamp t2, and message identifier id2, is sent from S, and has a `<Security>` header containing S's certificate cert and a signature of id1, id2, t2, and b2 by S.
- The client and server enact begin- and end-events labelled C1(U,S,id1,t1,b1) to record the data agreed after receipt of the first message. Similarly, the begin- and end-events labelled C2(U,S,id1,t1,b1,id2,t2,b2) record the data agreed after both messages are received. Each begin-event marks an intention to send data. Each end-event marks apparently successful agreement on data.

The begin- and end-events define our authentication and correlation goals: for every end-event with a particular label, there is a preceding begin-event with the same label in any run of the system, even in the presence of an active attacker. Such goals are known as one-to-many correspondences [WL93] or non-injective agreements [Low97]. The C1 events specify authentication of the request, while the C2 events specify authentication of the response. By including data from the request, C2 events also specify correlation of the request and response.

Like most message sequence notations, Fig. 2 simply illustrates a typical protocol run, and is not in itself an adequate specification. In Sections 4 and 6 we present a formal specification in TulaFale: we define the principals Client(kr,U) and Server(sx,cert,S) as parametric processes, and we define the checks isMsg1

Suppose a client does not sign the timestamp t1...

Client(kr,U)   Opponent   Server(sx,cert,S)

begin C1 (U,S,id1,t1,b1)

isMsg1(-,U,S, id1,t1,b1)   Copy

end C1 (U,S,id1,t1,b1)

Wait till t2, t1:=t2, Replay

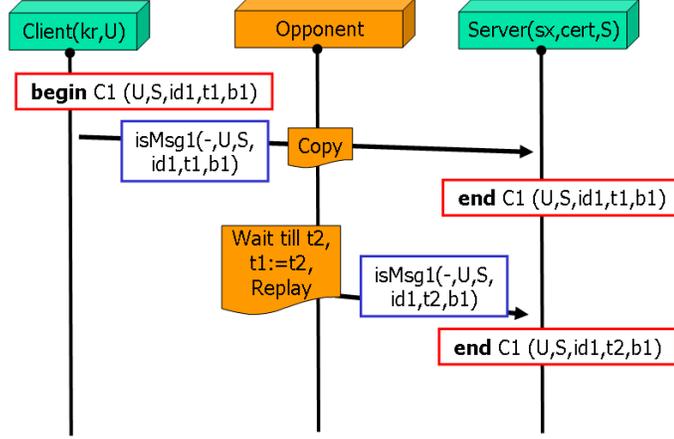isMsg1(-,U,S, id1,t2,b1)

end C1 (U,S,id1,t2,b1)

Fig. 3. A replay attack

and isMsg2 as predicates on our model of XML with symbolic cryptography. The formal model clarifies the following points, which are left implicit in the figure:

- The client can arbitrarily choose which service S to call, and which data t1 and b1 to send. (In the formal model, we typically make such arbitrary choices by inputting the data from the opponent.) Conversely, the client must generate a fresh identifier id1 for each request, or else it is impossible to correlate the responses from two simultaneous requests to the same service.
- Similarly, the server can arbitrarily choose the response data id2, t2, and b2.

On the other hand, our formal model does not directly address replay protection. To rule out direct replays of correctly signed messages, we would need to specify that for each end-event there is a unique preceding begin-event with the same label. This is known as a one-to-one correspondence or injective agreement. In practice, we can protect against direct replays using a cache of recently received message identifiers and timestamps to ensure that no two messages are accepted with the same identifier and timestamp. Hence, if we can prove that the protocol establishes non-injective agreement on data including the identifiers and timestamps, then, given such replay protection, the protocol implementation also establishes injective agreement.

We end this section by discussing some flawed variations of the protocol, corresponding to actual flaws we have encountered in user code for web services.

- Suppose that the check isMsg1(−,U,S,id1,t1,b1) only requires that S, id1, and b1, are signed by U, but not the timestamp t1. Replay protection based on the timestamp is now ineffective: the opponent can record a message with timestamp t1, wait until some time t2 when the timestamp has expired,
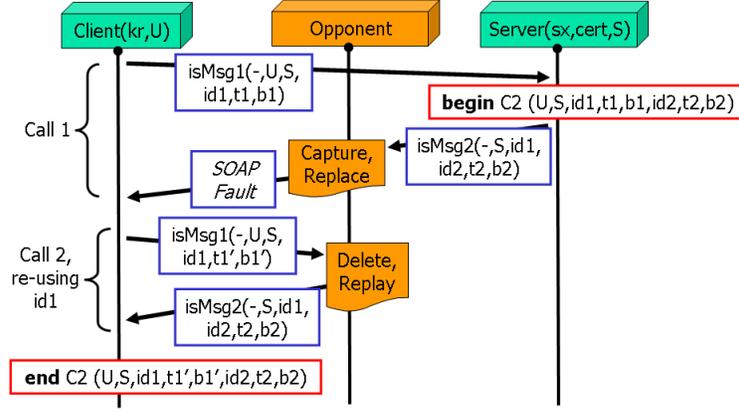
Fig. 4. A failure of message correlation

and the message identifier id1 is no longer being cached, rewrite the original message with timestamp t2, and then replay the message. The resulting message satisfies isMsg1($-$,U,S,id1,t2,b1), since t2 does not need to be signed, and hence is accepted by the server. Fig. 3 shows the attack, and the resulting failure of correspondence C1.

– Suppose that a client re-uses the same message identifier in two different calls to a web service; the opponent can manipulate messages so that the client treats the response to the first call as if it were the response to the second call. Fig. 4 shows the attack. The client sends a first request with body b1 and identifier id1. The opponent intercepts the response with body b2, and sends a SOAP fault back to the client. Subsequently, the client sends a second request with the same identifier id1 as the first, and body b1'. The opponent can delete this request to prevent it reaching the service, and then replay the original response. The client now considers that b2 is the response to b1', when in fact it is the response to b1, perhaps completely different. Formally, this is a failure of correspondence C2.

