A Semantics for Web Services Authentication

Karthikeyan Bhargavan
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Abstract

We consider the problem of specifying and verifying cryptographic security protocols for XML web services. The security specification WS-Security describes a range of XML security tokens, such as username tokens, public-key certificates, and digital signature blocks, amounting to a flexible vocabulary for expressing protocols. To describe the syntax of these tokens, we extend the usual XML data model with symbolic representations of cryptographic values. We use predicates on this data model to describe the semantics of security tokens and of sample protocols distributed with the Microsoft WSE implementation of WS-Security. By embedding our data model within Abadi and Fournet's applied pi calculus, we formulate and prove security properties with respect to the standard Dolev-Yao threat model. Moreover, we informally discuss issues not addressed by the formal model. To the best of our knowledge, this is the first approach to the specification and verification of security protocols based on a faithful account of the XML wire format.

Categories and Subject Descriptors: F.3.2 [Theory of Computation]: Logics and meanings of programs—*Semantics of Programming Languages*

General Terms: Security, Languages, Theory, Verification

Keywords: Web Services, Applied Pi Calculus, XML Security

1 Motivations and Outline

Over the past few years, a growing list of specifications has been defining aspects of XML web services. Security is a serious concern and is addressed, in particular, by the recent WS-Security specification [3]. This specification provides an XML vocabulary for designing cryptographic protocols, is widely implemented, and is undergoing standardization at OASIS [28]. Still, it provides no formal basis for stating security goals or reasoning about correctness. The trouble is, new cryptographic protocols are often flawed, XML or no XML.

Meanwhile, there has been a sustained and successful effort to develop formalisms for expressing and verifying cryptographic protocols ([5, 9, 10, 19, 22, 24, 31, 35] etc). Formal methods can verify various security properties against a standard threat model based on an opponent able to monitor and manipulate messages sent over the network. Such verifications are typically of abstract versions of protocols, expressed using fixed, high-level, ad-hoc message formats, rather than the ordered-tree-with-pointers syntax of XML.

This paper brings these developments together. We introduce a language-based model for XML security protocols, and establish process calculus techniques for verifying authentication properties for a wide class of WS-Security protocols.

Background: Web Services Security. Web services are a distributed systems technology based on network endpoints exchanging SOAP [6] envelopes—XML documents with a mandatory `Body` element containing a request, response, or fault element, together with an optional `Header` element containing routing or security information. SOAP allows for network intermediaries—such as routers or firewalls—to process an envelope, by adding or modifying headers. Examples of web services include Internet-based services for ordering books or invoking search engines, and intranet-based services for linking enterprise applications.

A common technique for securing SOAP exchanges is to rely on a lower-level secure tunnel between the endpoints, typically an SSL connection. This works well in many situations, but has the usual disadvantages of transport-level security: the secrecy or integrity of messages can be guaranteed while in the tunnel, but not subsequently in files or databases, and they may not match the security requirements of the application. Pragmatically, client authentication is often performed by the application rather than by SSL. Besides, SSL does not fit SOAP's message-based architecture: intermediaries cannot see the contents of the tunnel, and so cannot route or filter messages.

To better support end-to-end security [32], WS-Security defines how to sign or encrypt SOAP messages without relying on a secure transport. A central—but informal—abstraction is the *security token*, which covers data making security claims, such as user identifiers, cryptographic keys, or certificates. WS-Security provides a precise syntax for multiple token formats, such as XML username tokens and XML-encoded binary tokens conveying X.509 public-key certificates or symmetric keys. It also specifies syntax for applying encryption and signature to selected elements of SOAP messages. Like many traditional protocol specifications, WS-Security details message formats—such as the names of XML tags—rather than security goals and their enforcement, thereby focusing on interoperability rather than security. Although it gives a syntax for a broad range of protocols, WS-Security also emphasizes flexibility, and does not define any particular protocol. As a result, for each given WS-Security compliant protocol, security goals still have to be carefully specified and validated.

Background: Security Protocol Verification. This paper addresses authentication properties of XML-based security protocols against a standard threat model: an opponent able to read, replay,

POPL'04 January 14–16, 2004, Venice, Italy.

redirect, and transform messages, but who cannot simply guess secrets. Needham and Schroeder describe such an opponent in their pioneering work on cryptographic protocols [29]. The first formalization was by Dolev and Yao [15]. A great many formal methods have been deployed to verify protocols against this threat model, with particular progress in the past few years.

This paper uses Abadi and Fournet's applied pi calculus [1, 18] as the underlying specification language for protocols, and relies on proof techniques from concurrency theory. In this approach, the opponent is simply an arbitrary context within the calculus; the scoping rules of the pi calculus ensure that the opponent cannot learn names representing secrets such as passwords.

We formalize authentication properties using standard correspondence assertions [37], as embedded within the pi calculus by Gordon and Jeffrey [19]. These assertions are based on two kinds of events, which can be thought of as logfile entries by protocol participants. A begin-event marks the initiation of a run of a protocol, while an end-event marks the commitment that a run has correctly completed. Event labels include data identifying the run, such as the names of the client and server, message identifier, and payload. A protocol is safe if in every run, every end-event corresponds to a previous begin-event with the same label. Moreover, a protocol is robustly safe if it is safe in the presence of an arbitrary opponent process. Robust safety establishes message authentication, and rules out a range of attacks.

This Paper. We tackle the problem of formal reasoning about XML-encoded cryptographic protocols. The interest and novelty in this problem arises not from the XML syntax itself, but from the need to model low-level detail, in particular, the flexibility and hierarchical structure of XML signatures [17], designed to tolerate changes to the headers of a SOAP message over its lifetime.

We base our approach on three formalisms: a symbolic syntax for XML with cryptography and a predicate language for defining acceptable messages—both defined in Section 2—and a specialized version of the applied pi calculus. We explain the purpose of each of these in turn.

First, to represent XML messages with embedded cryptography, we enrich the standard XML data model with an abstract syntax for embedded byte arrays and cryptographic functions. Formally, we define a many-sorted algebra with sorts for the usual constructs of XML—strings, attributes, and so on—plus a new sort of symbolic byte arrays, in the style of Dolev and Yao, to represent cryptographic materials embedded in XML.

Second, to define the semantics of security tokens and validity conditions on messages, we introduce a Prolog-like language of predicates on XML data. By insisting on fidelity to the low-level XML format, we are confronted with the difficulty of defining rather intricate conditions of message acceptability, and hence we need some language of predicates on XML. It may be possible to extend some standard type system or query language for XML (such as DTDs, XML Schema, or XPath) to express conditions on cryptographic values. Instead, for the sake of simplicity and being self-contained, we rely on a fairly standard Horn-clause logic.

Third, to describe complete protocols, we embed these messages and predicates within the applied pi calculus. We state and prove protocol properties against a large class of contexts representing attackers. Applied pi is parameterized in general by an arbitrary equational theory of terms, which we specialize to our data model for XML with cryptography. We obtain a calculus expressive enough to implement our predicates, and to describe complex protocol configurations.

In Section 3, given these notations, we formalize the security goals and message formats of a series of sample protocols. These protocols illustrate a range of WS-Security concepts including message identifiers, password digests, username tokens, X.509 public-key certificates, XML signatures based on both password-derived keys and certificates, and processing by SOAP intermediaries as well as end-points. For each protocol, we give predicates describing acceptable messages, and state authentication goals using the usual message-sequence notation. WS-Security itself defines a formal syntax for messages, but gives only an informal account of the security checks performed by compliant implementations, as each token is processed in the SOAP protocol stack. Through formalizing the series of protocols, we accumulate a collection of re-usable predicates reflecting the semantics of these tokens. Hence, we obtain a first formal semantics for a significant fragment of WS-Security.

In Section 4, we formalize the message-sequence notation within the applied pi calculus so as to verify our authentication goals. We explain the structure of the proof of one of the sample protocols from Section 3. The proofs are compositional, and rely on identifying a "trusted computing base" embodying the essential checks underlying the protocol.

In Section 5, we conclude, and discuss related and future work.

A technical report [4] provides details omitted in this version of the paper, including sample SOAP messages obtained experimentally for the protocols of Section 3, a brief overview of the applied pi calculus calculus, and detailed proof of all our protocols.

Contributions. In summary, we make three main contributions:

(1) A new data model and predicate language for describing XML-level cryptographic protocols.

Fidelity to the detailed messaging format enables us to address its subtleties, such as the interpretation of compound signatures.

(2) A collection of predicates defining a semantics for the security tokens of WS-Security and related specifications.

We cover only a substantial fragment of WS-Security, but our semantics does establish the feasibility of applying our formal developments to a large class of protocols.

(3) Proofs for a series of concrete protocols drawn from the WSE distribution.

At an abstract level, the protocols we consider are quite simple. Still, we have encountered vulnerabilities to XML rewriting attacks in implementations of these conceptually simple protocols. So it is worth establishing correctness at this level, and indeed the formal Dolev-Yao properties are non-trivial.

2 Symbolic Cryptography in XML

The core of our data model—or abstract syntax—for XML trees is adapted from Siméon and Wadler's grammar for XML [34].

XML Data Model: Standard Core

Tag	::= `anyLegalXmlName`	XML name
str : string	::= `any legal XML string`	XML string
a : att	::= *Tag*`="`*str*`"`	attribute
as : atts	::= $a\ as \mid \varepsilon$	attribute sequence
i : item	::= $Elem \mid str$	item
is : items	::= $i\ is \mid \varepsilon$	item sequence
Elem	::= <*Tag as*>*is*</*Tag*>	element

Our data model represents valid, parsed XML. It resembles the XML infoset recommendation [11] but with some differences. For the sake of clarity, we completely suppress information about XML namespaces, and the document `<?xml ...>` directive. As a minor technical convenience, we model an element's attributes as an ordered sequence rather than a set. (This also reflects the capability of an attacker to observe ordering information.)

Our syntax is intentionally close to the standard XML wire format, but for brevity we rely on three notational conventions. First, although formally an element's attributes *as* and body *is* are recursively defined lists, we typically use a standard tuple notation for fixed-length sequences. Second, instead of writing an element `<Envelope></Envelope>`, say, in full, we drop the tag from the closing bracket, and simply write `<Envelope></>`. Third, when writing an element that spans several lines, we rely on indentation (as in Haskell or Python) to delimit the body, and so omit the closing bracket.

Conventions for Sequences and Elements:

$a_1 \ldots a_n \triangleq a_1 \ (\ldots \ (a_n \ \varepsilon))$: atts for $n \geq 0$; similarly for items.

