Trust But Verify: Authorization for Web Services

Christian Skalka
Department of Computer Science
University of Vermont
skalka@cs.uvm.edu

X. Sean Wang
Department of Computer Science
University of Vermont
xywang@cs.uvm.edu

ABSTRACT

Through web service technology, distributed applications can be built in an open and flexible manner, bringing tremendous power to applications on the web. However, this flexibility poses significant challenges to security. Traditional access control for distributed systems is not flexible and efficient enough in such an environment; in particular, fully secure online authorization decisions may be too inefficient in practice, requiring simplifications which may have only an informal and unverifiable relation to fully secure authorization decisions.

This paper introduces a trust-but-verify framework for web services authorization. In this framework, each web service maintains the usual access control policies, as well as a "trust transformation" policy, that formally specifies how to simplify full authorization into a more efficient form for online checking. This formalization allows certainty that offline checking verifies the trust relation between full security and online checking.

Keywords

Distributed Authorization, Access Control Logic, Web Services

1. INTRODUCTION

Web services promise a new era of flexibility and power for web applications. However, this power also promises security challenges for web service providers, especially regarding access control, aka *authorization*. In particular, an end user, either human or machine, may access a web service through a number of intermediary web services, possibly formed on the fly for the task, so that access control is not so simple as validating authorization for the invoker. Additionally, due to the pluralistic and volatile nature of the web, most web services cannot expect to anticipate all access patterns, or have foreknowledge of all possible user identities, so specifying access control policies via an access control list in the usual manner is unrealistic.

Furthermore, not only does the complexity of authorization in web services pose a basic theoretical problem, but also a practical one, since robust security solutions can be too costly for online checks, especially for high-volume web services that are expected to function much like RPC/RMI for applications– that is, expeditiously.

Consider a vendor that accepts credit card payments. The vendor needs to access the card-issuing bank to check the credit card information and obtain payment confirmation; this involves a distributed authorization problem. Strictly speaking, the bank should allow a vendor to charge a card holder only when the card holder authorizes the vendor to do so. Current practice, however, is for the bank to allow vendors access to its automated services without much proof that the card holder actually initiated the transaction. The reason for this is that (1) the actual verification is costly if done for each transaction, and (2) more importantly, it's in any reputable vendor's best interest to be a good "citizen", hence problems occur only rarely. In case of disputes, the bank can investigate the involved vendors and verify any questionable charges, possibly revising its assumptions about who is to be trusted in case a cheat is discovered. The bank can also request a general audit of vendors. From a vendor perspective, appropriate records should be kept in case of a dispute or an audit.

We call this approach a *trust-but-verify (TbV)* authorization solution: in the online phase, vendors are trusted to a certain degree, but offline, the option to verify this trust can be pursued. We propose that a trust-but-verify approach is appropriate for web services security in general. Furthermore, we argue that the relation between trust and verification should be meaningful– that is, what is checked offline should *provably* verify what is implicitly trusted online. To this end, we specify a formal framework for characterizing trust-but-verify systems, that requires any implementation to provide a *trust transformation* formalizing the relation between online and offline checking. This also allows a precise characterization of auditing. Returning to the above example, we can define a trust transformation that converts the request "card holder H requests charge on his card A" into "charge card A"; thus, the transformation always takes for granted that the request originates from the cardholder. This trust transformation also serves as the starting point for auditing; verification requires the vendor to deliver proof that the card holder did in fact make the request.

We establish a formal setting for our web service authorization framework in the Calculus of Access Control [2], which we call ABLP logic following previous references in

ACM Workshop on Secure Web Services, October 29, 2004, Fairfax VA, USA.

$$\begin{array}{rcll}
A, B, C, \ldots & \in & \mathit{Atom} & \textit{atomic principals} \\
P & ::= & A \mid P \wedge P \mid P \vee P \mid P|P & \textit{principals} \\
p & \in & \mathit{Prop} & \textit{primitive propositions} \\
s & ::= & p \mid \neg s \mid s \wedge s \mid P \Rightarrow P \mid P \ \mathit{says}\ s & \textit{formulae}
\end{array}$$

Figure 1: ABLP Syntax

$$s_1 \vee s_2 \triangleq \neg(\neg s_1 \wedge \neg s_2) \qquad s_1 \supset s_2 \triangleq \neg s_1 \vee (s_1 \wedge s_2)$$

$$A \ \mathit{as}\ R \triangleq A|R \qquad A \ \mathit{controls}\ s \triangleq (A \ \mathit{says}\ S) \supset S$$

Figure 2: ABLP Formulae Abbreviations

the literature. The use of an access control logic for framework specification and system design is fundamental to our proposal, and we argue that this approach promotes unambiguous specification languages, reliability, and verifiability. Furthermore, as this presentation intends to describe and characterize our architecture, ABLP is at an appealing level of abstraction, allowing understandability and expressiveness in the high-level characterization presented here, but leaving flexibility for low-level implementation details in future work.