– Suppose that the server does not include the request identifier id1 in the signature on the response message. Then the opponent can mount a similar correlation attack, breaking C2—we omit the details.

We can easily adapt our TulaFale script to model these variations in the protocol. Our tool automatically and swiftly detects the errors, and returns descriptions of the messages sent during the attacks. These flaws in web services code are typical of errors in cryptographic protocols historically. The practical impact of these flaws is hard to assess, as they were found in preliminary code, before deployment. Still, it is prudent to eliminate these vulnerabilities, and tools such as TulaFale can systematically rule them out.

3 XML, Principals, and Cryptography in TulaFale

This section introduces the term and predicate language of TulaFale, via a series of basic constructions needed for the example protocol of Section 2. Throughout the paper, for the sake of exposition, we elide some details of SOAP envelopes, such as certain headers and attributes, that are unconnected to security.

3.1 XML Elements, Attributes, and Strings

Here is a TulaFale term for a SOAP request, illustrating the format of the first message in our example protocol:

```
<Envelope>
  <Header>
    <To>uri</>
    <Action>ac</>
    <MessageId>id</>
    <Security>
      <Timestamp><Created>"2004-03-19T09:46:32Z"</></>
      utok
      sig
    </>
  </>
  <Body Id="1">request</>
</>
```

Every SOAP message consists of an XML `<Envelope>` element, with two children: an optional `<Header>` and a mandatory `<Body>`. In this example, the header has four children, and the body has an Id-attribute, the literal string `"1"`.

We base TulaFale on a sorted (or typed) term algebra, built up from a set of function symbols and variables. The basic sorts for XML data include **string** (for string literals), **att** (for named attributes), and **item** (either an element or a string). Every element or attribute tag (such as Envelope or Id, for example) corresponds to a sorted function symbol in the underlying algebra.

Although TulaFale syntax is close to the XML wire format, it is not identical. We suppress all namespace information. As previously mentioned, we omit closing element tags; for example, we write `</>` instead of `</Envelope>`. Literal strings are always quoted, as in `<Created>"2004-03-19T09:46:32Z"</>`. In the standard wire format, the double quotes would be omitted when a string is an element body. We use quotation to distinguish strings from term variables, such as the variables uri, ac, id, utok, sig, and request in the example above.

3.2 Symbolic Cryptography

In TulaFale, we represent cryptographic algorithms symbolically, as function symbols that act on a sort **bytes** of byte arrays. Each function is either a data constructor, with no accompanying rewrite rule, or it is a destructor, equipped with

9

a rewrite rule for testing or extracting data from an application of a constructor. For example, encryption functions are constructors, and decryption functions are destructors. This approach, the basis of the Dolev-Yao model [DY83], assumes that the underlying cryptography is *perfect*, and can be faithfully reflected by abstract equational properties of the functions. It also abstracts some details, such as the lengths of strings and byte arrays. The TulaFale syntax for introducing constructors and destructors is based on the syntax used by ProVerif.

For instance, we declare function symbols for RSA key generation, public-key encryption, and private-key decryption using the following TulaFale declarations:

constructor pk(**bytes**):**bytes**.
constructor rsa(**bytes**,**bytes**):**bytes**.
destructor decrsa(**bytes**,**bytes**):**bytes** **with**
  decrsa(s,rsa(pk(s),b)) = b.

The constructor pk represents the relationship between private and public keys (both byte arrays, of sort **bytes**); it takes a private key and returns the corresponding public key. There is no inverse or destructor, as we intend to represent a one-way function: given only pk(s) it is impossible to extract s.

The constructor rsa(k,x) encrypts the data x:**bytes** under the public key k, producing an encrypted byte array. The destructor decrsa(s,e) uses the corresponding private key s to decrypt a byte array generated by rsa(pk(s),x). The destructor definition expresses the decryption operation as a rewrite rule: when an application of decrsa in a term matches the left-hand side of the rule, it may be replaced by the corresponding right-hand side.

To declare RSA public-key signatures, we introduce another constructor rsasha1(s,x) that produces a RSA signature of a cryptographic hash of data x under the private key s:

constructor rsasha1(**bytes**,**bytes**):**bytes**.
destructor checkrsasha1(**bytes**,**bytes**,**bytes**):**bytes** **with**
  checkrsasha1(pk(s),x,rsasha1(s,x))=pk(s).

To check the validity of a signature sig on x using a public key k, one can form the term checkrsasha1(k,x,sig) and compare it to k. If k is a public key of the form pk(s) and sig is the result of signing x under the corresponding private key s, then this term rewrites to k.

For the purposes of this paper, an X.509 certificate binds a key to a subject name by embedding a digital signature generated from the private key of some certifying authority (CA). We declare X.509 certificates as follows:

constructor x509(**bytes**,**string**,**string**,**bytes**):**bytes**.
destructor x509key(**bytes**):**bytes** **with**
  x509key(x509(s,u,a,k))=k.
destructor x509user(**bytes**):**string** **with**
  x509user(x509(s,u,a,k))=u.
destructor x509alg(**bytes**):**string** **with**
  x509alg(x509(s,u,a,k))=a.

destructor checkx509(**bytes**,**bytes**):**bytes** **with**
  checkx509(x509(sr,u,a,k),pk(sr))=pk(sr).

The term x509(sr,u,a,k) represents a certificate that binds the subject name u to the public key k, for use with the signature algorithm a (typically rsasha1). This certificate is signed by the CA with private key sr. Given such a certificate, the destructors x509key, x509user, and x509alg extract the three public fields of the certificate. Much like checkrsasha1 for ordinary digital signatures, an additional destructor checkx509 can be used to check the authenticity of the embedded signature.