$<Tag \ as>is</> \triangleq <Tag \ as>is</Tag>$

$$\left.\begin{array}{l} <Tag \ as> \\ \quad i_1 \\ \quad \vdots \\ \quad i_n \end{array}\right\} \triangleq <Tag \ as>i_1 \cdots i_n</>$$

Formally, our data model is a many-sorted algebra, based on the sorts string, att, atts, item, items, plus a sort bytes for binary data. We embed this algebra within the applied pi calculus as described in Section 4. The complete algebra is given by the "XML Data Model" table above plus two more below.

We need the following general definitions. We let T, U, V range over terms of arbitrary sort in the algebra, and write T : string, for example, to mean that T belongs to the sort string. Throughout we assume that terms, predicates, and equations are well-sorted, but for the sake of brevity keep the details implicit. In addition to the syntax defined in this section, terms include sorted variables, $x, y, z, \ldots$. We let $fv(T)$ be the set of variables occurring in a term T. We say a term T is *closed* if and only if $fv(T) = \varnothing$. Otherwise, we say the term is *open*—an open term represents a closed term with some undetermined subterms, represented by the variables. We let σ range over parallel substitutions $\{\widetilde{x} = \widetilde{V}\}$ of the terms $\widetilde{V}$ for the variables $\widetilde{x}$, and we define $dom(\{\widetilde{x} = \widetilde{V}\}) \triangleq \{\widetilde{x}\}$. We say that a substitution σ is *closed* if and only if $\sigma(x)$ is a closed term for each $x \in dom(\sigma)$. We let $\widetilde{V}$ range over vectors $V_1, \ldots, V_n$ of terms, and similarly $\widetilde{x}$ ranges over vectors $x_1, \ldots, x_n$ of variables. We often treat such vectors as sets.

Next, we extend the standard data model with a symbolic representation of cryptography and related operations. We introduce a sort bytes representing byte arrays, and extend string with Base64-encoded arrays (base64(x)). We assume there is an infinite set of atomic, abstract *names*, ranged over by s. Each name is either of sort bytes or string. We use these names to represent arbitrary, unstructured cryptographic materials such as passwords and keys. We let $fn(T)$ be the set of names occurring in a term T.

XML Data Model: Byte Arrays, Symbolic Cryptography

x : bytes ::=	byte array (not itself XML)
s	abstract name (key, nonce)
concat(x_1, x_2 : bytes)	array concatenation
c14n(i : item)	canonical bytes of an item
utf8(str : string)	UTF8 representation of strings
sha1(x : bytes)	cryptographic digest
p-sha1(pwd : string, $salt$: bytes)	key from salted password
hmac-sha1(key, x : bytes)	keyed hash
pk(k_{priv} : bytes)	map from private to public key
rsa-sha1(x, k_{priv} : bytes)	public key signature
x509(s_r : bytes, u : string, a : string, k_{pub} : bytes)	X.509 certificate
str : string ::=	XML string
s	abstract name (password)
base64(x : bytes)	Base64-encoding of byte array
principal(pwd : string)	map from password to principal

The exact choice of primitives is a little arbitrary; we include enough operations here for the protocols of Section 3. The term concat(x_1, x_2) represents the concatenation of the two arrays x_1 and x_2. The term c14n(i) represents the array obtained by canonicalizing the XML represented by i, according to some standard algorithm [7, 8]. (In fact, for our purposes, c14n is simply a way of symbolically treating an XML item as a byte array; our c14n does not sort attribute lists, for example.) The term utf8(str) represents the UTF8 encoding of the XML string str. The term sha1(x) represents the one-way SHA1 hash of the x array. The term p-sha1(pwd, $salt$) represents a key obtained by hashing the pwd password and the $salt$ array [14]. The term hmac-sha1(key, x) represents a keyed hash of the x array using the key array as the key [23]. The term pk(k_{priv}) represents the public key associated with a private signing key k_{priv}. The term rsa-sha1(x, k_{priv}) is a public-key signature of x under the private key k_{priv} [21]. The term x509(s_r, u, a, k) represents a basic X.509 public-key certificate, where s_r is the private signing key of the certifier and u, a, k are the signed user name, algorithm, and key for a given principal. (Such binary certificates can be embedded as XML items; they are used here to carry public keys for the asymmetric signing algorithm rsa-sha1.) Finally, the term principal(pwd) is used to represent a database of user names associated with secrets, such as passwords, and is explained in Section 3.2.

Our threat model is that SOAP messages may be intercepted, decomposed, modified, assembled, and replayed by the attacker [15, 29]. The following selector functions and inverses symbolically represent the ability of the attacker to decompose messages. It is deliberate that there are no inverses for the three hash functions (sha1, p-sha1, and hmac-sha1), and for the public-key (pk) and user name (principal) maps; the attacker cannot reverse these one-way functions.

XML Data Model: Selectors and Inverses

x : bytes ::=	byte array
fst(x : bytes)	left part of concat
snd(x : bytes)	right part of concat
i-base64(str : string)	inverse of base64
x509-key($cert$: bytes)	public key in X.509 certificate
check-x509($cert, k_r$: bytes)	X.509 certificate verification
check-rsa-sha1(x, sig, k_{pub} : bytes)	public key verification
str : string ::=	XML string
Tag.parm(a : att)	string param of Tag-attribute
i-utf8(x : bytes)	inverse of utf8
x509-user($cert$: bytes)	name in X.509 certificate
x509-alg($cert$: bytes)	algorithm in X.509 certificate
a : att ::=	attribute
hd(as : atts)	head of a sequence
as : atts ::=	attributes
Tag.att(i : item)	attributes of Tag-element
tl(as : atts)	tail of a sequence
i : item ::=	item
hd(is : items)	head of a sequence
i-c14n(x : bytes)	inverse of c14n

is : items ::=	items
Tag.body(i : item)	body of Tag-element
tl(is : items)	tail of a sequence

Most of these selectors are straightforward inverses with single arguments. The two exceptions are check-x509 and check-rsa-sha1. The term check-x509$(cert, k_r)$ checks that the certificate $cert$ is signed with a private key associated with the public key k_r, and yields k_r if this succeeds. The term check-rsa-sha1(x, sig, k_{pub}) checks that sig is the rsa-sha1 signature of x under the private key corresponding to the public key k_{pub} and yields k_{pub} if this succeeds. (Some of the inverses, such as the functions fst and snd, would be impossible to implement in general, and we do not rely on them to program compliant principals; they exist to represent the possibility of the attacker correctly guessing, for example, how to divide an array obtained by concatenation into its original two halves.)

We represent evaluation of selectors and inverses by an equivalence, $U = V$, the least sort-respecting congruence induced by the following axioms.

Equivalence of Terms of the Data Model: $U = V$

$\mathsf{hd}(a\ as) = a$	$\mathsf{tl}(a\ as) = as$
$\mathsf{hd}(i\ is) = i$	$\mathsf{tl}(i\ is) = is$
$Tag.\mathsf{att}(\texttt{<}Tag\ as\texttt{>}is\texttt{</>}) = as$	$\mathsf{i\text{-}base64}(\mathsf{base64}(x)) = x$
$Tag.\mathsf{body}(\texttt{<}Tag\ as\texttt{>}is\texttt{</>}) = is$	$\mathsf{i\text{-}utf8}(\mathsf{utf8}(str)) = str$
$Tag.\mathsf{parm}(Tag\texttt{="}str\texttt{"}) = str$	$\mathsf{i\text{-}c14n}(\mathsf{c14n}(i)) = i$
$\mathsf{fst}(\mathsf{concat}(x_1, x_2)) = x_1$	$\mathsf{snd}(\mathsf{concat}(x_1, x_2)) = x_2$
$\mathsf{x509\text{-}user}(\mathsf{x509}(s_r, u, a, k)) = u$	$\mathsf{x509\text{-}alg}(\mathsf{x509}(s_r, u, a, k)) = a$
$\mathsf{x509\text{-}key}(\mathsf{x509}(s_r, u, a, k)) = k$	

$$\mathsf{check\text{-}x509}(\mathsf{x509}(s_r, u, a, k), \mathsf{pk}(s_r)) = \mathsf{pk}(s_r)$$
$$\mathsf{check\text{-}rsa\text{-}sha1}(x, \mathsf{rsa\text{-}sha1}(x, k_{priv}), \mathsf{pk}(k_{priv})) = \mathsf{pk}(k_{priv})$$

In the absence of additional equivalences between terms, we implicitly assume that our cryptographic operations have no additional interactions. For instance, the hash functions sha1, p-sha1, hmac-sha1, and rsa-sha1 are independent here. This can be informally checked from their cryptographic specifications [16, 14, 23, 21], or modelled as a refinement of the term equivalence, as in [1].

We end this section by defining a logical notation for predicates on XML terms. Formally, we present a Horn logic over our many-sorted algebra, with primitive formulas for equality and list membership, but no recursively-defined predicates. Our notation is simple, and suffices for reasoning about security; other languages feature more expressive pattern-matching for XML, but their semantics would be harder to formalize.

We assume there is a fixed, finite set of *predicates*, ranged over by p. For each predicate p, we assume there is a single definition $p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n$, where each Φ_i is a *formula*, and $n > 0$. (When $n > 1$, we usually present each clause $p(\widetilde{x}) :- \Phi_i$ separately, in the style of Prolog.) Next, we define the syntax of formulas.

Syntax of Formulas and Predicate Definitions:

Φ ::=	formula
$V = T$	term comparison
$U \in V$	list membership
$p(\widetilde{V})$	predicate instance
Φ_1, Φ_2	conjunction
$p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n$	definition of predicate p with $n > 0$

We assume that formulas are well-sorted according to the following rules: in $V = T$ both terms belong to the same sort; in $U \in V$ either U : item and V : items or U : att and V : atts; in $p(\widetilde{V})$ when $p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n$, the length and sorts of $\widetilde{V}$ match the length and sorts of $\widetilde{x}$.

Let p contribute to q if and only if an instance $p(\widetilde{V})$ occurs in one of the disjuncts defining q. We stipulate that this relation is well-founded, to avoid recursively-defined predicates. We let $fv(\Phi)$ be the free variables of all the terms occurring in Φ, and in particular, $fv(p(V_1, \ldots, V_n)) \triangleq fv(V_1) \cup \cdots \cup fv(V_n)$. In any clause $p(\widetilde{x}) :- \Phi$, we say that each $z \in fv(\Phi) \setminus \widetilde{x}$ is a *local variable*. By convention, each occurrence in a clause of the wildcard symbol _ is short for the only occurrence of a fresh local variable. Local variables are existentially quantified in our semantics; we identify clauses up to the consistent renaming of local variables.