The contributions of this paper include an overview and formal characterization of the TbV framework, as well the definition of lower-level implementation details such as an XML wire format for ABLP assertions. By formally characterizing the system within an authorization logic, we provide a rigorous logical foundation for authorization in web service environments.

1.1 Overview of the Paper

In Sect. 2, we provide a brief review of ABLP logic. In Sect. 3, we discuss and formalize the conditions that characterize the TbV framework in ABLP. In Sect. 4, we provide TbV implementation details, by proposing an XML wire format for ABLP assertions, and by sketching an example TbV system. This example adheres to framework conditions, and shows how transactions could be authorized and audited in the system. We conclude with remarks about related and future work in Sect. 5.

2. ABLP LOGIC

In this section we give a brief review of ABLP logic, focusing on those elements of the theory that are most relevant to this paper. For a thorough description and metatheory of the logic the reader is referred to [2].

2.1 Syntax of Principals and Formulae

The syntax of ABLP principals, constituting identities in distributed communications, and formulae, representing statements and beliefs, is defined in Fig. 1. Regarding principles, we will mostly be concerned with atomic principles A and principles $P|Q$, pronounced "P quoting Q".

Statements $P \ \mathit{says}\ s$ generally represent that an assertion s has originated with a principal P. The relation $P \Rightarrow Q$, pronounced "P speaks for Q", denotes that statements uttered by P can also be attributed to Q; this is clarified in Sect. 2.3, which describes the derivation rules for the system.

2.2 Abbreviations and Conventions

We assume that all principals can perform digital signatures. We let K range over public keys as ABLP principals, and write K_A to denote the public keys of A, and K_A^{-1} the matching private key. The formula $K \ \mathit{says}\ s$ represents the formula s encrypted under K.

A number of useful abbreviations are defined in Fig. 2. Along with macros for standard logical connectives, these include $A \ \mathit{as}\ R$, denoting the principal obtained when A assumes the role R, and $A \ \mathit{controls}\ s$, denoting that A is directly authorized for s. See [2] for a complete explanation of these abbreviations.

2.3 Proof Theory

We write $\vdash s$ to denote that a formulae s is logically derivable, on the basis of the axioms and inference rules of the theory. A selection of the ABLP inference rules, connecting the calculus of principals to the underlying propositional logic, is given in Fig. 3, specifically those which will be relevant to our presentation. We note that the rule names given are of our own devise, for easy reference in the remaining presentation. Also for convenience, we write $s \vdash s'$ iff s' is derivable given assumption s.

Here is an example showing how the logic can be used to model and reason about statements signed by digital signatures associated with particular principals.

Example 2.1. *We trust that private keys remain indeed private, so that messages signed with K_J carry the authority of J:*

$$K_J \Rightarrow J$$

Thus, if any statement s is ever signed with Js private key:

$$K_J \ \mathit{says}\ s$$

By rule Speaksfor*:*

$$(K_J \Rightarrow J) \supset (K_J \ \mathit{says}\ s \supset J \ \mathit{says}\ s)$$

hence by two applications of Modus Ponens *we have:*

$$J \ \mathit{says}\ s$$

That is, any signed message can be taken as a statement of the owner of the signature key.

2.4 Access Control Lists

Access control lists (ACLs) are fundamental to access control systems, providing an explicit association of principals with the privileges for which they're authorized. In the original presentation of ABLP [2], ACLs are conjunctions of statements of the form $P \ \mathit{controls}\ s$, where s is some privilege. We adapt this approach, letting $\mathcal{A}$ range over ACLs. Furthermore, we designate a subset of Prop as the set of

TAUT
$$\frac{s \text{ is a tautology of propositional logic}}{\vdash s}$$

MODUS PONENS
$$\frac{\vdash s' \qquad \vdash s' \supset s}{\vdash s}$$

SUBTEXT
$$\vdash A \textit{ says } (s \supset s') \supset (A \textit{ says } s \supset A \textit{ says } s')$$

PARROT
$$\frac{\vdash s}{\vdash A \textit{ says } s}$$

QUOTE
$$\vdash (B|A) \textit{ says } s \equiv B \textit{ says } A \textit{ says } s$$

SPEAKSFOR
$$\vdash (A \Rightarrow B) \supset ((A \textit{ says } s) \supset (B \textit{ says } s))$$

Figure 3: Selected ABLP Inference Rules

privileges in the system, letting **priv** range over this set. Hence, ACLs are conjunctions of statements P *controls* **priv**. Our justification for designating atomic propositions as privileges, and our use thereof, is discussed in Sect. 3.1 in more detail.

3. FRAMEWORK DEFINITIONS

Prior to the specifics of system design, the trust-but-verify framework can be precisely characterized as a set of conditions that any implementation must satisfy. In this section, we motivate and describe these conditions, which are stated at a sufficiently high level to allow flexibility in lower-level system design, but are mathematically rigorous.

3.1 Authorization Contexts and Decisions

Web services authorization is based on requests for the service made by invokers. Here we describe our proposed structure for these requests, and for the authorization decision predicated on them.