3.3   XML Encryption and Decryption

Next, we write logical predicates to construct and parse XML encrypted under some known RSA public key. A predicate is written using a Prolog-like syntax; it takes a tuple of terms and checks logical properties, such as whether two terms are equal or whether a term has a specific format. It is useful to think of some of the terms given to the predicate as inputs and the others as outputs. Under this interpretation, the predicate computes output terms that satisfy the logical properties by pattern-matching.

The predicate mkEncryptedData takes a plaintext plain:**item** and an RSA public encryption key ek:**bytes**, and it generates an XML element encrypted containing the XML encoding of plain encrypted under ek.

predicate mkEncryptedData (encrypted:**item**,plain:**item**,ek:**bytes**) :−
  cipher = rsa(ek,c14n(plain)),
  encrypted = `<EncryptedData>`
                `<CipherData>`
                  `<CipherValue>`base64(cipher)`</></></>`.

The first binding in the predicate definition computes the encrypted byte array, cipher, using the rsa encryption function applied to the key ek and the plaintext plain. Since rsa is only defined over byte arrays, plain:**item** is first converted to **bytes** using the (reversible) c14n constructor. The second binding generates an XML element (`<EncryptedData>`) containing the encrypted bytes. Since only strings or elements can be embedded into XML elements, the encrypted byte array, cipher, is first converted to a string using the (reversible) base64 constructor.

In this paper, we use three transformation functions between sorts: c14n (with inverse ic14n) transforms an **item** to a **bytes**, base64 (with inverse ibase64) transforms a **bytes** to a **string**, and utf8 (with inverse iutf8) transforms a **string** to a **bytes**. All three functions have specific meanings in the context of XML transformations, but we treat them simply as coercion functions between sorts.

To process a given element, `<Foo>` say, we sometimes write distinct predicates on the sending side and on the receiving side of the protocol, respectively. By convention, to construct a `<Foo>` element, we write a predicate named mkFoo whose first parameter is the element being constructed; to parse and check it, we write a predicate named isFoo.

For `<EncryptedData>`, for instance, the logical predicate isEncryptedData parses elements constructed by mkEncryptedData; it takes an element encrypted and a decryption key dk:**bytes** and, if encrypted is an `<EncryptedData>` element with some plaintext encrypted under the corresponding encryption key pk(dk), it returns the plaintext as plain.

predicate isEncryptedData (encrypted:**item**,plain:**item**,dk:**bytes**) :−
  encrypted = `<EncryptedData>`
        `<CipherData>`
          `<CipherValue>`base64(cipher)`</></></>`,
  c14n(plain) = decrsa(dk,cipher).

Abstractly, this predicate reverses the computations performed by mkEncryptedData. One difference, of course, is that while mkEncryptedData is passed the public encryption key ek, the isEncryptedData predicate is instead passed the private key, dk. The first line matches encrypted against a pattern representing the `<EncryptedData>` element, extracting the encrypted byte array, cipher. Relying on pattern-matching, the constructor base64 is implicitly inverted using its destructor ibase64. The second line decrypts cipher using the decryption key dk, and implicitly inverts c14n using its destructor ic14n to compute plain.

3.4 Services and X509 Security Tokens

We now implement processing for the service identifiers used in Section 2. We identify each web service by a structure consisting of a `<Service>` element containing `<To>`, `<Action>`, and `<Subject>` elements. For message routing, the web service is identified by the HTTP URL uri where it is located and the name of the action ac to be invoked at the URL. (In SOAP, there may be several different actions available at the same uri.) The web service is then willing to accept any SOAP message with a `<To>` header containing uri and an `<Action>` header containing ac. Each service has a public key with which parts of requests may be encrypted, and parts of responses signed. The `<Subject>` element contains the subject name bound to the server's public key by the X.509 certificate issued by the CA. For generality, we do not assume any relationship between the URL and subject name of a service, although in practice the subject name might contain the domain part of the URL.

The logical predicate isService parses a service element to extract the `<To>` field as uri, the `<Action>` field as ac, and the `<Subject>` field as subj:

predicate isService(S:**item**,uri:**item**,ac:**item**,subj:**string**) :−
  S = `<Service><To>`uri`</>` `<Action>`ac`</>` `<Subject>`subj`</></>`.

We also define predicates to parse X.509 certificates and to embed them in SOAP headers:

predicate isX509Cert (xcert:**bytes**,kr:**bytes**,u:**string**,a:**string**,k:**bytes**) :−
  checkx509(xcert,kr) = kr,
  u = x509user(xcert),

```
k = x509key(xcert),
a = x509alg(xcert).
```

predicate isX509Token (tok:**item**,kr:**bytes**,u:**string**,a:**string**,k:**bytes**) :−
```
tok = <BinarySecurityToken ValueType="X509v3">base64(xcert)</>,
```
isX509Cert (xcert,kr,u,a,k).

The predicate isX509Cert takes a byte array xcert containing an X.509 certificate, checks that it has been issued by a certifying authority with public key kr, and extracts the user name u, its user public key k, and its signing algorithm a. In SOAP messages, certificates are carried in XML elements called security tokens. The predicate isX509Token checks that an XML token element contains a valid X.509 certificate and extracts the relevant fields.

3.5 Users and Username Security Tokens

In our system descriptions, we identify each user by a `<User>` element that contains their username and password. The predicate isUser takes such an element, U, and extracts its embedded username u and password pwd.

predicate isUser (U:**item**,u:**string**,pwd:**string**) :−
```
U = <User><Username>u</><Password>pwd</></>.
```

In SOAP messages, the username is represented by a UsernameToken that contains the `<Username>` u, a freshly generated nonce n, and a timestamp t. The predicate isUserTokenKey takes such a token tok and extracts u, n, t, and then uses a user U for u to compute a key from pwd, n, and t.