Semantics of Formulas: $\models \Phi$ **where** $fv(\Phi) = \varnothing$

$$\models V = T \triangleq (V = T)$$
$$\models U \in V \triangleq (V = U_1 \ldots U_i\, U\, V') \quad \text{for some } U_1, \ldots, U_i, V', \text{ with } i \geq 0$$
$$\models p(\widetilde{V}) \triangleq\ \models \Phi_i\{\widetilde{x} = \widetilde{V}\}\{\widetilde{z} = \widetilde{U}\} \quad \text{for some } i \in 1..n \text{ and closed terms } \widetilde{U} \text{ where } p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n \text{ and } \widetilde{z} = fv(\Phi_i) \setminus \widetilde{x}$$
$$\models \Phi_1, \Phi_2 \triangleq\ \models \Phi_1 \text{ and } \models \Phi_2$$

A formula Φ is *valid* if we have $\models \Phi\sigma$ whenever $fv(\Phi\sigma) = \varnothing$.

For open formulas, we introduce a logical equivalence.

Logical Equivalence of Formulas:

Two formulas Φ, Φ' are *logically equivalent* when, for all substitutions σ such that $\Phi\sigma$ and $\Phi'\sigma$ are closed, we have $\models \Phi\sigma$ iff $\models \Phi'\sigma$.

3 Example Protocols

This section describes some WS-Security protocols, whose goal is to authenticate access to a basic web service. We first present a typical (unauthenticated) web service, then successively refine it by introducing password-based digests, signatures, X.509 certificates, and a firewall intermediary. The first four protocols are taken from the samples provided with WSE 1.0 [26]; we used the actual SOAP messages produced by this implementation to experimentally validate our model. (The technical report includes sample messages produced by WSE.)

3.1 An (Unauthenticated) Web Service

We consider a typical e-commerce website application where customers can browse and purchase items [25]. The orders are stored on a database server, and can be retrieved and viewed on later visits. For security, customers are required to login, with username and password, before placing and retrieving orders. In addition to the standard website interface, the server provides a SOAP web service *GetOrder* that a customer may invoke to retrieve their order in XML format, to save it as a receipt, for instance. Our aim is to provide the same level of security for this web service as the website login.

A call to *GetOrder* consists of a SOAP request and a SOAP response. We introduce predicates to describe these messages. As a first example, a valid SOAP message is an XML `Envelope`, containing a `Header` and a `Body`. The predicate $hasBody(e, b)$ below means b is the body of envelope e (the wildcard _ matches anything):

$$\begin{aligned}&hasBody(e : \mathsf{item}, b : \mathsf{item}) :- \\ &\quad e = \texttt{<Envelope><Header>\_</>}b\texttt{</>}, \\ &\quad b = \texttt{<Body \_>\_</>}.\end{aligned}$$

The SOAP request for *GetOrder* is an envelope, where the body encodes the parameters of the call. The resulting SOAP response has a body containing the order, in XML:

```
isGetOrder(b : item, OrderId : string) :-
  b = <Body _>
        <GetOrder>
          <orderId>OrderId</>

isGetOrderResponse(b : item, OrderId : string, u : string) :-
  b = <Body _>
        <GetOrderResponse>
          <orderId>OrderId</>
          <date>_</>
          <userId>u</>
          _
```

We suppose there is a single server, identified by the URL S, hosting the *GetOrder* web service, identified by the URI W, and multiple client computers that may connect to S on behalf of users. Here is a protocol for a client computer, identified by its IP address I, to request information about order number *OrderId* from the web service W on server S, on behalf of a human user u.

Message 1:	$I \rightarrow S, W \quad e$
	where $\mathit{hasBody}(e,b), \mathit{isGetOrder}(b, \mathit{OrderId})$
Message 2:	$S \rightarrow I \quad e'$
	where $\mathit{hasBody}(e',b')$,
	and $\mathit{isGetOrderResponse}(b', \mathit{OrderId}, u')$

- Message 1 is an HTTP POST request to the URL S, with an HTTP header `SOAPAction:` W, and with the SOAP envelope e as its content. The predicates $\mathit{hasBody}(e,b)$ and $\mathit{isGetOrder}(b, \mathit{OrderId})$ specify the behaviour of both client and server: that is, a client will only send Message 1, and a server will only accept it, if the message e is a suitably formatted request for some order *OrderId*. We implicitly specify that if the server receives a message that does not satisfy these predicates, it will reject the message.
- Message 2 is the HTTP response, containing the SOAP envelope e'. The predicates $\mathit{hasBody}(e',b')$ and $\mathit{isGetOrderResponse}(b', \mathit{OrderId}, u')$ constrain the server to send a reply that concerns the order *OrderId* requested in Message 1. In this first protocol, the user u whose client computer sends the request need not be the same as the user u' who is associated with the order.

It is not a goal here to fully specify the correct behaviour of either client or server. We are only concerned about security properties, and authentication in particular, and suppress other information. For example, we suppress the rest of the response, which includes details such as the credit card type, number and expiration date, billing and shipping addresses, and the sequence of line items in the order.

Our predicates express constraints on messages sent and received by compliant implementations of our protocols. On the sender side, they express post-conditions for every outgoing message. On the receiver side, they express pre-conditions that must be checked before incoming messages are processed. In the presence of an active attacker, it is essential that the receiver dynamically check these conditions, even if the sender enforces them. Our first protocol offers no protection against active attacks, since any well-formed envelope is accepted by the server.

3.2 Password Digest

Username tokens with a cryptographic digest provide a first, basic mechanism for authenticating web service requests. Such tokens include a username identity u, together with a digest of a password and a fresh timestamp. We assume that each password pwd_u is a shared, unguessable secret between u and S, so that only u (or S, in principle) can generate a valid digest—this hypothesis excludes dictionary attacks, for instance. (To justify this assumption, passwords need to be strong cryptographic secrets; one might also modify the protocol to encrypt the digest of a weak password, but we do not pursue this alternative.) Moreover, as in other applications of the applied pi calculus, we abstractly relate the password and the user using the special one-way function principal from passwords to users: we let u stand for $\mathsf{principal}(pwd_u)$.

To model this protocol, we develop predicates for describing WS-Security headers and embedded username tokens. (Our predicate definitions are not specific to this protocol, and can be re-used for any protocol relying on these tokens.) First, we define a predicate to extract the security tokens from some security header of the envelope: the predicate $\mathit{hasSecurityHeader}(e, \mathit{toks})$ means that *toks* is a sequence of security tokens attached to message e. The first formula in the predicate body extracts the list of headers ($\mathit{headers}$: items) from the envelope. The second formula, $\mathit{header} \in \mathit{headers}$, means that *header* is some member of the header list. The third formula means that *header* must be a security header, and extracts the security tokens from it.

```
hasSecurityHeader(e : item, toks : items) :-
  e = <Envelope><Header>headers</>_</>,
  header ∈ headers,
  header = <Security>toks</>.
```

With username tokens, the unique identifier of a message is a pair (n : bytes, t : string) where n is a nonce—some byte array—and t is a timestamp represented as a string. The predicate $\mathit{isDigestUserToken}(\mathit{tok}, u, pwd, n, t)$ means that *tok* is a username token for user u with password *pwd*, identifier (n,t), and a valid digest.

```
isDigestUserToken(tok : item, u, pwd : string, n : bytes, t : string) :-
  tok = <UsernameToken _>
          <Username>u</>
          <Password Type="PasswordDigest">base64(d)</>
          <Nonce>base64(n)</>
          <Created>t</>,
  u = principal(pwd),
  d = sha1(concat(n, concat(utf8(t), utf8(pwd)))).
```

Finally, a top-level authentication predicate, *hasUserTokenDigest*, gathers all the conditions checked on envelopes received by the server. $\mathit{hasUserTokenDigest}(e, u, pwd, n, t, b)$ means that the envelope e with body b contains a valid username token for u, pwd, n, t.

```
hasUserTokenDigest(e : item, u, pwd : string,
                   n : bytes, t : string, b : item) :-
  hasSecurityHeader(e, toks),
  utok ∈ toks,
  isDigestUserToken(utok, u, pwd, n, t),
  hasBody(e, b).
```

The following protocol description includes both SOAP messages and additional begin- and end-events, in the style of Woo and Lam [37]. We use these events to express the authentication guarantee obtained by the server from running this protocol.

Event 1: I logs `<Begin>`$u\ n\ t$`</>`
Message 1: $I \rightarrow S, W \quad e$
where $\mathit{hasUserTokenDigest}(e,u,\mathit{pwd},n,t,b)$,
and $\mathit{isGetOrder}(b,\mathit{OrderId})$
Event 2: S logs `<End>`$u\ n\ t$`</>`
Message 2: $S \rightarrow I \quad e'$
where $\mathit{hasBody}(e',b')$,
and $\mathit{isGetOrderResponse}(b',\mathit{OrderId},u)$

We interpret events in the abstract log as follows: before issuing a request, the initiator logs its intent as an entry `<Begin>`$u\ n\ t$`</>` that contains the user name u and the message identifier. Conversely, after checking an envelope, the server logs `<End>`$u\ n\ t$`</>` to manifest that it accepts a request with these parameters. In any case, the attacker cannot log entries. Ideally, begin- and end-events should be in direct correspondence, but this is clearly not the case if the attacker can delete, reorder, or replay u's messages. Instead, we have the following correspondence property:

CLAIM 1. *In the presence of an active Dolev-Yao attacker, if* `<End>`$u\ n\ t$`</>` *is logged by S, then* `<Begin>`$u\ n\ t$`</>` *has been logged by I.*

This is a fairly weak authentication property, which can be read as "if S accepts a request from u, then u recently sent some request". The two requests are not necessarily the same: for instance, an active attacker can intercept the envelope, modify its body, and pass it to the server. In many settings, it may be suitable to have a richer correspondence between u and S's actions, for example between entries `<Begin>`$u\ S\ W\ n\ t\ \mathit{OrderId}$`</>` and `<End>`$u\ S\ W\ n\ t\ \mathit{OrderId}$`</>`.

Although the password digest is optional in WS-Security username tokens, our claim would clearly not hold if the server accepted tokens without checking the digest, since the attacker could then forge a message with any identifier (n,t) irrespective of the user's requests.

In itself, our protocol does not eliminate replays. (Technically, our correspondence assertion is non-injective.) However, since the identifier is authenticated, the application can safely use it to filter duplicate or expired messages.