A *request* is an ABLP assertion s uttered by the invoker of a web service, which the invoker intends to be used by the web service for authorization of its use. In addition to the request, a web service may possess other facts and beliefs, e.g. ACLs and role certifications, that affect the authorization judgement; we assume that these facts and beliefs are expressed as ABLP formulae. The conjunction of these components constitutes an authorization *context*; authorization for a web service is granted upon a particular invocation iff the context of the invocation allows the privilege required for use of the web service to be derived in the ABLP proof theory.

As mentioned in Sect. 2, we posit a set of atomic formulae **priv**, each of which represent the privilege required to access a particular web service. Authorization for **priv** in a context s is effected by checking validity of $s \vdash \textbf{priv}$. Thus, our system is inspired by access control mechanisms such as stack inspection [20], which are specialized for program procedure calls, rather than challenge/response systems such as [6], which are adapted to human usage (i.e. web browsing) patterns. Our justification for this is that web service invocation bears a strong similarity to RPC/RMI, as observed in [14, 7], with chains of web service invocations resembling call stacks.

Here is a brief example illustrating the concepts of the framework described thus far:

EXAMPLE 3.1. Suppose some web service WS requires the privilege **priv** to be used, and the web service ACL $\mathcal{A}$ grants this privilege to a principal D, i.e. $\mathcal{A} \equiv \mathcal{A}' \wedge D$ *controls* **priv** for some $\mathcal{A}'$. Suppose also that D invokes WS on its own behalf, making the request D *says* **priv**. Thus, the authorization context is D *says* $\textbf{priv} \wedge \mathcal{A}$. Clearly, D *says* $\textbf{priv} \wedge \mathcal{A} \vdash \textbf{priv}$, since the context implies both $\vdash D$ *says* **priv** and $\vdash D$ *controls* **priv**, which implies $\vdash \textbf{priv}$ by MODUS PONENS.

Naturally, it is desirable for authorization judgements to be decidable. Although ABLP logic is undecidable in general, various presentations have described non-trivial, decidable access control mechanisms [19, 6, 2]. Therefore, it is realistic to make this a formal condition of the trust-but-verify framework:

CONDITION 1. *Let s be an authorization context; then validity of $s \vdash$ **priv** is decidable.*

This condition requires any trust-but-verify implementation to provide a decision procedure for validity of authorization judgements, and also implicitly requires the form of authorization contexts to be well-defined. As with all the framework conditions, the formal statement of the condition allows correctness of a decision procedure to be provable, i.e. implementations can (and should) be accompanied with proofs of their adherence to framework conditions.

3.2 Trust Transformations

The distinguishing characteristic of our proposal is the separation of online and offline checking phases, where in the online phase certain elements of authorization are taken for granted, or trusted to hold. This yields a simpler authorization decision, which can be verified more rigorously during the offline phase. However, with security at stake, vague accounts of the relation between these phases does not suffice– rather, we desire a formal relationship, so that offline *verification* of online authorization is meaningful. We embody this notion in the trust transformation, which specifies what elements of online authorization are to be taken for granted, by specifying how to transform untrusted requests into trusted ones.

A *trust transformation* is a function from ABLP formulae to ABLP formulae. Any trust transformation's domain is formulae in *extrapolated* form, which take into account all components of access control, down to every detail verified during offline checking. The range of any trust transformation is formulae in *trusted* form, which are the "watered down" formulae that exclude restrictions the system takes for granted during online checking. The trust transformation mapping rigorously defines the relation between extrapolated and trusted forms. Since notions of trust can vary depending on the system, we specify the type and necessary preconditions of trust transformations, but the definition of the function itself is left up to a particular system designer. For any extrapolated formula s, we denote its trust transformation as $[\![s]\!]$. Any trust-but-verify implementation must define extrapolated and trusted authorization forms, and the trust transformation between them, with the requirement that it be a total function on the set of extrapolated statements. Also, we specify that the authorization contexts mentioned in Condition 1 are in trusted form.

EXAMPLE 3.2. Suppose access control for a web service WS is based on requests made in both signed and unsigned form, so that all requests are of the form:

$$B\ \mathit{says}\ s \wedge K_B\ \mathit{says}\ s$$

Suppose further that in the online component, players are trusted to communicate messages faithfully, and signatures are not checked. All authorization contexts include an ACL $\mathcal{A}$, which is left unchanged by the trust transformation. Thus, for all B and s, the trust transformation is defined as:

$$[\![B\ \mathit{says}\ s \wedge K_B\ \mathit{says}\ s \wedge \mathcal{A}]\!] = B\ \mathit{says}\ s \wedge \mathcal{A}$$

Note that this transformation is total for the extrapolated form of requests in this example.

3.3 Auditing

While online checking takes trust into account, the purpose of offline checking is to verify that this trust is warranted. As the trust transformation injects trust into authorization, offline checking inverts the trust transformation, to verify online trust in the offline phase– that is, given a trusted request s, offline checking searches for an extrapolated request of which s is the trust transform. We call this process *auditing*, and require that any trust-but-verify implementation provide a function audit that given any trust-transformed context s, retrieves its extrapolated form.