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

The first line parses U to extract the username u and password pwd. The second line parses tok to extract n and t, implicitly checking that the username u is the same. In TulaFale, lists of terms are written as $tm_1 \ldots tm_m @ tm$ with $m \geq 0$, where the terms tm_1, ..., tm_m are the first m members of the list, and the optional term tm is the rest of the list. Here, the wildcard @ _ of the `<UsernameToken>` element matches the entire list of attributes. The last line computes the key k by applying the cryptographic hash function psha1 to pwd, n, and t (converted to **bytes**). (This formula for k is a slight simplification of the actual key derivation algorithm used by WSE.) The concat function returns the concatenation of two byte arrays.

3.6 Polyadic Signatures

An XML digital signature consists of a list of references to the elements being signed, together with a *signature value* that binds hashes of these elements using some signing secret. The signature value can be computed using several different cryptographic algorithms; in our example protocol, we rely on hmacsha1 for user signatures and on rsasha1 for service signatures.

The following predicates describe how to construct (mkSigVal) and check (isSigVal) the signature value sv of an XML element si, signed using the algorithm a with a key k. Each of these predicates is defined by a couple of clauses, representing symmetric and asymmetric signature algorithms. When a predicate is defined by multiple clauses, they are interpreted as a disjunction; that is, the predicate is satisfied if any one of its clauses is satisfied.

predicate mkSigVal (sv:**bytes**,si:**item**,k:**bytes**,a:**string**) :−
  a = `"hmacsha1"`, sv = hmacsha1(k,c14n(si)).

predicate isSigVal (sv:**bytes**,si:**item**,k:**bytes**,a:**string**) :−
  a = `"hmacsha1"`, sv = hmacsha1(k,c14n(si)).

predicate mkSigVal (sv:**bytes**,si:**item**,k:**bytes**,a:**string**) :−
  a = `"rsasha1"`, sv = rsasha1(k, c14n(si)).

predicate isSigVal (sv:**bytes**,si:**item**,p:**bytes**,a:**string**) :−
  a = `"rsasha1"`, p = checkrsasha1(p,c14n(si),sv).

The first clause of mkSigVal takes an item si to be signed and a key k for the symmetric signing algorithm hmacsha1, and generates the signature value sv. The first clause of isSigVal reverses this computation, taking sv, si, k, and a = `"hmacsha1"` as input and checking that sv is a valid signature value of si under k. Since the algorithm is symmetric, the two clauses are identical. The second clause of mkSigVal computes the signature value using the asymmetric rsasha1 algorithm, and the corresponding clause of isSigVal checks this signature value. In contrast to the symmetric case, the two clauses rely on different computations.

A complete XML signature for a SOAP message contains both the signature value sv, as detailed above, and an explicit description of the message parts are used to generate si. Each signed item is represented by a `<Reference>` element.

The predicate mkRef takes an item t and generates a `<Reference>` element r by embedding a sha1 hash of t, with appropriate sort conversions. Conversely, the predicate isRef checks that r is a `<Reference>` for t.

predicate mkRef(t:**item**,r:**item**) :−
  r = `<Reference>`
     `<Other></> <Other></>`
     `<DigestValue>` base64(sha1(c14n(t))) `</> </>`.

predicate isRef(t:**item**,r:**item**) :−

```
r = <Reference>
        _ _
       <DigestValue> base64(sha1(c14n(t))) </> </>.
```

(The XML constructed by mkRef abstracts some of the detail that is included in actual signatures, but that tends not to vary in practice; in particular, we include `<Other>` elements instead of the standard `<Transforms>` and `<DigestMethod>` elements. On the other hand, the `<DigestValue>` element is the part that depends on the subject of the signature, and that is crucial for security, and we model this element in detail.)

More generally, the predicate mkRefs(ts,rs) constructs a list ts and from a list rs, such that their members are pairwise related by mkRef. Similarly, the predicate mkRefs(ts,rs) checks that two given lists are pairwise related by mkRef. We omit their definitions.

A `<SignedInfo>` element is constructed from `<Reference>` elements for every signed element. A `<Signature>` element consists of a `<SignedInfo>` element si and a `<SignatureValue>` element containing sv. Finally, the following predicates define how signatures are constructed and checked.

predicate mkSigInfo (si:**item**,a:**string**,ts:**item**) :−
　mkRefs(ts,rs),
　rs = `<list>`@ refs`</>`,
　si = `<SignedInfo>`
　　　　`<Other></>` `<SignatureMethod Algorithm=a>` `</>`
　　　　@ refs `</>`.

predicate isSigInfo (si:**item**,a:**string**,ts:**item**) :−
　si = `<SignedInfo>`
　　　　_ `<SignatureMethod Algorithm=a>` `</>`
　　　　@ refs`</>`,
　rs = `<list>`@ refs`</>`,
　isRefs(ts,rs).

predicate mkSignature (sig:**item**,a:**string**,k:**bytes**,ts:**item**) :−
　mkSigInfo(si,a,ts),
　mkSigVal(sv,si,k,a),
　sig = `<Signature>` si `<SignatureValue>` base64(sv) `</>` `</>`.

predicate isSignature (sig:**item**,a:**string**,k:**bytes**,ts:**item**) :−
　sig = `<Signature>` si `<SignatureValue>` base64(sv) `</>`@ _`</>`,
　isSigInfo(si,a,ts),
　isSigVal(sv,si,k,a).

The predicate mkSigInfo takes a list of items to be signed, embedded in a `<list>` element ts, and generates a list of references refs for them, embedded in a `<list>` element rs, which are then embedded into si. The predicate isSigInfo checks si has been correctly constructed from ts.

The predicate mkSignature constructs si using mkSigInfo, generates the signature value sv using mkSigVal, and puts them together in a `<Signature>` element called sig; isSignature checks that a signature sig has been correctly generated from a, k, and ts.

4 Modelling SOAP Envelopes for our protocol

Relying on the predicate definitions of Section 3, which reflect (parts of) the SOAP and WS-Security specifications but do not depend on the protocol, we now define custom "top-level" predicates to build and check Messages 1 and 2 of our example protocol.