3.3 Password-based Signature

In order to achieve more precise authentication properties under the same assumptions—a shared password between u and S—one can use an XML digital signature on selected elements of the envelope [17]. In addition to the username token, we add a signature token that signs (for instance) the envelope body, with a signing key derived from the password and the username token.

A hash-based signature of items x_1, ..., x_n using a key k, may be roughly pictured as follows.

```
<Signature>
  <SignedInfo>
    <Reference>...hash of x1...</>
    ...
    <Reference>...hash of xn...</>
  <SignatureValue>
    ...hash of SignedInfo element with key k...
```

See Section 4.3 for a full example of a signed envelope. Next, we define the additional predicates needed for our modified protocol, including predicates defining the various parts of a signature.

- $\mathit{isUserTokenKey}(\mathit{tok},u,\mathit{pwd},n,t,k)$ means that tok is a username token for user u with password pwd, unique identifier (n,t), and derived key k. The key derivation uses a p-sha1 keyed hash salted with the message identifier.
- $\mathit{isSigVal}(\mathit{sv},\mathit{si},k,a)$ means that sv is the digital signature computed on the item si with key k using algorithm a (which for password-based signatures is `hmac-sha1`).
- $\mathit{ref}(t,r)$ means that the item r is a reference containing the digest of item t. (We use the three wildcards _ to match certain attributes and elements irrelevant to security here.)
- $\mathit{isSigInfo}(\mathit{si},a,x_1,\ldots,x_n)$ means that the signed information si, for signature algorithm a, contains a list of references of which the first n are for the items $x_1,\ldots,x_n$. After these references, si may contain any number of references to other items (represented in the predicate by an _). This flexibility in the predicate enables the client to sign additional items even if not required by the server (to conform to a uniform send policy, for example).
- $\mathit{isSignature}(\mathit{sig},a,k,x_1,\ldots,x_n)$ means that the signature sig signs $x_1,\ldots,x_n$ with algorithm a and verification key k.
- $\mathit{hasUserSignedBody}(e,u,\mathit{pwd},n,t,b)$ is the top-level predicate. It means that the envelope e contains a username token for u,pwd,n,t, and that the body b of e is signed by the key derived from the token.

$\mathit{isUserTokenKey}(\mathit{tok}:\text{item},u,\mathit{pwd}:\text{string},$
$\quad n:\text{bytes},t:\text{string},k:\text{bytes})$:-
$\quad \mathit{tok} =$ `<UsernameToken _>`
$\qquad$ `<Username>`u`</> _`
$\qquad$ `<Nonce>`$\text{base64}(n)$`</>`
$\qquad$ `<Created>`t`</>`,
$\quad u = \text{principal}(\mathit{pwd})$,
$\quad k = \text{p-sha1}(\mathit{pwd},\text{concat}(n,\text{utf8}(t)))$.

$\mathit{isSigVal}(\mathit{sv}:\text{bytes},\mathit{si}:\text{item},k:\text{bytes},a:\text{string})$:-
$\quad a =$ `hmac-sha1`,
$\quad \mathit{sv} = \text{hmac-sha1}(k,\text{c14n}(\mathit{si}))$.

$\mathit{ref}(t:\text{item},r:\text{item})$:-
$\quad r =$ `<Reference _>`
$\qquad$ `_ _ <DigestValue>`$\text{base64}(\text{sha1}(\text{c14n}(t)))$`</>`.

(for each $n \geq 1$)
$\mathit{isSigInfo}(\mathit{si}:\text{item},a:\text{string},x_1,\ldots,x_n:\text{item})$:-
$\quad \mathit{si} =$ `<SignedInfo>`
$\qquad$ `_ <SignatureMethod Algorithm="`a`"></>`
$\qquad r_1\ \ldots\ r_n$ _,
$\quad \mathit{ref}(x_1,r_1),\ldots,\mathit{ref}(x_n,r_n)$.

$\mathit{isSignature}(\mathit{sig}:\text{item},a:\text{string},k:\text{bytes},x_1,\ldots,x_n:\text{item})$:-
$\quad \mathit{sig} =$ `<Signature>`si `<SignatureValue>`$\text{base64}(\mathit{sv})$`</>_</>`,
$\quad \mathit{isSigInfo}(\mathit{si},a,x_1,\ldots,x_n)$,
$\quad \mathit{isSigVal}(\mathit{sv},\mathit{si},k,a)$.

$\mathit{hasUserSignedBody}(e:\text{item},u:\text{string},\mathit{pwd}:\text{string},$
$\quad n:\text{bytes},t:\text{string},b:\text{item})$:-
$\quad \mathit{hasBody}(e,b)$,
$\quad \mathit{hasSecurityHeader}(e,\mathit{toks})$,
$\quad \mathit{utok} \in \mathit{toks}$,
$\quad \mathit{isUserTokenKey}(\mathit{utok},u,\mathit{pwd},n,t,k)$,
$\quad \mathit{sig} \in \mathit{toks}$,
$\quad \mathit{isSignature}(\mathit{sig},$`hmac-sha1`$,k,b)$.

The message exchange is much as in Section 3.2, with two differences: each log entry now contains $u\ n\ t\ \mathit{OrderId}$ instead of just *u n t*; we use the top-level predicate $hasUserSignedBody(e,u,pwd,n,t,b)$ instead of $hasUserTokenDigest(e,u,pwd,n,t,b)$. We obtain a similar, but stronger authentication property:

CLAIM 2. *In the presence of an active Dolev-Yao attacker, if* `<End>`*u n t OrderId*`</>` *is logged by S,* `<Begin>`*u n t OrderId*`</>` *has been logged by I.*

This claim can be read as "if S accepts a request from u, then u recently sent this request". As before, we can rely on the identifier (n,t) for replay protection.

We make two observations concerning these predicates. First, *isUserTokenKey* does not check the presence or validity of the optional username token digest. In fact, checking the password digest would not provide any additional authentication guarantee here. Conversely, its (potential) occurrence in the envelope slightly complicates our proofs in the next section. Arguably, the initiator should not include both a digest and a signature, since this may facilitate a dictionary attack on the password, unless it does not know which evidence will be considered by the server.

Second, although each reference *r* typically provides a pointer to the digested element, either as a fragment URI or as an XPath expression, we do not rely on this information in the *ref* predicate. Instead, we check that the actual item we are interested in—the body *b*—is targeted by the reference. In general, this approach is preferable, since it leaves the resolution of pointers outside the trusted computing base. Otherwise, one should also carefully check that these pointers are well-defined and unambiguous.

Our specification captures the flexibility of WS-Security signatures. The predicates for key derivation (*isUserTokenKey*) are independent from those interpreting the signature. So, we can compose *isSignature* with some other keying material, such as an X.509 certificate. Similarly, we can support additional algorithms for computing the actual signature by adding alternatives to the predicate *isSigVal*—see Section 3.4.

Furthermore, *isSignature* allows additional elements of the message to be signed. Signing the username, nonce, or timestamp elements is not necessary with this particular signing-key derivation, but is harmless, and becomes necessary with other kinds of keys (see Section 3.5). In case there are several actions on the same server, or if the same password is shared with two different (honest) servers, then the path header (S,W) should also be signed (as in the next section). Otherwise, the attacker might redirect an envelope from one web service to another.

3.4 X.509 Signature

The next protocol does not depend on password-based authentication. Instead, it uses public-key signatures based on X.509 certificates. We assume that the user u has a public/private key pair and keeps the private key secret. We also assume that u and S agree on the public key k_r of some X.509 certification authority, and that this authority issued only one certificate for u, with u's public key.

In contrast with password-based signatures, X.509 signature tokens cannot use fragments of the username token as message identifier. Instead, they can sign the globally unique identifier included in the path header of every SOAP message, as defined in WS-Routing [30]. This is reflected by the following additional predicates:

- $isX509Token(tok,k_r,u,a,k)$ means that *tok* is a binary token that contains a certificate $\mathsf{x509}(s_r,u,a,k)$ with certifier's public key $k_r = \mathsf{pk}(s_r)$.
- $isSigVal(sv,si,k,a)$ is extended with a clause that checks signatures using the rsa-sha1 algorithm.
- $hasPathHeader(e,ac,to,id,ea,et,ei)$ means that envelope e has a path header with action *ac*, destination *to*, and message identifier *id* in elements *ea*, *et*, and *ei*, respectively.
- $hasX509SignedBody(e,k_r,u,ac,to,id,b,ea,et,ei)$ is the top-level predicate. It means that the envelope e has an X.509 token for u certified by k_r whose public key signs the body b and a path header ea,et,ei containing ac,to,id.

$isX509Token(tok:\mathsf{item},k_r:\mathsf{bytes},u:\mathsf{string},a:\mathsf{string},k:\mathsf{bytes})$:-
  $tok =$ `<BinarySecurityToken _>`$\mathsf{base64}(xcert)$`</>`,
  $\mathsf{check\text{-}x509}(xcert,k_r) = k_r$,
  $u = \mathsf{x509\text{-}user}(xcert)$,
  $a = \mathsf{x509\text{-}alg}(xcert)$,
  $k = \mathsf{x509\text{-}key}(xcert)$.

$isSigVal(sv:\mathsf{bytes},si:\mathsf{item},k:\mathsf{bytes},a:\mathsf{string})$:-
  $a =$ `rsa-sha1`, $\mathsf{check\text{-}rsa\text{-}sha1}(\mathsf{c14n}(si),sv,k) = k$.

$hasPathHeader(e:\mathsf{item},ac,to,id:\mathsf{string},ea,et,ei:\mathsf{item})$:-
  $e =$ `<Envelope><Header>`*headers*`</>_</>`,
  $header \in headers$,
  $header =$ `<path _>`*ea et ei*`</>`,
  $ea =$ `<action _>`*ac*`</>`,
  $et =$ `<to _>`*to*`</>`,
  $ei =$ `<id _>`*id*`</>`.

$hasX509SignedBody(e:\mathsf{item},k_r:\mathsf{bytes},u,ac,to,id:\mathsf{string},$
    $b,ea,et,ei:\mathsf{item})$:-
  $hasBody(e,b)$,
  $hasPathHeader(e,ac,to,id,ea,et,ei)$,
  $hasSecurityHeader(e,toks)$,
  $xtok \in toks$,
  $isX509Token(xtok,k_r,u,$`rsa-sha1`$,k)$,
  $sig \in toks$,
  $isSignature(sig,$`rsa-sha1`$,k,b,ea,et,ei)$.