The details of auditing constitute a significant engineering element of any trust-but-verify system. As will be exemplified in Sect. 4, we expect that certain elements of extrapolated statements will be logged for later offline retrieval during auditing, but what elements, and how and where they are logged and retrieve, is at a much lower level of detail than we're concerned with here. But at the abstract level we are concerned with, we can characterize the formal requirements of auditing.

Firstly, as mentioned above, audit returns the extrapolated form of trusted authorization contexts. Since the trust transformation has been defined formally, we can precisely characterize this condition as follows:

CONDITION 2. *Let s be a trusted context; then if audit(s) succeeds, audit$(s) \vdash s'$ such that $[\![s']\!] = s$.*

Note that this condition allows a certain degree of flexibility, in that audit must return a statement that is *at least* as strong as an extrapolated form of the input, not necessarily an extrapolated form per se.

Furthermore, we say *an* extrapolated form, since it is possible that any given trust transformation is many-to-one. Significantly, it is not even necessary that an extrapolated form of an authorized statement be authorized. However, since auditing seeks to verify trust implicit in an online check, we require that auditing not only return an extrapolated form of input statements, but one that is also authorized for the privilege in question; otherwise, auditing fails. This motivates the third condition of our framework:

CONDITION 3. *Let s be a trusted context and* **priv** *be a privilege. If $s \vdash$* **priv** *holds, then so does audit$(s) \vdash$* **priv**.

It is important to note that this condition does not necessarily require theorem proving on extrapolated forms, but rather provability should follow by adherence to this condition generally. This point is revisited in more detail in Sect. 4.2.6. Here is an example that illustrates an auditing technique satisfying the specified conditions:

EXAMPLE 3.3. Given both the online and offline checking scheme, as well as the trust transformation defined in Example 3.2, we define audit as follows. First, we assume that while trust transformations discard the signed portion of requests for online checking, the signed portion is actually saved (logged) as part of the implementation. Auditing of any request $B\ \mathit{says}\ \mathbf{priv}$ will then retrieve the signed portion of the original request discarded by the trust transformation, and verify that it is of the form $K_B\ \mathit{says}\ \mathbf{priv}$ by decrypting it, yielding the extrapolated context $B\ \mathit{says}\ \mathbf{priv} \wedge K_B\ \mathit{says}\ \mathbf{priv} \wedge \mathcal{A}$ as the result of auditing.

Clearly, this example is vague with regard to how signed portions of requests are saved and retrieved. While these details are naturally addressed in implementations, we revisit this issue with some general suggestions for implementations in Sect. 4.

4. IMPLEMENTATION

In this section we explore implementation details of a particular instance of the TbV framework, based on ABLP. We first define an XML wire format for ABLP assertions, that can be integrated into SOAP messages [8] for standardized Web Service communication. After that, we investigate an example TbV architecture and policy setting.

4.1 XML Wire Format

Web service communication is based on SOAP messages [8], which are XML documents containing a Body element and possibly a Header element. The Body contains request information– i.e. the name of the service being invoked, and values provided as parameters– or response information– i.e. the result of successful service invocation, or failure information. The Header contains security and routing information. To communicate authorization assertions between intermediaries, an AuthInfo element can be added as a Security item, so Headers have the form:

```
<Header>
   <Security>
      <AuthInfo>...</AuthInfo>,
      ...
   </Security>
   ...
</Header>
```

The encoding of ABLP terms in XML for implementation is complicated by the representation of signed expressions, i.e. expressions of the form $K\ \mathit{says}\ s$; in the implementation, these should actually be signed. Therefore, it is necessary to integrate some cryptographic technique into the representation. Furthermore, since we assume a Web Service environment where interacting parties are possibly unknown to each other, it is unrealistic to rely on techniques that require shared secrets. Therefore, we propose to use a public-key signing technique based on x509 certificates.

In more detail, we use a subset of the syntax developed by Bhargavan et al. [7], specified in Fig. 6. This syntax extends

Tag	::=	`any legal XML name`	*XML name*
a : att	::=	Tag="str"	*attribute*
as : atts	::=	$a\ as \mid \epsilon$	*attribute sequence*
i : item	::=	$Elem \mid str$	*item*
is : items	::=	$i\ is \mid \epsilon$	*item sequence*
$Elem$	::=	$<Tag\ as>is</Tag>$	*element*
str : string	::=	`any legal XML string`	*XML string*
		$\mathbf{base64}(x : \text{bytes})$	*Base64-encoding of byte array*
x : bytes	::=	$\mathbf{c14n}(i : \text{item})$	*canonical bytes of an item*
		$\mathbf{rsa.sha1}(x, k : \text{bytes})$	*public key signature*
		$\mathbf{x509}(k : \text{bytes}, u : \text{string}, a : \text{string}, k : \text{bytes})$	*X.509 certificate*

Figure 4: XML Data Model, with Byte Arrays and Symbolic Cryptography

$$x_1 \ldots x_n \triangleq x_1, (\ldots(x_n\epsilon)) \qquad \text{for } n > 0, x \in \text{atts, items}$$