4.1 Building and Checking Message 1

Our goal C1 is to reach agreement on the data

(U,S,id1,t1,b1)

where

    U=`<User><Username>`u`</><Password>`pwd`</></>`
    S=`<Service><To>`uri`</><Action>`ac`</><Subject>`subj`</></>`

after receiving and successfully checking Message 1. To achieve this, the message includes a username token for U, encrypted with the public key of S (that is, one whose certificate has the subject name subj), and also includes a signature token, signing (elements containing) uri, ac, id1, t1, b1, and the encrypted username token, signed with the key derived from the username token.

We begin with a predicate setting the structure of the first envelope:

```
predicate env1(msg1:item,uri:item,ac:item,id1:string,t1:string,
               eutok:item,sig1:item,b1:item) :−
  msg1 =
    <Envelope>
      <Header>
        <To>uri</>
        <Action>ac</>
        <MessageId>id1</>
        <Security>
          <Timestamp><Created>t1</></>
          eutok
          sig1</></>
      <Body>b1</></>.
```

On the client side, we use a predicate mkMsg1 to construct msg1 as an output, given its other parameters as inputs:

predicate mkMsg1(msg1:**item**,U:**item**,S:**item**,kr:**bytes**,cert:**bytes**,
                     n:**bytes**,id1:**string**,t1:**string**,b1:**item**) :−
  isService(S,uri,ac,subj),
  isX509Cert(cert,kr,subj,`"rsasha1"`,ek),
  isUserTokenKey(utok,U,n,t1,sk),
  mkEncryptedData(eutok,utok,ek),
  mkSignature(sig1,`"hmacsha1"`,sk,
    `<list>`
      `<Body>`b1`</>`
      `<To>`uri`</>`
      `<Action>`ac`</>`
      `<MessageId>`id1`</>`
      `<Created>`t1`</>`
      eutok`</>`),
  env1(msg1,uri,ac,id1,t1,eutok,sig1,b1).

On the server side, with server certificate cert, associated private key sx, and expected user U, we use a predicate isMsg1 to check the input msg1 and produce S, id1, t1, and b1 as outputs:

predicate isMsg1(msg1:**item**,U:**item**,sx:**bytes**,cert:**bytes**,S:**item**,
                id1:**string**,t1:**string**,b1:**item**) :−
  env1(msg1,uri,ac,id1,t1,eutok,sig1,b1),
  isService(S,uri,ac,subj),
  isEncryptedData(eutok,utok,sx),
  isUserTokenKey(utok,U,n,t1,sk),
  isSignature(sig1,`"hmacsha1"`,sk,
    `<list>`
      `<Body>`b1`</>`
      `<To>`uri`</>`
      `<Action>`ac`</>`
      `<MessageId>`id1`</>`
      `<Created>`t1`</>`
      eutok`</>`).

4.2 Building and Checking Message 2

Our goal C2 is to reach agreement on the data

(U,S,id1,t1,b1,id2,t2,b2)

where

U=`<User><Username>`u`</><Password>`pwd`</></>`
S=`<Service><To>`uri`</><Action>`ac`</><Subject>`subj`</></>`

after successful receipt of Message 2, having already agreed on

(U,S,id1,t1,b1)

after receipt of Message 1.

A simple implementation is to make sure that the client's choice of id1 in Message 1 is fresh and unpredictable, to include `<relatesTo>`id1`</>` in Message 2, and to embed this element in the signature to achieve correlation with the data sent in Message 1. In more detail, Message 2 includes a certificate for S (that is, one with subject name subj) and a signature token, signing (elements containing) id1, id2, t2, and b2 and signed using the private key associated with S's certificate. The structure of the second envelope is defined as follows:

predicate env2(msg2:**item**,uri:**item**,id1:**string**,id2:**string**,
               t2:**string**,cert:**bytes**,sig2:**item**,b2:**item**) :−
  msg2 =
    `<Envelope>`
      `<Header>`
        `<From>`uri`</>`
        `<RelatesTo>`id1`</>`
        `<MessageId>`id2`</>`
        `<Security>`
          `<Timestamp><Created>`t2`</></>`
          `<BinarySecurityToken>`base64(cert)`</>`
          sig2`</></>`
      `<Body>`b2`</></>`.

A server uses the predicate mkMsg2 to construct msg2 as an output, given its other parameters as inputs (including the signing key):

predicate mkMsg2(msg2:**item**,sx:**bytes**,cert:**bytes**,S:**item**,
               id1:**string**,id2:**string**,t2:**string**,b2:**item**):−
  isService(S,uri,ac,subj),
  mkSignature(sig2,`"rsasha1"`,sx,
    `<list>`
      `<Body>`b2`</>`
      `<RelatesTo>`id1`</>`
      `<MessageId>`id2`</>`
      `<Created>`t2`</></>`),
  env2(msg2,uri,id1,id2,t2,cert,sig2,b2).

A client, given the CA's public key kr, and awaiting a response from S to a message with unique identifier id1, uses the predicate isMsg2 to check its input msg2, and produce data id2, t2, and b2 as outputs.

predicate isMsg2(msg2:**item**,S:**item**,kr:**bytes**,
               id1:**string**,id2:**string**,t2:**string**,b2:**item**) :−
  env2(msg2,uri,id1,id2,t2,cert,sig2,b2),
  isService(S,uri,ac,subj),
  isX509Cert(cert,kr,subj,`"rsasha1"`,k),
  isSignature(sig2,`"rsasha1"`,k,

18

```
<list>
  <Body>b2</>
  <RelatesTo>id1</>
  <MessageId>id2</>
  <Created>t2</></>).
```

5   Processes and Assertions in TulaFale

A TulaFale script defines a system to be a collection of concurrent processes
that may compute internally, using terms and predicates, and may also commu-
nicate by exchanging terms on named channels. The top-level process defined
by a TulaFale script represents the behaviour of the principals making up the
system—some clients and servers in our example. The attacker is modelled as an
arbitrary process running alongside the system defined by the script, interacting
with it via the public channels. The style of modelling cryptographic protocols,
with an explicit given process representing the system and an implicit arbitrary
process representing the attacker, originates with the spi calculus [AG99]. We
refer to the principals coded explicitly as processes in the script as being *compli-
ant*, in the sense they are constrained to follow the protocol being modelled, as
opposed to the non-compliant principals represented implicitly by the attacker
process, who are not so constrained.