The message exchange for the X.509 signature protocol is almost the same as the one in Section 3.3, with two differences. First, the contents of the log entries is now *u W S id OrderId* (instead of *u n t OrderId*). Second, we use the top-level predicate $hasX509SignedBody(e,k_r,u,W,S,id,b,ea,et,ei)$ instead of *hasUserSignedBody*. We now obtain the authentication property:

CLAIM 3. *In the presence of an active Dolev-Yao attacker, if* `<End>`*u W S id OrderId*`</>` *is logged by S, then* `<Begin>`*u W S id OrderId*`</>` *has been logged by I.*

This claim can be read as "if S accepts a request from u, then u, at some point, sent this request to S". So by signing the path header, we obtain an additional authenticity guarantee as regards u's intended target (S,W), and thus prevent some redirection attacks. One can easily implement replay protection using the authenticated message identifier. This supposes that clients do generate globally unique id's (although this is not actually required to obtain our correspondence property). Alternatively, one may use a custom unique identifier in the envelope body.

3.5 Firewall-Based Authentication

By specifying the structure of security tokens, rather than their use, WS-Security encourages a flexible approach to web service security. For instance, a server may naturally accept both password-based and X.509-based signatures for authentication, leaving that choice to the client. This flexibility yields useful compositional properties in our formal developments. For instance, a web service that runs both protocols is formally equivalent to two web services in parallel, one for each authentication mechanism.

In this section, we illustrate this flexibility with a different composite architecture that chains WS-Security authentication schemes along a WS-Routing path. In addition to a server S and a client I acting on behalf of u, we consider an intermediate SOAP-level firewall F. The firewall holds the password database, and is responsible for authenticating access to S (and possibly other servers). The client I sends a *GetOrder* request with a username signature (for u) to S via F. The path header indicates to F that the message is intended for S. The firewall F checks the password-based signature, adds a new `firewall` header indicating that it has authenticated u, signs the message using its X.509 certificate, and forwards the message to S. The server S checks the X.509 signature, and thus authenticates the original sender u without knowledge of u's password.

Next, we define (predicates on) the message forwarded by the firewall. To indicate to the server that it has checked the credentials of the user, the firewall adds a new firewall header containing the username token, but with the password digest deleted. It then embeds an X.509 signature that includes this header as well. The predicates for this message are:

- $\mathit{isFirewallHeader}(h,u,n,t)$ means that the envelope h is a firewall header with the username token u,n,t.
- $\mathit{hasFWHeader}(e,h,u,n,t)$ means that the envelope e has a firewall header h with u,n,t.
- $\mathit{hasX509SignedBodyFw}(e,k_r,f,u,n,t,b)$ is the top-level predicate checked by the server. It means that the envelope e has a firewall header with u,n,t, body b, and is signed with a valid certificate for f issued by k_r.

$$\begin{array}{l}
\mathit{isFirewallHeader}(h:\mathsf{item},u:\mathsf{string},n:\mathsf{bytes},t:\mathsf{string})\ :-\\
\quad h = \texttt{<firewall \_>}\mathit{utok}\texttt{</>},\\
\quad \mathit{utok} = \texttt{<UsernameToken>}\\
\qquad\qquad \texttt{<Username>}u\texttt{</>}\\
\qquad\qquad \texttt{<Nonce>}\mathsf{base64}(n)\texttt{</>}\\
\qquad\qquad \texttt{<Created>}t\texttt{</>}.
\end{array}$$

$$\begin{array}{l}
\mathit{hasFWHeader}(e,h:\mathsf{item},u:\mathsf{string},n:\mathsf{bytes},t:\mathsf{string})\ :-\\
\quad e = \texttt{<Envelope \_><Header>}\mathit{headers}\texttt{</>\_</>},\\
\quad h \in \mathit{headers},\\
\quad \mathit{isFirewallheader}(h,u,n,t).
\end{array}$$

$$\begin{array}{l}
\mathit{hasX509SignedBodyFw}(e:\mathsf{item},k_r:\mathsf{bytes},f,u:\mathsf{string},\\
\qquad\qquad\qquad\qquad\qquad n:\mathsf{bytes},t:\mathsf{string},b:\mathsf{item})\ :-\\
\quad \mathit{hasBody}(e,b),\\
\quad \mathit{hasFWHeader}(e,h,u,n,t),\\
\quad \mathit{hasSecurityHeader}(e,\mathit{toks}),\\
\quad \mathit{xtok} \in \mathit{toks},\\
\quad \mathit{isX509Token}(\mathit{xtok},k_r,f,\texttt{rsa-sha1},p),\\
\quad \mathit{sig} \in \mathit{toks},\\
\quad \mathit{isSignature}(\mathit{sig},\texttt{rsa-sha1},p,b,h).
\end{array}$$

The protocol involves three messages, as follows:

Event 1:	I logs `<Begin>`$u\ n\ t\ \mathit{OrderId}$`</>`
Message 1:	$I \rightarrow F, W \quad e$
	where $\mathit{hasUserSignedBody}(e,u,\mathit{pwd},n,t,b)$
Message 2:	$F \rightarrow S, W \quad e'$
	where $\mathit{hasX509SignedBodyFw}(e',k_r,f,u,n,t,b)$
	and $\mathit{isGetOrder}(b,\mathit{OrderId})$
Event 2:	S logs `<End>`$u\ n\ t\ \mathit{OrderId}$`</>`
Message 3:	$S \rightarrow I \quad e''$
	where $\mathit{hasBody}(e'',b')$
	and $\mathit{isGetOrderResponse}(b',\mathit{OrderId},u)$

CLAIM 4. *In the presence of an active Dolev-Yao attacker, if* `<End>`$u\ n\ t\ \mathit{OrderId}$`</>` *is logged by* S*, then* `<Begin>`$u\ n\ t\ \mathit{OrderId}$`</>` *has been logged by* I*.*

Thus, we obtain the same end-to-end authenticity guarantee as with the password-based signature protocol of Section 3.3, but for a different implementation that does not require S to know u's password. We prove this claim by composing the correspondence property for the password-based signature in Message 1 with that for the X.509 signature in Message 2.

4 A Pi Calculus Semantics

In order to validate the claims of Section 3, we specify the behaviour of the participants (and in particular their implementation of predicates) as processes in the applied pi calculus. We refer to [1] for a general presentation of the calculus, and rely on their definitions for processes and their semantics. Here, we use the sorts, terms, and equations described in Section 2, with coercion functions from strings to items, and with additional sorts for communication channels [27]. We always assume that terms, formulas, processes, and contexts are well-sorted, but usually keep sort information implicit.

This section divides into the following parts. Section 4.1 describes our computational interpretation of formulas as certain nondeterministic processes in the applied pi calculus. Section 4.2 introduces formal notions of robust safety—that embedded correspondence assertions hold in spite of the presence of an attacker—and functional adequacy—that a protocol may run to successful completion in the absence of an attacker. Section 4.3 uses these definitions to state results about the username-signing protocol of Section 3.3. Theorem 1 asserts that a process formalizing this protocol is robustly safe—our previous claim is a corollary. Moreover, Theorem 2 asserts the formalization is functionally adequate. Section 4.4 gives the structure of our proof for Theorem 1, which relies on a decomposition of the protocol into simpler components.

The technical report contains a brief overview of the applied pi calculus, detailed proofs of these two theorems and of their counterparts for the other protocols of Section 3, and an account of how to generalize our results to a situation with multiple servers and users.

4.1 Interpretation of Formulas

We describe a (partial) implementation of our logic in the applied pi calculus. We inductively define processes of the form *filter* $\Phi \mapsto \widetilde{y}$ *in* P, where the variables $\widetilde{y}$ are bound in P and get assigned to terms making the formula Φ true. When the formula is an equality $V = T$ we assume that one of the terms is known, and that the other can be treated as a pattern, matching variables to known subterms in the known term. In the following formal definitions, we always assume that V is the known term, and that T is the pattern, but in our example predicates we allow either of the terms to be the pattern. For a pattern to be implementable, there must be an inverse term for each bound variable, able to compute the value of the variable from the known term.

Patterns:

The equality $V = T$ binds variables $\widetilde{y}$ with pattern T, written $V = T \mapsto \widetilde{y}$, when (1) $\widetilde{y} \subseteq \mathit{fv}(T) \setminus \mathit{fv}(V)$, and (2) T has *inverse terms* $\widetilde{S}$, with $\mathit{fv}(\widetilde{S}) \subseteq \{x\}$, $\mathit{fn}(\widetilde{S}) = \varnothing$, and, for all terms $V, \widetilde{W}$, if $V = T\{\widetilde{y} = \widetilde{W}\}$, then $\widetilde{W} = \widetilde{S}\{x = V\}$.

For instance, the pattern $\mathsf{base64}(y)$ has inverse $S \triangleq \mathsf{i\text{-}base64}(x)$; for all V and W, if $V = \mathsf{base64}(W)$ then $W = S\{x = V\} = \mathsf{i\text{-}base64}(\mathsf{base64}(W))$. On the other hand, the pattern $\mathsf{sha1}(y)$ has no inverse, and therefore would not satisfy point (2).

The following table is the partial inductive definition of *filter* $\Phi \mapsto \widetilde{y}$ *in* P. If such a process is defined by the following rules, we say that the formula Φ is *implementable* with bound variables $\widetilde{y}$. When $\mathit{filter}\ \Phi \mapsto \widetilde{y}\ \mathit{in}\ P$ is defined and closed, we intend that it seeks closed terms $\widetilde{V}$ such that $\models \Phi\{\widetilde{y}=\widetilde{V}\}$, and acts as $P\{\widetilde{y}=\widetilde{V}\}$. We refer to the technical report for a formal statement of this property.