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

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

Figure 5: XML Abbreviations

α	::=	$<$`Prin`$>str</>$	*atomic principals*
κ	::=	$<$`Key`$>str</>$	*keys-as-principals*
ρ	::=	$\alpha \mid \kappa$	*principals*
φ	::=	$<$`Conj`$>\varphi_1 \ldots \varphi_n</>$	*formulae*
		$<$`Prop`$>str</>$	
		$<$`Says`$>\alpha, \varphi</>$	
		$<$`Says`$>\varsigma, \varphi</>$	
		$<$`Speaksfor`$>\rho, \rho</>$	
ς	::=	$<$`CertSig`$>str, str</>$	*signed formulae*

Figure 6: ABLP Wire Format Syntax

a basic XML format with byte arrays. Strings may be of the format $\mathbf{base64}(x)$, which is the base64 encoding of byte array x. Byte arrays may either be of the form $\mathbf{c14n}(i)$, which is a standard canonicalization of item i [9], or they may be of the form $\mathbf{rsa.sha1}(x, k)$ which is the public-key encoding of x under key k, or they may be an x509 certificate. The latter are of the form $\mathbf{x509}(k_1, u, a, k_2)$, where k_1 is the signing key of a certification authority, and u,a, and k_2 are the signed user name, signing algorithm, and user's public key. We assume the existence of an agreed upon certification authority CA, who issues only one public/private key pair per user. For any given u, we denote their public key pk_u, and their private key sk_u.

To symbolize decidable deconstruction of x509 certificates and public-key decryption, we posit several functions. The function:

$$\mathbf{check.rsa.sha1}(x_1, x_2, pk_u : \text{bytes})$$

checks that x_1 is the signature of x_2 under sk_u. The function:

$$\mathbf{check.x509}(cert, pk_r : \text{bytes})$$

checks that $cert$ is an x509 certificate signed by the private key of a certification authority r. The functions:

$$\begin{array}{c} \mathbf{x509.alg}(cert : \text{bytes}) \\ \mathbf{x509.key}(cert : \text{bytes}) \\ \mathbf{x509.user}(cert : \text{bytes}) \end{array}$$

return the signing algorithm, public key, and user in the certificate $cert$. Formally, these functions satisfy the following equations:

$$\mathbf{x509.alg}(\mathbf{x509}(k_1, u, a, k_2)) = a$$

$$\mathbf{x509.key}(\mathbf{x509}(k_1, u, a, k_2)) = k_2$$

$$\mathbf{x509.user}(\mathbf{x509}(k_1, u, a, k_2)) = u$$

$$\mathbf{check.x509}(\mathbf{x509}(sk_r, u, a, k_2), pk_r) = pk_r$$

$$\mathbf{check.rsa.sha1}(x, \mathbf{rsa.sha1}(x, sk_u), pk_u) = pk_u$$

The XML representation of ABLP terms we give here will not be complete, but rather based on the terms of interest in our proposed implementation. In particular, we will only be concerned with atomic principles A and keys-as-principles K, and formulae will not involve negation. The XML grammar for encoding ABLP terms of interest is given in Fig. 6; for brevity in this figure and elsewhere, abbreviations are defined in Fig. 5, where closing delimiters can either be shortened to omit the matching opening delimiter name, or omitted entirely in case delimitation is denoted by indentation.

The most interesting components of our wire format representation of ABLP terms is the representation of keys-as-principles, which are denoted by x509 certificates, and signed assertions K *says* s. The latter are represented using not only an x509 certified public key, but also a signature of the relevant assertion under the corresponding private key. Thus, keys literally sign statements ascribed to them. The encoding is fully formalized by a mapping from XML items i to ABLP principals and formulae, written $(\!| i |\!)$ and defined in Fig. 7. This definition assumes that any given principal A and primitive proposition p have canonical string representations, denoted $\hat{A}$ and $\hat{p}$ respectively.