A TulaFale script consists of a sequence of declarations. We have seen already
in Sections 3 and 4 many examples of Prolog-style declarations of clauses defining
named predicates. This section describe three further kinds of declaration—
for channels, correspondence assertions, and processes. Section 6 illustrate their
usage in a script that models the system of Section 2.

We describe TulaFale syntax in terms of several metavariables or nonter-
minals: *sort*, *term*, and *form* range over the sorts, algebraic terms, and logical
formulas, respectively, as introduced in Section 3; and *ide* ranges over alphanu-
meric identifiers, used to name variables, predicates, channels, processes, and
correspondence assertions.

A declaration **channel** $ide(sort_1, \ldots, sort_n)$ introduces a channel, named *ide*,
for exchanging n-tuples of terms with sorts $sort_1, \ldots, sort_n$. As in the asyn-
chronous pi calculus, channels are named, unordered queues of messages. By
default, each channel is public, that is, the attacker may input or output mes-
sages on the channel. The declaration may be preceded by the **private** keyword
to prevent the attacker accessing the channel. Typically, SOAP channels are
public, but channels used to represent shared secrets, such as passwords, are
private.

In TulaFale, as in some forms of the spi calculus, we embed correspondence
assertions in our process language in order to state certain security properties
enjoyed by compliant principals.

A declaration **correspondence** $ide(sort_1, \ldots, sort_n)$ introduces a label, *ide*,
for events represented by n-tuples of terms with sorts $sort_1, \ldots, sort_n$. Each
event is either a begin-event or an end-event; typically, a begin-event records an

```

initiation of a session, and an end-event records the satisfactory completion of a session, from the compliant principals' viewpoint. The process language includes constructs for logging begin- and end-events. The attacker cannot observe or generate events. We use correspondences to formalize the properties (C1) and (C2) of Section 2. The declaration of a correspondence on *ide* specifies a security assertion: that in any run of the explicit system composed with an arbitrary implicit attacker, every end-event labelled *ide* logged by the system corresponds to a previous begin-event logged by the system, also labelled *ide*, and with the same tuple of data. We name this property *robust safety*, following Gordon and Jeffrey [GJ03]. The implication of robust safety is that two compliant processes have reached agreement on the data, which typically include the contents of a sequence of one or more messages.

A declaration **process** $ide(ide_1{:}sort_1, \ldots, ide_n{:}sort_n) = proc$ defines a parametric process, with body the process *proc*, named *ide*, whose parameters $ide_1$, $\ldots$, $ide_n$ have sorts $sort_1, \ldots, sort_n$, respectively.

Next, we describe the various kinds of TulaFale process.

- A process **out** $ide(tm_1, \ldots, tm_n); proc$ sends the tuple $(tm_1, \ldots, tm_n)$ on the *ide* channel, then runs *proc*.
- A process **in** $ide(ide_1, \ldots, ide_n); proc$ blocks awaiting a tuple $(tm_1, \ldots, tm_n)$ on the *ide* channel; if one arrives, the process behaves as *proc*, with its parameters $ide_1, \ldots, ide_n$ bound to $tm_1, \ldots, tm_n$, respectively.
- A process **new** *ide*:**bytes**; *proc* binds the variable *ide* to a fresh byte array, to model cryptographic key or nonce generation, for instance, then runs *proc*. Similarly, a process **new** *ide*:**string**; *proc* binds the variable *ide* to a fresh string, to model password generation, for instance, then runs as *proc*.
- A process $proc_1|proc_2$ is a parallel composition of subprocesses $proc_1$ and $proc_2$; they run in parallel, and may communicate on shared channels.
- A process **!***proc* is a parallel composition of an unbounded array of replicas of the process *proc*.
- The process **0** does nothing.
- A process **let** $ide = tm; proc$ binds the term *tm* to the variable *ide*, then runs *proc*.
- A process **filter** $form \rightarrow ide_1, \ldots, ide_n; proc$ binds terms $tm_1, \ldots, tm_n$ to the variables $ide_1, \ldots, ide_n$ such that the formula *form* holds, then runs *proc*. (The terms $tm_1, \ldots, tm_n$ are computed by pattern-matching, as described in a previous paper [BFG04a].)
- A process $ide(tm_1, \ldots, tm_n)$, where *ide* corresponds to a declaration **process** $ide(ide_1{:}sort_1, \ldots, ide_n{:}sort_n) = proc$ binds the terms $tm_1, \ldots, tm_n$ to the variables $ide_1, \ldots, ide_n$, then runs *proc*.
- A process **begin** $ide(tm_1, \ldots, tm_n); proc$ logs a begin-event labelled with *ide* and the tuple $(tm_1, \ldots, tm_n)$, then runs *proc*.
- A process **end** $ide(tm_1, \ldots, tm_n); proc$ logs an end-event labelled with *ide* and the tuple $(tm_1, \ldots, tm_n)$, then runs *proc*.
- Finally, the process **done** logs a done-event. (We typically mark the successful completion of the whole protocol with **done**. Checking for the reachability

of the done-event is then a basic check of the functionality of the protocol, that it may run to completion.)

The main goal of the TulaFale tool is to prove or refute robust safety for all the correspondences declared in a script. Robust safety may be proved by a range of techniques; the first paper on TulaFale [BFG04a] uses manually developed proofs of behavioural equivalence. Instead, our TulaFale tool translates scripts into the applied pi calculus, and then runs Blanchet's resolution-based protocol verifier; the translation is essentially the same as originally described [BFG04a]. ProVerif (hence TulaFale) can also check secrecy assertions, but we omit the details here. In addition, TulaFale includes a sort-checker and a simulator, both of which help catch basic errors during the development of scripts. For example, partly relying on the translation, TulaFale can show the reachability of **done** processes, which is useful for verifying that protocols may actually run to completion.

# 6 Modelling and Verifying our Protocol

Relying on the predicates given in Section 4, we now define the processes modelling our sample system.

## 6.1 Modelling the System with TulaFale Processes

In our example, a public channel publish gives the attacker access to the certificates and public keys of the CA and two servers, named "BobsPetShop" and "ChasMarket". Another channel soap is for exchanging all SOAP messages; it is public to allow the attacker to read and write any SOAP message.