Formula Implementation: $\mathit{filter}\ \Phi \mapsto \widetilde{y}\ \mathit{in}\ P$ **when** $\widetilde{y} \subseteq \mathit{fv}(\Phi)$

$$\begin{array}{l}
\mathit{filter}\ V = T \mapsto \widetilde{y}\ \mathit{in}\ P \quad \triangleq \\
\qquad \mathit{let}\ \widetilde{y} = \widetilde{S}\{x=V\}\ \mathit{in\ if}\ V = T\ \mathit{then}\ P \\
\qquad \text{when } V = T \mapsto \widetilde{y} \text{ with inverse terms } \widetilde{S} \\
\mathit{filter}\ x \in V \mapsto x\ \mathit{in}\ P \quad \triangleq \\
\qquad \nu s,c.(c(x).P \mid \overline{s}\langle V\rangle \mid !s(z).\mathit{filter}\ z = h\ t \mapsto h,t\ \mathit{in}\ (\overline{c}\langle h\rangle \mid \overline{s}\langle t\rangle)) \\
\qquad \text{when } x \notin \mathit{fv}(V) \text{ and with } \{s,c\} \cap \mathit{fn}(P) = \varnothing \\
\mathit{filter}\ p(\widetilde{V}) \mapsto \widetilde{y}\ \mathit{in}\ P \quad \triangleq \\
\qquad \nu s.(\overline{s}\langle\varepsilon\rangle \mid \prod_{i\in 1..n} s(\_).\mathit{filter}\ \Phi_i\{\widetilde{x}=\widetilde{V}\} \mapsto \widetilde{y},\widetilde{z}_i\ \mathit{in}\ P) \\
\qquad \text{when } p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n,\ s \notin \mathit{fn}(P) \\
\qquad \text{and, } \forall i \in 1..n, \mathit{fv}(\Phi_i) = \widetilde{x} \uplus \widetilde{z}_i \text{ and } (\mathit{fv}(\widetilde{V}) \cup \mathit{fv}(P)) \cap \widetilde{z}_i = \varnothing \\
\mathit{filter}\ \Phi_1, \Phi_2 \mapsto \widetilde{y}\ \mathit{in}\ P \quad \triangleq \\
\qquad \mathit{filter}\ \Phi_1 \mapsto (\widetilde{y} \cap \mathit{fv}(\Phi_1))\ \mathit{in}\ (\mathit{filter}\ \Phi_2 \mapsto (\widetilde{y} \setminus \mathit{fv}(\Phi_1))\ \mathit{in}\ P)
\end{array}$$

When $V = T \mapsto \widetilde{y}$, with inverse terms $\widetilde{S}$, $\mathit{filter}\ V = T \mapsto \widetilde{y}\ \mathit{in}\ P$ binds the variables $\widetilde{y}$ of the pattern T to components of the term V, and verifies that hence the pattern matches the term. If so, the match succeeds, and P runs. Otherwise, the match fails, and the implementation deadlocks.

When $x \notin \mathit{fv}(V)$, $\mathit{filter}\ x \in V \mapsto x\ \mathit{in}\ P$ outputs V on a fresh channel s, and runs the process $!s(z).\mathit{filter}\ z = h\ t \mapsto h,t\ \mathit{in}\ (\overline{c}\langle h\rangle \mid \overline{s}\langle t\rangle)$ which binds $h = V_1$ and $t = V_2 \dots V_n\,\varepsilon$, provided $V = V_1\,V_2 \dots V_n\,\varepsilon$ with $n \geq 1$, then outputs h on c, and t on the fresh channel s. The effect of this replication is to output each of the terms $V_1, \dots, V_n$ on the fresh channel c. The process $c(x).P$ is the only listener on c; so the outcome is $P\{x = V_i\}$ for some $i \in 1..n$. If, in fact, V is the empty list, the implementation deadlocks.

When $p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n$, $\mathit{filter}\ p(\widetilde{V}) \mapsto \widetilde{y}\ \mathit{in}\ P$ generates a separate process $s(\_).\mathit{filter}\ \Phi_i\{\widetilde{x}=\widetilde{V}\} \mapsto \widetilde{y},\widetilde{z}_i\ \mathit{in}\ P)$ for each clause $i \in 1..n$, where $\widetilde{z}_i$ are the local variables for clause i. We make an internal choice of which to run by arranging all to listen on the fresh channel s, on which only a single message is sent.

The implementation $\mathit{filter}\ \Phi_1, \Phi_2 \mapsto \widetilde{y}\ \mathit{in}\ P$ works by evaluating Φ_1 then Φ_2 before running P.

As an example, we show an implementation of $\mathit{hasBody}(e,b)$:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{filter}\ \mathit{hasBody}(e,b) \mapsto b\ \mathit{in}\ [\text{-}]} \\
= & \nu s.(\overline{s}\langle\varepsilon\rangle \mid s(\_). \\
& \mathit{filter}\ e = \texttt{<Envelope><Header>}y_1\texttt{</>}b\texttt{</>} \mapsto y_2, b\ \mathit{in} \\
& \quad \mathit{filter}\ b = \texttt{<Body } y_2\texttt{>}y_3\texttt{</>} \mapsto y_2, y_3\ \mathit{in}\ [\text{-}]) \\
= & \nu s.(\overline{s}\langle\varepsilon\rangle \mid s(\_). \\
& \mathit{let}\ y_1 : \mathsf{items} = \texttt{Header}.\mathsf{body}(\mathsf{hd}(\texttt{Envelope}.\mathsf{body}(e)))\ \mathit{in} \\
& \mathit{let}\ b : \mathsf{item} = \mathsf{hd}(\mathsf{tl}(\texttt{Envelope}.\mathsf{body}(e)))\ \mathit{in} \\
& \mathit{if}\ e = \texttt{<Envelope><Header>}y_1\texttt{</>}\ b\ \varepsilon\texttt{</>}\ \mathit{then} \\
& \quad \mathit{let}\ y_2 : \mathsf{atts} = \texttt{Body}.\mathsf{att}(b)\ \mathit{in} \\
& \quad \mathit{let}\ y_3 : \mathsf{items} = \texttt{Body}.\mathsf{body}(b)\ \mathit{in} \\
& \quad \mathit{if}\ b = \texttt{<Body } y_2\texttt{>}y_3\texttt{</>}\ \mathit{then}\ [\text{-}])
\end{array}$$

4.2 Safety Properties, Functional Properties

To formalize the authenticity properties claimed in Section 3, we mark the progress of the client and server processes with begin- and end-events, represented as message outputs on the channels *begin* and *end*, respectively. Hence, our authenticity properties become non-injective correspondence assertions [37] between messages. To capture the occurrence of one of these events, we define a notion of observation of messages on free channels. We write $\approx$ for (weak) observational congruence in applied pi.

Observation: $A \triangleright \overline{a}\langle V\rangle$

A outputs V on channel a, written $A \triangleright \overline{a}\langle V\rangle$, when $A \approx \overline{a}\langle V\rangle \mid A'$.

Much as in Gordon and Jeffrey's formulation of correspondence assertions [19], we define safety and robust safety: a process is safe if every end-event has a matching begin-event, and is robustly safe if it is safe in the presence of any opponent. We write $\rightarrow^*$ for a series of reduction steps.

Safety and Robust Safety:

A is *safe* if and only if, whenever $A \rightarrow^* B$, $B \triangleright \overline{\mathit{end}}\langle V\rangle$ implies $B \triangleright \overline{\mathit{begin}}\langle V\rangle$.
A is *robustly safe* if and only if, for all evaluation contexts E where the channels *begin* and *end* do not occur, $E[A]$ is safe.

Intuitively, E represents any active attacker (in the applied pi calculus) that controls both the network and the client application behaviour, A is the initial configuration of the protocol being considered, and B represents any reachable state of the protocol, after interleaving any number of sessions.

In addition to security properties such as robust safety, one should check that the protocol works as intended and may indeed succeed, at least in the absence of an attacker:

Functional Adequacy:

A is *functionally adequate for* V when $A \rightarrow^* B$ with $B \triangleright \overline{\mathit{end}}\langle V\rangle$ for some B.

The next lemma states that our main security properties can be established using the theory of observational equivalence in the applied pi calculus.

LEMMA 1. *Suppose $A \approx B$. If A is robustly safe then so is B. Moreover, if A is functionally adequate for V then so is B.*

Moreover, logical equivalence, when lifted to processes, also preserves robust safety.

Logical Equivalence of Processes:

Two processes are logically equivalent when they only differ in their choices of implementable, logically-equivalent predicates.

LEMMA 2. *Logical equivalence preserves robust safety.*

4.3 Stating Password-Based Authentication

We are now ready to formulate and prove Claim 2 of Section 3.3 for envelopes with password-based signatures, with or without a password digest. (The other claims in the paper are handled similarly.) For the sake of simplicity, we focus on protocol configurations Q with a single user u, with initiator process I_u and a single server S_u that share a secret password with that user, represented as a restricted name s_{pwd}. The two parts of the protocol also share a communication channel, *http*. Since *http* is not restricted, an environment that encloses Q can also read, modify, and write any SOAP message.

Protocol Configurations: Q **(parameterized by** *Envelope***)**

$$\begin{array}{rcl}
Q & \triangleq & \nu s_{pwd}.(\{u = \mathsf{principal}(s_{pwd})\} \mid I_u \mid S_u) \\
I_u & \triangleq & !\mathit{init}_u(n,t,b).(\overline{\mathit{begin}}\langle u\ n\ t\ b\rangle \mid \overline{\mathit{http}}\langle \mathit{Envelope}\rangle) \\
S_u & \triangleq & !\mathit{http}(e).\mathit{filter}\ \mathit{hasUserSignedBody}(e,u',s_{pwd},n,t,b) \\
& & \qquad \mapsto u',n,t,b\ \mathit{in}\ \overline{\mathit{end}}\langle u'\ n\ t\ b\rangle
\end{array}$$

The initiator, I_u, repeatedly receives high-level requests on a control channel $init_u$. Using that control channel, the environment can thus initiate any number of requests on behalf of u, for any terms N,T,B. These requests are "genuine": they are echoed on channel *begin*. The process I_u is also parameterized by a term *Envelope* that determines the actual SOAP envelopes constructed and sent by the initiator.

The server repeatedly receives low-level envelopes on channel *http*, filters them using the top-level predicate defined in Section 3.3 (one easily checks that this predicate is implementable) and finally sends a message on channel *end* for each accepted envelope. (More generally, we would represent a server that accepts requests from users $u_1,\dots,u_n$ as a parallel composition $\prod_{i\in 1..n} S_{u_i}$.)

The scope restriction on s_{pwd} models our secrecy assumption on the password, essentially supposing that it is a strong secret shared between the initiator and the server and used only in this kind of envelope.

The active substitution $\{u = \mathsf{principal}(s_{pwd})\}$ binds the variable u to the expression $\mathsf{principal}(s_{pwd})$, and exports u (but not s_{pwd}) to the environment.