$$
\begin{aligned}
\llparenthesis \texttt{<Prin>}\hat{A}\texttt{</>} \rrparenthesis &= A \\
\llparenthesis \texttt{<Key>}\mathit{str}\texttt{</>} \rrparenthesis &= K_A \qquad \text{if } \mathit{certname}(\mathit{str}) = \hat{A} \\[6pt]
\llparenthesis \texttt{<Prop>}\hat{p}\texttt{</>} \rrparenthesis &= p \\
\llparenthesis \texttt{<Conj>}\varphi_1 \ldots \varphi_n\texttt{</>} \rrparenthesis &= \llparenthesis \varphi_1 \rrparenthesis \wedge \cdots \wedge \llparenthesis \varphi_n \rrparenthesis \\
\llparenthesis \texttt{<SpeaksFor>}\rho_1, \rho_2\texttt{</>} \rrparenthesis &= \llparenthesis \rho_1 \rrparenthesis \Rightarrow \llparenthesis \rho_2 \rrparenthesis \\
\llparenthesis \texttt{<Says>}\alpha, \varphi\texttt{</>} \rrparenthesis &= \llparenthesis \alpha \rrparenthesis \; \mathit{says} \; \llparenthesis \varphi \rrparenthesis \\
\llparenthesis \texttt{<Says>}\varsigma, \varphi\texttt{</>} \rrparenthesis &= K_A \; \mathit{says} \; \llparenthesis \varphi \rrparenthesis \qquad \text{if } \mathit{signedby}(\varsigma, \varphi) = \hat{A}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{certname}(\mathit{str}) \;=\; & \text{if } \mathit{str} = \mathbf{base64}(\mathit{cert}) \\
& \quad \text{where } \mathbf{check.x509}(\mathit{cert}, \mathit{pk}_{CA}) = \mathit{pk}_{CA} \\
& \text{then return } \mathbf{x509.user}(\mathit{cert}) \\
& \text{else fail}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{signedby}(\varsigma, \varphi) \;=\; & \text{if } \varsigma = \texttt{<CertSig>}\mathbf{base64}(\mathit{cert}), \mathbf{base64}(\mathit{sig})\texttt{</>} \\
& \quad \text{where } \mathbf{check.x509}(\mathit{cert}, \mathit{pk}_{CA}) = \mathit{pk}_{CA} \\
& \qquad \text{and } \mathbf{x509.alg}(\mathit{cert}) = \texttt{rsh-sha1} \\
& \qquad \text{and } \mathbf{x509.key}(\mathit{cert}) = k \\
& \qquad \text{and } \mathbf{check.rsa.sha1}(\mathbf{c14n}(\varphi), \mathit{sig}, k) = k \\
& \text{then return } \mathbf{x509.user}(\mathit{cert}) \\
& \text{else fail}
\end{aligned}
$$

Figure 7: XML to ABLP Transformation

A number of example assertions in this wire format are presented and discussed in the next section.

4.2 Policies and Architecture

We now describe a sample TbV authorization architecture, including (some) implementation details and general authorization policies. While this is mainly intended to clarify the proposals of the previous sections, we make some substantive suggestions and observations for TbV implementations in general. We also discuss a running example transaction within this architecture. While most of the discussion is based in concrete ABLP syntax for brevity and clarity in specification, we also provide XML wire formats of ABLP assertions in the example transaction.

4.2.1 Individuals

Many concrete entities take part in web service transactions; machines, users, applications, web services, domains, etc. A complete treatment would consider all possible players, since for example the same application run by the same user on two different machines might inspire different levels of trust, depending on the status of the machines. However, for the purposes of simplicity in this presentation, we will assume the existence of only two sorts of concrete entities in web service transactions, *users* and *web services*, termed *individuals* taken together.

We also assume given finite, disjoint set of available principal identifiers for user and web services, respectively, and let J range over the former, and WS the latter. Furthermore, we will assume that these names have some known and decidable format, allowing automatic determination of whether a given individual is a user or a web service.

Example 4.1. We posit the following individuals: Joe is a user, and WS_M is an intermediary web service providing service to users, part of a system that includes access to a centralized medical database web service WS_{MDB} (to be continued...)

4.2.2 Roles

As in many other systems, *roles* are an important component of our authorization scheme. In particular, due to the inherently volatile and popular nature of the web, web services cannot in general be expected to know the names of every possible invoker *a priori*; thus, authorization for privileges will be granted to known roles, and individuals must prove that they may assume claimed roles.

Furthermore, the same characteristics that inspire the use of roles for authorization, imply that role membership should not be established via explicit role membership lists. Rather, we posit that every known role R is associated with a key K_R, and the ability to sign messages with K_R (that is, proof of possession of the associated private key) is sufficient to establish role membership. Thus, we introduce the axiom:

$$
\begin{array}{l}
\textsc{RoleKey} \\
\vdash K_R \; \mathit{controls} \; (P \Rightarrow R)
\end{array}
$$

So for example, if an individual A wishes to legitimately assume a role R, it could make the statement K_R *says* $(A \Rightarrow R)$; this and the assumed authority of role keys imply $A \Rightarrow R$.

Membership in roles is established by role certificates, the grammar for which is given in Fig. 8. They are conjunctions of role certifications, with **true** the empty certificate.

Example 4.1. *(Continued)* In our transaction example, we assume that Joe can take on the role of a doctor (denoted D). This means that Joe is capable of signing his requests using K_D. We assume a "trusted medical web services" role, denoted M, and that WS_M can take on role M. We also assume that the web service WS_{MDB} will authorize doctors

$$\begin{array}{rcll} \mathcal{R} & ::= & K_R \textit{ says } (A \Rightarrow R) \mid \mathcal{R} \wedge \mathcal{R} & \textit{role certificates} \\ \mathbf{req} & ::= & K_{\mathrm{WS}} \textit{ says } R \textit{ says } \mathbf{req} \mid K_J \textit{ says } R \textit{ says } \mathbf{priv} & \textit{extrapolated requests} \\ \hat{s} & ::= & K_{\mathrm{WS}} \textit{ says } R \textit{ says } \iota \mid K_J \textit{ says } R \textit{ says } \mathbf{priv} \mid \hat{s} \wedge \mathcal{R} & \textit{indexed statements} \end{array}$$

Figure 8: Components of Authorization Requests

to use WS_{MDB}. Hence, we have:

$$D \textit{ controls } \mathbf{priv}_{MDB}$$

is in $\mathcal{A}_{MDB}$. Note that we *do not* assume role M has access to WS_{MDB}. Role M will be used to "carry" doctor's requests, as defined below.