```
channel publish(item).
channel soap(item).
```

The following is the top-level process, representing the behaviour of all compliant principals.

```
new sr:bytes; let kr = pk(sr);
new sx1:bytes; let cert1 = x509(sr,"BobsPetShop","rsasha1",pk(sx1));
new sx2:bytes; let cert2 = x509(sr,"ChasMarket","rsasha1",pk(sx2));
out publish(base64(kr));
out publish(base64(cert1));
out publish(base64(cert2));
( !MkUser(kr) |!MkService(sx1,cert1) |!MkService(sx2,cert2) |
   (!in anyUser(U); Client(kr,U)) |
   (!in anyService(sx,cert,S); Server(sx,cert,S)) )
```

The process begins by generating the private and public keys, sr and kr, of the CA. It generates the private keys, sx1 and sx2, of the two servers, plus their certificates, cert1 and cert2. Then it outputs the public data kr, cert1, cert2 to the attacker. After this initialization, the system behaves as the following parallel composition of five processes.

```
!MkUser(kr) |!MkService(sx1,cert1) |!MkService(sx2,cert2) |
(!in anyUser(U); Client(kr,U)) |
(!in anyService(sx,cert,S); Server(sx,cert,S)
```

As explained earlier, the **!** symbol represents unbounded replication; each of
these processes may execute arbitrarily many times. The first process allows the
opponent to generate fresh username/password combinations that are shared
between all clients and servers. The second and third allow the opponent to
generate fresh services with subject names `"BobsPetShop"` and `"ChasMarket"`,
respectively. The fourth acts as an arbitrary client U, and the fifth acts as an
arbitrary service S.

The process MkUser inputs the name of a user from the environment on
channel genUser, then creates a new password and records the username/pass-
word combination U as a message on private channel anyUser, representing the
database of authorized users.

```
channel genUser(string).
private channel anyUser(item).
process MkUser(kr:bytes) =
    in genUser(u);
    new pwd:string;
    let U = <User><Username>u</><Password>pwd</></>;
    !out anyUser (U).
```

The process MkService allows the attacker to create services with the subject
name of the certificate cert.

```
predicate isServiceData(S:item,sx:bytes,cert:bytes) :−
    isService(S,uri,ac,x509user(cert)),
    pk(sx) = x509key(cert).
```

```
channel genService(item).
private channel anyService(bytes,bytes,item).
process MkService(sx:bytes,cert:bytes) =
    in genService(S);
    filter isServiceData(S,sx,cert) →;
    !out anyService(sx,cert,S).
```

Finally, we present the processes representing clients and servers. Our desired
authentication and correlation properties are correspondence assertions embed-
ded within these processes. We declare the sorts of data to be agreed by the
clients and servers as follows.

```
correspondence C1(item,item,string,string,item).
correspondence C2(item,item,string,string,item,string,string,item).
```

The process Client(kr:bytes,U:item) acts as a client for the user U, assuming
that kr is the CA's public key.

```
channel init(item,bytes,bytes,string,item).
```

```
process Client(kr:bytes,U:item) =
  in init (S,certA,n,t1,b1);
  new id1:string;
  begin C1 (U,S,id1,t1,b1);
  filter mkMsg1(msg1,U,S,kr,certA,n,id1,t1,b1) →msg1;

  out soap(msg1);
  in soap(msg2);
  filter isMsg2(msg2,S,kr,id1,id2,t2,b2) →id2,t2,b2;
  end C2 (U,S,id1,t1,b1,id2,t2,b2);
  done.
```

The process generates a globally fresh, unpredictable identifier id1 for Message 1, to allow correlation of Message 2 as discussed above. It then allows the attacker to control the rest of the data to be sent by receiving it off the public init channel. Next, the TulaFale **filter** operator evaluates the predicate mkMsg1 to bind variable msg1 to Message 1. At this point, the client marks its intention to communicate data to the server by logging an end-event, labelled C1, and then outputs the message msg1. The process then awaits a reply, msg2, checks the reply with the predicate isMsg2, and if this check succeeds, ends the C2 correspondence. Finally, to mark the end of a run of the protocol, it becomes the **done** process—an inactive process, that does nothing, but whose reachability can be checked, as a basic test of the protocol description.

The process Server(sx:**bytes**,cert:**bytes**,S:**item**) represents a service S, with private key sx, and certificate cert.

```
channel accept(string,string,item).
process Server(sx:bytes,cert:bytes,S:item) =
  in soap(msg1);
  in anyUser(U);
  filter isMsg1(msg1,U,sx,cert,S,id1,t1,b1) →id1,t1,b1;
  end C1 (U,S,id1,t1,b1);