Crucially, we do not want our robust safety result to depend on every detail of the envelope. Instead, we express minimal requirements as follows:

Safe Envelopes:

A *safe envelope* is a term of the form *Envelope* $= T\varphi$ with

$$\begin{aligned}\varphi \;\triangleq\; \{&d = \mathsf{sha1}(\mathsf{concat}(n,\mathsf{concat}(\mathsf{utf8}(t),\mathsf{utf8}(s_{pwd})))),\\ &sv = \mathsf{hmac\text{-}sha1}(\mathsf{p\text{-}sha1}(s_{pwd},\mathsf{concat}(n,\mathsf{utf8}(t))),\\ &\qquad \mathsf{c14n}(SI))\}\end{aligned}$$

for some terms T,SI such that $s_{pwd} \notin fn(T,SI)$ and $isSigInfo(SI,\texttt{hmac-sha1},b)$ is valid.

To elaborate, as regards safety properties, *Envelope* may be any XML term, as long as the password occurs at most in the digest and signature values. Similarly, most of the subterms in the signature information are irrelevant for safety, even if they happen to be signed in *SI*.

THEOREM 1. *For any safe envelope, the configuration Q is robustly safe.*

From this theorem and the definition of $isGetOrder(b, orderId)$, we easily derive the more specific claim of Section 3.3.

For functional adequacy, the structure of the envelope is more constrained. For example, T and SI can be instantiated as follows:

```
T ≜ <Envelope>
      <Header>
        <Security>
          <UsernameToken Id="utoken">
            <Username>u</>
            <Password Type="PasswordDigest">
              base64(d)
            <Nonce>base64(n)</>
            <Created>t</>
          <Signature>
            SI
            <SignatureValue>base64(sv)</>
            <KeyInfo>
              <SecurityTokenReference>
                <Reference URI="#utoken"></>
      b
```

```
SI ≜ <SignedInfo>
       <CanonicalizationMethod Algorithm="c14n"></>
       <SignatureMethod Algorithm="hmac-sha1"></>
       <Reference URI="#body">
         <Transforms>
           <Transform Algorithm="c14n"></>
         <DigestMethod Algorithm="sha1"></>
         <DigestValue>base64(sha1(c14n(b)))</>
```

THEOREM 2. *The envelope $T\varphi$ with T and SI defined above is safe and, for any ground terms N : bytes, T : string, B : item with $B =$ `<Body Id="body">_</>`, the configuration $\overline{init_u}\langle N,T,B\rangle \mid Q$ with that envelope is functionally adequate for the term $u\ N\ T\ B$.*

Conversely, by Theorem 1, if we have $\overline{init_u}\langle N,T,B\rangle \mid Q \rightarrow^* A$ and $A \triangleright \overline{end}\langle u'\ N'\ T'\ B'\rangle$, then $u',N',T',B' = u,N,T,B$.

4.4 Proving Password-Based Authentication

We present the structure of the proof for Theorem 1. We refer to the technical report for additional lemmas and proofs. An intuition behind the proof is that the security property relies only on a few elements in the envelope. For instance, the signature bytes are sufficient for authentication, whereas the other elements in the envelope only provide the server with (untrusted) hints to verify the signature. Hence, to establish robust safety, we rely on a stronger, more specific lemma about a core protocol that explicitly deals only with these bytes.

The proof is in two stages. First, we show how the password-based signature protocol can be decomposed into a "core protocol" that deals with authentication and an XML wrapper. The XML wrapper has no access to the password, and need not be trusted: formally, it becomes part of the hostile environment. We show that it is enough to prove robust safety for the core protocol (Lemma 4). In the second stage, we prove that the core protocol itself is robustly safe (Lemma 7) by exhibiting an invariant on its reachable states (Lemma 6).

We decompose $hasUserSignedBody(e,u,pwd,n,t,b) \mapsto u,n,t,b$ into two implementable formulas

$$\begin{aligned}&hasUserSignatureEvidence(e,u,n,t,b,sv,si) \mapsto u,n,t,b,sv,si,\\ &checkEvidence(sv,si,u,pwd,n,t,b) \mapsto \varnothing\end{aligned}$$

hasUserSignatureEvidence parses the envelope and extracts the bits that are needed to verify the signature; it has no access to the password. (We refer to the technical report for details.) All the authentication-related checks are contained in *checkEvidence*, which is defined as follows:

$$\begin{aligned}&checkEvidence(sv:\mathsf{bytes}, si:\mathsf{item}, u, pwd:\mathsf{string}, n:\mathsf{bytes},\\ &\qquad\qquad t:\mathsf{string}, x_1,\dots,x_m:\mathsf{item}) \;{:}\!-\\ &\quad isSigInfo(si,\texttt{hmac-sha1},x_1,\dots,x_m),\\ &\quad u = \mathsf{principal}(pwd),\\ &\quad k = \mathsf{p\text{-}sha1}(pwd,\mathsf{concat}(n,\mathsf{utf8}(t))),\\ &\quad isSigVal(sv,si,k,\texttt{hmac-sha1}).\end{aligned}$$

We can state the correctness of this decomposition in terms of logical equivalence.

LEMMA 3. *The formula $hasUserSignedBody(e,u,pwd,n,t,b)$ and its decomposition defined above are logically equivalent.*

Using this decomposition, we define the core protocol configuration Q°, a counterpart of Q for the simpler predicate *checkEvidence* that binds no variables, and for replicated processes I_u° and S_u° that communicate with the environment on channels c and s, respectively, instead of channel *http*.

Core Protocol Configurations: Q°

$$\begin{aligned}
Q^\circ[\text{-}] &\triangleq \nu s_{pwd}.(\{u = \mathsf{principal}(s_{pwd})\} \mid I^\circ_u \mid S^\circ_u \mid [\text{-}])\\
I^\circ_u &\triangleq\ !init_u(n,t,b).(\overline{begin}\langle u\ n\ t\ b\rangle \mid \overline{c}\langle d, sv, SI, u, n, t, b\rangle\varphi)\\
S^\circ_u &\triangleq\ !s(sv, si, u', n, t, b).\mathit{filter\ checkEvidence}\\
&\qquad (sv, si, u', s_{pwd}, n, t, b) \mapsto \varnothing\ \mathit{in}\ \overline{end}\langle u'\ n\ t\ b\rangle
\end{aligned}$$

We write Q° for $Q^\circ[\mathbf{0}]$ (the initial state of the core protocol). Lemma 4 shows that this core protocol is logically equivalent, under an evaluation context, to the original protocol. This implies that if Q° is robustly safe, so is Q.

LEMMA 4 (XML/CORE). *For any safe envelope, there exists an evaluation context $E_Q[\text{-}]$ where the names begin, end do not occur and a process $Q^\bullet$ logically equivalent to Q such that $Q^\bullet \approx E_Q[Q^\circ]$.*

To prove robust safety for the core protocol, we first define the valid states of the core protocol in an evaluation context. Valid states are our correctness invariant. They describe protocol states reachable from Q° in which no secrets have been leaked and only messages sent by the client have been accepted by the server.

Valid States for the Core Protocol:

(1) φ_i is adapted from φ in the definition of safe envelopes with variables d_i, sv_i, n_i, t_i, b_i and term SI_i instead of $d, \underline{sv, n, t}, b$ and SI.
(2) A *session state* is a process of the form $C_i = \overline{begin}\langle u\ n_i\ t_i\ b_i\rangle \mid \varphi_i \mid J_i$ where J_i is any parallel composition of processes from $\{\overline{end}\langle u\ n_i\ t_i\ b_i\rangle\} \cup \bigoplus\{\overline{end}\langle u\ n_i\ t_i\ b_i\rangle\} \cup \bigoplus\{\}$ (with free variables n_i, t_i, b_i and defined variables d_i, sv_i). (The operator $\bigoplus$ represents internal choice over a set of processes.)
(3) An *internal state* is a parallel composition of session states $C = \prod_{i<n} C_i$, for some $n \geq 0$.
(4) A *valid state* is a closed process of the form $A = E[Q^\circ[C]]$ where $E[\text{-}]$ is an evaluation context where *begin* and *end* do not occur and C is an internal state.

For a given internal state C, let σ_C be the (ordinary) substitution obtained by composing $\{u = \mathsf{principal}(s_{pwd})\}$ and each φ_i for $i = 0 \ldots n-1$. By definition, the frame obtained from $Q^\circ[C]$, which represents the attacker's knowledge about s_{pwd}, is $\varphi_C = \nu s_{pwd}.\sigma_C$.

The next lemma states that if a message is received in a valid state of the protocol, and it satisfies the predicate *checkEvidence*, then it must have been sent by the client.

LEMMA 5 (*checkEvidence* IS SAFE). *Let C be an internal state with $n \geq 0$ sessions. Let σ' be a substitution that ranges over open terms where the name s_{pwd} does not appear such that $\sigma \triangleq \sigma' \mid \sigma_C$ is closed.*

If $\models checkEvidence(sv, si, u', s_{pwd}, n, t, b)\sigma$, then there exists $i < n$ such that $(u', sv, si, n, t, b = u, sv_i, SI_i, n_i, t_i, b_i)\sigma$.

Using this lemma, we can show that all reachable configurations of the core protocol are valid states.

LEMMA 6 (INVARIANT LEMMA). *If A is a valid state and $A \rightarrow T$ then $T \sim A'$ for some valid state A'.*

As a corollary, we can show robust safety for the core protocol.

LEMMA 7 (CORE ROBUST SAFETY). *Q° is robustly safe.*

Theorem 1 follows as a corollary. By Lemma 7, Q° is robustly safe (RS). By Lemma 4, $Q^\bullet \approx E_Q[Q^\circ]$ and, by hypothesis on E_Q, $E_Q[Q^\circ]$ is RS. By Lemma 1, $Q^\bullet$ is RS. Finally, $Q^\bullet$ is logically equivalent to Q, and thus, by Lemma 2, Q is RS.

5 Conclusions and Future Work

In this paper, we introduced a framework for reasoning about the security of SOAP protocols and their cryptographic implementations in terms of WS-Security tokens. We illustrated our framework using a series of simple authentication protocols. Surprisingly, perhaps, these XML-based protocols can be studied at the same (syntactic) level of abstraction:

- formally, using a faithful, predicate-based implementation in the applied pi calculus with proofs of correspondence properties against a Dolev-Yao adversary;
- experimentally, using sample programs and SOAP traces on top of the WSE toolkit [26].

This should provide a principled basis for testing compliant implementations, and also reduce the risk of attacks in concrete refinements of correct, abstract protocols.

As can be expected, this also complicates the formal model, with for example a large syntax and equational theory for terms in the applied pi calculus. However, our experience suggests that a modular definition of predicates, together with standard compositional techniques in the pi calculus, should enable a good reuse of the proof effort for numerous WS-Security protocols.