4.2.3 Carrier Authority

In any access control statement P *controls* **priv**, it is not necessary for P to be atomic, allowing a fine-grained approach to access control. For example, if it is not desirable to grant a role R direct access to **priv**, but only on behalf of a principal D, the ACL can specify $R|D$ *controls* **priv**, disallowing R direct access to **priv**.

However, we're concerned with access control decisions in the presence of multiple intermediaries– that is, several intervening nodes may transport an authorization request from source to target. The above scheme would require separate entries for every possible chain of intermediaries; for example, given statements:

$$R_1 \textit{ says } R_2 \textit{ says } D \textit{ says } \mathbf{priv}$$
$$R_2 \textit{ says } R_1 \textit{ says } D \textit{ says } \mathbf{priv}$$

authorization for both would require both of the following statements to be present in the relevant ACL:

$$R_1|R_2|D \textit{ controls } \mathbf{priv}$$
$$R_2|R_1|D \textit{ controls } \mathbf{priv}$$

Either that, or it would require access statements:

$$R_1|D \textit{ controls } \mathbf{priv}$$
$$R_2|D \textit{ controls } \mathbf{priv}$$

to be present, along with known relations $R_1 \Rightarrow R_2$ and $R_2 \Rightarrow R_1$. The former solution is clearly cumbersome, unrealistically so given the number of possible intermediaries on the web. The latter is better, but is restrictive, requiring every intermediary to adopt the same role as, or a more powerful role than, its predecessor (although this problem could be alleviated by adapting the "speaks for regarding" relation proposed in [10] as an extension to ABLP). Since the only privilege at issue is **priv**, it is intuitively sufficient for each intermediary to have some sort of authorization for **priv**, as in e.g. stack inspection [20].

However, unlike stack inspection, web service intermediaries should often not be granted direct access to privileges– for example, a web service should not be granted direct access to withdraw cash from an individual's bank account, but only on behalf of that individual. To maintain this property, and to overcome the drawbacks of the approaches described in the previous paragraph, we introduce the notion of a *carrier authority.* Intuitively, a carrier authority is the authority to carry an authorization request for a particular principal, but not the authority to make the request itself. Formally:

$$R \textit{ carries } \mathbf{priv} \textit{ for } D \triangleq (R|D \textit{ says } \mathbf{priv}) \supset (D \textit{ says } \mathbf{priv})$$

In the above example, access requires the carrier authorities R_1 *carries* **priv** *for* D and R_2 *carries* **priv** *for* D, as well as the direct authority D *controls* **priv**. In general, carrier authority allows a fine-grained and flexible approach to authorization in the context of web services. We call conjunction of carrier authority statements *carrier control lists* (CCLs), which we denote $\mathcal{C}$.

EXAMPLE 4.1. *(Continued)* In our transaction example, M is not directly authorized for $\mathbf{priv}_{MDB}$, but it should be possible for trusted medical web services to carry this privilege for doctors, hence WS_{MDB} defines a carrier control list $\mathcal{C}_{MDB}$ that includes:

$$M \textit{ carries } \mathbf{priv}_{MDB} \textit{ for } D$$

4.2.4 Extrapolated Statements

At a high level, we assume that extrapolated contexts possess the following characteristics:

1. All requests are made by individuals in a particular role.

2. All requests are signed by the requesting individual, to establish its authenticity (both on and offline).

3. All requests are accompanied by role certificates, to establish the relevant individual's role membership.

Characteristics (1) and (2) together determine the form of extrapolated statements specified in Fig. 8. Characteristic (3) implies that extrapolated authorization contexts will include role certificates, along with requests, ACLs, and CCLs. Hence, extrapolated authorization contexts are statements of the form:

$$\mathbf{req} \wedge \mathcal{R} \wedge \mathcal{C} \wedge \mathcal{A}$$

EXAMPLE 4.1. *(Continued)* Joe, a doctor, wishes to use a medical web service WS_M to diagnose a patient, which in turn invokes WS_{MDB} for the patient's medical history. To initialize the appropriate role memberships and authorizations, Joe's request, in extrapolated form, is:

$$K_J \textit{ says } D \textit{ says } \mathbf{priv}_{MDB} \wedge K_D \textit{ says } J \Rightarrow D$$

This means that Joe (J), speaking as a doctor, wants to access $\mathbf{priv}_{MDB}$, and Joe (J) establishes that he can take the role of doctor (D).