  in accept (id2,t2,b2);
  filter mkMsg2(msg2,sx,cert,S,id1,id2,t2,b2) →msg2;
  begin C2 (U,S,id1,t1,b1,id2,t2,b2);
  out soap(msg2).
```

The process begins by selecting a SOAP message msg1 and a user U off the public soap and the private anyUser channels, respectively. It filters this data with the isMsg1 predicate, which checks that msg1 is from U, and binds the variables S, id1, t1, and b1. At this point, it asserts an end-event, labelled C1, to signify apparent agreement on this data with a client. Next, the process inputs data from the opponent on channel accept, to determine the response message. The server process completes by using the predicate mkMsg2 to construct the response msg2, asserting a begin-event for the C2 correspondence, and finally sending the message.

6.2 Analysis

The TulaFale script for this example protocol consists of 200 lines specific to the protocol, and 200 lines of library predicates. (We have embedded essentially the whole script in this paper.) Given the applied pi calculus translation of this script, ProVerif shows that our two correspondences C1 and C2 are robustly safe. Failure of robust safety for C1 or C2 would reveal that the server or the client has failed to authenticate Message 1, or to authenticate Message 2 and correlate it with Message 1, respectively. Processing is swift enough—around 25s on a 2.4GHz 1GB P4—to support easy experimentation with variations in the protocol, specification, and threat model.

7 Conclusions

TulaFale is a high-level language based on XML with symbolic cryptography, clausally-defined predicates, pi calculus processes, and correspondence assertions. Previous work [BFG04a] introduces a preliminary version of TulaFale, defines its semantics via translation into the applied pi calculus [AF01], illustrates TulaFale via several single-message protocols, and describes hand-crafted correctness proofs.

The original reasons for choosing to model WS-Security protocols using the the pi calculus, rather than some other formal method, include the generality of the threat model (the attacker is an unbounded, arbitrary process), the ease of simulating executable specifications written in the pi calculus, and the existence of a sophisticated range of techniques for reasoning about cryptographic protocols.

Blanchet's ProVerif [Bla01,Bla02] turns out to be a further reason for using the pi calculus to study SOAP security. Our TulaFale tool directly implements the translation into the applied pi calculus and then invokes Blanchet's verifier, to obtain fully automatic checking of SOAP security protocols. This checking shows no attacker expressible as a formal process can violate particular SOAP-level authentication or secrecy properties. Hence, we expect TulaFale will be useful for describing and checking security aspects of web services specifications. We have several times been surprised by vulnerabilities discovered by the TulaFale tool and the underlying verifier. Of course, every validation method, formal or informal, abstracts some details of the underlying implementation, so checking with TulaFale only partially rules out attacks on actual implementations. Still, ongoing work is exploring how to detect vulnerabilities in web services deployments, by extracting TulaFale scripts from XML-based configuration data [BFG04b].

The request/response protocol presented here is comparable to the abstract RPC protocols proposed in earlier work on securing web services [GP03], but here we accurately model the SOAP and WS-Security syntax used on the wire. Compared with the SOAP-based protocols in the article introducing the TulaFale semantics [BFG04a], the novelties are the need for the client to correlate the request with the reply, and the use of encryption to protect weak user passwords

from dictionary attacks. In future work, we intend to analyse more complex SOAP protocols, such as WS-SecureConversation [KN⁺04], for securing client-server sessions.

Although some other process calculi manipulate XML [BS03,GM03], they are not intended for security applications. We are beginning to see formal methods applied to web services specifications, such as the TLA+ specification [JLLV04] of the Web Services Atomic Transaction protocol, checked with the TLC model checker. Still, we are aware of no other security tool for web services able to analyze protocol descriptions for vulnerabilities to XML rewriting attacks.

References

AF01.   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.

AG99.   M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

BEK⁺00.   D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*, 2000. W3C Note, at `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

BFG04a.   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.

BFG04b.   K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. Submitted for publicaton, 2004.

Bla01.   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.

Bla02.   B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359. Springer, 2002.

Box01.   D. Box. A brief history of SOAP. At `http://webservices.xml.com/pub/a/ws/2001/04/04/soap.html`, 2001.

BS03.   G. Bierman and P. Sewell. Iota: a concurrent XML scripting language with application to Home Area Networks. Technical Report 557, University of Cambridge Computer Laboratory, 2003.

DHK⁺04.   M. Davis, B. Hartman, C. Kaler, A. Nadalin, and J. Schwarz. *WS–I Security Scenarios*, February 2004. Working Group Draft Version 0.15, at `http://www.ws-i.org/Profiles/BasicSecurity/2004-02/SecurityScenarios-0.15-WGD.pdf`.

DY83.   D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

GJ03. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

GM03. P. Gardner and S. Maffeis. Modeling dynamic web data. In *DBPL'03*, LNCS. Springer, 2003.

GP03. 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.

HL03. M. Howard and D. LeBlanc. *Writing secure code*. Microsoft Press, second edition, 2003.

IM02. IBM Corporation and Microsoft Corporation. Security in a web services world: A proposed architecture and roadmap. At http://msdn.microsoft.com/library/en-us/dnwssecur/html/securitywhitepaper.asp, April 2002.

JLLV04. 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.

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

Low97. 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.

Mic02. Microsoft Corporation. *Web Services Enhancements for Microsoft .NET*, December 2002. At http://msdn.microsoft.com/webservices/building/wse/default.aspx.

Mil99. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

NKHBM04. A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, March 2004. At http://www.oasis-open.org/committees/download.php/5941/oasis-200401-wss-soap-message-security-1.0.pdf.

NS78. R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

S⁺04. J. Schlimmer et al. *A Proposal for UPnP 2.0 Device Architecture*, May 2004. At http://msdn.microsoft.com/library/en-us/dnglobspec/html/devprof.asp.

SMWC03. J. Shewchuk, S. Millet, H. Wilson, and D. Cole. Expanding the communications capabilities of web services with WS-Addressing. At http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/soapmail.asp, 2003.

SS02. J. Scambay and M. Shema. *Hacking Web Applications Exposed*. McGraw-Hill/Osborne, 2002.

Vog03. W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.

W3C03. W3C. *SOAP Version 1.2*, 2003. W3C Recommendation, at http://www.w3.org/TR/soap12.

Win98. D. Winer. RPC over HTTP via XML. At http://davenet.scripting.com/1998/02/27/rpcOverHttpViaXml, 1998.

Win99.   D. Winer. Dave's history of SOAP. At `http://www.xmlrpc.com/discuss/msgReader$555`, 1999.

WL93.   T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.