Our choice of authentication protocols stresses that small variations in WS-Security envelope formats may lead to much weaker correspondence properties. Each service should therefore clearly prescribe (and enforce) the intended property. Specifically, a prudent practice in the selection of XML signatures is to request that all potentially relevant headers be jointly authenticated—not just the message identifier or its body. In the case authentication relies on username tokens, this strongly suggests the use of a signature instead of a digest. Moreover, XML signatures have a complex structure, which should be used with caution. Specifically, authentication should not rely on signed elements whose interpretation depends on an unsigned context.

Related Work. There have been many formal studies of remote procedure call (RPC) security mechanisms. The earliest we are aware of is the formalization within the BAN logic [9] of Secure RPC [33] in the Andrew distributed computing environment. More recently, process calculi [2] have been used to formalize the secure implementation of programming abstractions such as communication channels and network objects [36].

We are aware of very little prior formal work on XML security protocols. Gordon and Pucella [20] implement and verify attribute-driven SOAP-level security protocols, but do not use the WS-Security syntax. Their representation of SOAP messages abstracts many details of the XML wire format, and hence would be blind to any errors in the detailed structure of names or signatures. Damiani *et al.* [12] describe an access control model for SOAP messages, but rely on a secure transport rather than WS-Security; a subsequent paper [13] discusses connections between SOAP security and authorization languages such as SAML and XACML.

Future Work. Our approach to authenticity properties should easily extend to complementary security properties, such as secrecy and anonymity. Similarly, we should be able to deal with more complex protocols (with series of related messages) and configurations (with more principals and roles). Our predicate structure is quite modular, with predicates being re-used in different protocols. Hence, we are hopeful that the method will scale up. Moreover, our semantics appears to be suitable for automation, and we are investigating how to automate the proofs using Blanchet's recent logic-based tool for applied pi [5].

At this stage, we are exploring the range of WS-Security protocols, rather than attempting its thorough description. Our fragment of WS-Security omits certain features such as Kerberos tokens and encryption but we see no fundamental barrier to modelling all of WS-Security.

Finally, although all the protocols are implemented using WSE, our goal has not been to verify the WSE implementation itself. Still, we are investigating ways of verifying at least parts of that implementation by relating it to our semantics.

Acknowledgements. We thank Tony Hoare, Riccardo Pucella, and the anonymous reviewers for their comments.

6 References

[1] 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.

[2] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 302–315, 2000.

[3] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, C. Kaler, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuk, and D. Simon. Web services security (WS-Security), version 1.0. `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-security.asp`. Draft submitted to OASIS Web Services Security TC, April 2002.

[4] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. Technical Report MSR–TR–2003–83, Microsoft Research, 2003.

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

[6] 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, `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[7] J. Boyer. *Canonical XML*, 2001. W3C Recommendation, `http://www.w3.org/TR/2001/REC-xml-c14n-20010315/`.

[8] J. Boyer, D. E. Eastlake, and J. Reagle. *Exclusive XML Canonicalization*, 2002. W3C Recommendation, `http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/`.

[9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

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

[11] J. Cowan and R. Tobin. *XML Information Set*, 2001. W3C Recommendation, `http://www.w3.org/TR/2001/REC-xml-infoset-20011024/`.

[12] 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.

[13] E. Damiani, S. De Capitani di Vimercati, and P. Samarati. Towards securing XML web services. In *ACM Workshop on XML Security 2002*, pages 90–96, 2003.

[14] T. Dierks and C. Allen. The TLS protocol: Version 1.0, 1999. RFC 2246.

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

[16] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), 2001. RFC 3174.

[17] D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation, `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`.

[18] C. Fournet and M. Abadi. Hiding names: Private authentication in the applied pi calculus. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems. Mext-NSF-JSPS International Symposium, Tokyo, Nov. 2002 (ISSS'02)*, volume 2609 of *LNCS*, pages 317–338. Springer, 2003.

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

[20] A. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *ACM Workshop on XML Security 2002*, pages 18–29, 2003.

[21] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003. RFC 3447.

[22] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[23] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, 1997. RFC 2104.

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

[25] Microsoft Corporation. *Microsoft .NET Pet Shop*, 2002. `http://www.gotdotnet.com/team/compare/petshop.aspx`.

[26] Microsoft Corporation. *Web Services Enhancements for Microsoft .NET*, Dec. 2002. `http://msdn.microsoft.com/webservices/building/wse/default.aspx`.

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

[28] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security*, Aug. 2003. Available from `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.

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

[30] H. F. Nielsen and S. Thatte. Web services routing protocol (WS-Routing). `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp`, October 2001.

[31] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[32] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, November 1984.

[33] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Trans. Comput. Syst.*, 7(3):247–280, 1989.

[34] J. Siméon and P. Wadler. The essence of XML. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 1–13, 2003.

[35] F. Thayer Fábrega, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[36] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 211–221, 1996.

[37] 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.

Appendix: The Applied Pi Calculus (Overview)

The applied pi calculus is a simple, general extension of the pi calculus with value passing, primitive function symbols, and equations between terms. Abadi and Fournet [1], introduce this calculus, develop semantics and proof techniques, and apply those techniques in reasoning about some security protocols. This appendix gives only a brief overview derived from [18].

In the applied pi calculus, the constructs of the classic pi calculus can be used to represent concurrent systems that communicate on channels, and function symbols can be used to represent cryptographic operations and other operations on data. Large classes of important attacks can also be expressed in the applied pi calculus, as contexts. These include the typical attacks for which a symbolic, mostly "black-box" view of cryptography suffices (but not for example some lower-level attacks that depend on timing behaviour or dictionary attacks). Some of the properties of the protocol can be nicely captured in the form of equivalences between processes. Moreover, some of the properties are sensitive to the equations satisfied by the cryptographic functions upon which the protocol relies. The applied pi calculus is well-suited for expressing those equivalences and those equations.

Abstractly, a *signature* Σ consists of a finite set of function symbols, such as `concat` and `sha1`, each with an integer arity. Given a signature Σ, an infinite set of names, and an infinite set of variables, the set of *terms* is defined by the grammar:

Grammar for Terms:

$T, U, V, SI, Envelope ::=$	terms
$begin, end, http, init, c, s$	name (for channels)
s_{pwd}, s_r, s_u	name (for secrets)
b, e, n, x, y, t, u	variable
$f(T_1, \ldots, T_l)$	function application

where f ranges over the function symbols of Σ and l matches the arity of f. We use metavariables v and w to range over both names and variables.

The grammar for *processes* is similar to the one in the pi calculus, except that here messages can contain terms (rather than only names) and that names need not be just channel names:

Grammar for Processes:

$P, Q, R ::=$	processes (or plain processes)
$\mathbf{0}$	null process
$P \mid Q$	parallel composition
$!P$	replication
$\nu s.P$	name restriction ("new")
$\mathit{if}\ U = V\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional
$v(x).P$	message input
$\bar{v}\langle T \rangle.P$	message output

The null process $\mathbf{0}$ does nothing; $P \mid Q$ is the parallel composition of P and Q; the replication $!P$ behaves as an infinite number of copies of P running in parallel. The process $\nu s.P$ makes a new name s then behaves as P. The conditional construct $\mathit{if}\ U = V\ \mathit{then}\ P\ \mathit{else}\ Q$ is standard, but we should stress that $U = V$ represents equality in the equational theory, rather than strict syntactic identity. We abbreviate it $\mathit{if}\ U = V\ \mathit{then}\ P$ when Q is $\mathbf{0}$. Finally, the input process $v(x).P$ is ready to input from channel v, then to run P with the actual message replaced for the formal parameter x, while the output process $\bar{v}\langle T \rangle.P$ is ready to output message T on channel v, then to run P. In both of these, we may omit P when it is $\mathbf{0}$. When X is a set of processes $\{P_i \mid i \in I\}$ indexed by some finite set $I = \{i_1, \ldots, i_n\}$, we write $\prod_{i \in I} X$ as an abbreviation for $P_{i_1} \mid \ldots \mid P_{i_n} \mid \mathbf{0}$.

Further, we extend processes with *active substitutions*:

Grammar for Extended Processes:

$A, B, C, I, K, S ::=$	extended processes
P	plain process
$A \mid B$	parallel composition
$\nu n.A$	name restriction
$\nu s.A$	variable restriction
$\{x = T\}$	active substitution

We write $\{x = T\}$ for the substitution that replaces the variable x with the term T. The substitution $\{x = T\}$ typically appears when the term T has been sent to the environment, but the environment may not have the atomic names that appear in T; the variable x is just a way to refer to T in this situation. The substitution $\{x = T\}$ is active in the sense that it "floats" and applies to any process that comes into contact with it. In order to control this contact, we may add a variable restriction: $\nu x.(\{x = T\} \mid P)$ corresponds exactly to $\mathit{let}\ x = T\ \mathit{in}\ P$. Although the substitution $\{x = T\}$ concerns only one variable, we can build bigger substitutions by parallel composition. We always assume that our substitutions are cycle-free. We also assume that, in an extended process, there is at most one substitution for each variable, and there is exactly one when the variable is restricted.

A *frame* is an extended process built up from active substitutions by parallel composition and restriction. Informally, frames represent the static knowledge gathered by the environment after communications with an extended process. An *evaluation context* $E[\text{-}]$ is an extended process with a hole in the place of an extended process. As usual, names and variables have scopes, which are delimited by restrictions and by inputs. When X is any expression, $\mathit{fv}(X)$ and $\mathit{fn}(X)$ are the sets of free variables and free names of X, respectively.

We rely on a sort system for terms and extended processes [1, Section 2]. We always assume that terms and extended processes are well-sorted and that substitutions and context applications preserve sorts.

Given a signature Σ, we equip it with an equational theory (that is, with an equivalence relation on terms with certain closure properties). We write simply $U = V$ to mean the terms U and V are related by the equational theory associated with Σ.

Structural equivalences, written $A \equiv B$, relate extended processes that are equal by any capture-avoiding rearrangements of parallel compositions, restrictions, and active substitutions, and by equational rewriting of any terms in processes.

Reductions, written $A \rightarrow B$, represent steps of computation (in particular, internal message transmissions and branching on conditionals). Reductions are closed by structural equivalence, hence by equational rewriting on terms.

Observational equivalences, written $A \approx B$, relate extended processes that cannot be distinguished by any evaluation context in the applied pi calculus, with any combination of messaging and term comparisons. (We let $\approx$ be the largest weak bisimulation on extended processes for reductions that preserves all potential observation of input or output on free names and that is closed by application of evaluation contexts [1].) The applied pi calculus has a useful, general theory—parameterized by Σ and its equational theory [1]—of observational equivalence.