This request can be represented in XML wire format as follows. Assuming that `Joe`, `Doctor`, and `priv(MDB)` are $\hat{J}$,

$$\begin{array}{rcl} [\![\mathbf{req} \wedge \mathcal{R} \wedge \mathcal{C} \wedge \mathcal{A}]\!] & = & [\![\mathbf{req}]\!] \wedge \mathcal{C} \wedge \mathcal{A} \\ [\![K_J \ \textit{says}\ R \ \textit{says}\ \mathbf{priv}]\!] & = & R \ \textit{says}\ \mathbf{priv} \\ [\![K_{\mathrm{WS}} \ \textit{says}\ R \ \textit{says}\ \mathbf{req}]\!] & = & R \ \textit{says}\ [\![\mathbf{req}]\!] \end{array}$$

Figure 9: A Trust Transformation

$\hat{D}$, and $\widehat{\mathbf{priv}_{MDB}}$ respectively, let:

$$\begin{array}{lll} \varphi_1 & \triangleq & <\texttt{Says}> \\ & & \quad <\texttt{Prin}>\texttt{Doctor}</> \\ & & \quad <\texttt{Prop}>\texttt{priv(MDB)}</> \\ & & \\ \varphi_2 & \triangleq & <\texttt{Speaksfor}> \\ & & \quad <\texttt{Prin}>\texttt{Joe}</> \\ & & \quad <\texttt{Prin}>\texttt{Doctor}</> \end{array}$$

and observe $\llparenthesis \varphi_1 \rrparenthesis = D \ \textit{says}\ \mathbf{priv}_{MDB}$ and $\llparenthesis \varphi_2 \rrparenthesis = J \Rightarrow D$. Let:

$$\begin{array}{lll} \textit{cert}_1 & \triangleq & \mathbf{x509}(sk_{CA}, \texttt{Joe}, \texttt{rsa-sha1}, pk_{\texttt{Joe}}) \\ \textit{cert}_2 & \triangleq & \mathbf{x509}(sk_{CA}, \texttt{Doctor}, \texttt{rsa-sha1}, pk_{\texttt{Doctor}}) \end{array}$$

which are the x509-certified public keys of `Joe` and `Doctor`, and let:

$$\begin{array}{lll} \textit{sig}_1 & \triangleq & \mathbf{rsa.sha1}(\mathbf{c14n}(\varphi_1), sk_{\texttt{Joe}}) \\ \textit{sig}_2 & \triangleq & \mathbf{rsa.sha1}(\mathbf{c14n}(\varphi_2), sk_{\texttt{Doctor}}) \end{array}$$

Then, the following item can be included as the `AuthInfo` in the `Header` of Joe's SOAP request message to WS_M; the request component `Request` is explicitly separated from the role certifications component `RoleCerts` of the assertion, for ease of processing:

$$\begin{array}{l} <\texttt{Request}> \\ \quad <\texttt{Says}> \\ \quad\quad <\texttt{CertSig}>\mathbf{base64}(\textit{cert}_1), \mathbf{base64}(\textit{sig}_1)</> \\ \quad\quad \varphi_1 \\ <\texttt{RoleCerts}> \\ \quad <\texttt{Says}> \\ \quad\quad <\texttt{CertSig}>\mathbf{base64}(\textit{cert}_2), \mathbf{base64}(\textit{sig}_2)</> \\ \quad\quad \varphi_2 \end{array}$$

4.2.5 Trust Transformation

The extrapolated context described above uses encryption to determine authenticity of statements, but at significant cost. Messaging can become quite complex, even in our simplified model, motivating a more efficient online checking technique, wherein many elements of extrapolated checking are "trusted away". Thus, we make the following the following simplifications for more efficient online checking:

1. Individuals are trusted to make valid claims about role membership.

2. Authenticity of requests is assumed for any intermediary; for example, if the request R_1 *says* R_2 *says* s is received, then we trust that R_1 truly said "R_2 *says* s" and R_2 truly said "s".

In practice, these assumptions are clearly too simplistic, in that they allow any web service invoker to assume any role membership. Some justification for that claim should be provided, e.g. a role key signature on the "top-level" of the request, so that instead of R_1 *says* R_2 *says* s, the signed request K_{R_1} *says* R_2 *says* s would be communicated. However, for the purposes of this example we will set this issue aside.

We formalize these trust assumptions via the trust transformation defined in Fig. 9. Note that the transformation eliminates role certifications and private key signatures for individuals, leaving just trusted role statements. We assert that authorization for trusted contexts is decidable (and efficient), satisfying Condition 1:

LEMMA 1. *Let s be an extrapolated context; then for all* $\mathbf{priv}$, *validity of* $[\![s]\!] \vdash \mathbf{priv}$ *is decidable.*

The result follows thanks to the similarity of the current authorization scheme with stack inspection; a decision procedure is easily defined as a modification of that given in [18].

EXAMPLE 4.1. *(Continued)* Let:

$$\begin{array}{lll} s_1 & \triangleq & K_J \ \textit{says}\ D \ \textit{says}\ \mathbf{priv}_{MDB} \\ rc_1 & \triangleq & K_D \ \textit{says}\ J \Rightarrow D \end{array}$$

When WS invokes the service WS_{MDB} on Joe's behalf, WS's extrapolated authorization assertion will state that WS invokes WS_{MDB} in the role M, and will include all relevant role certifications, i.e. it will be of the form $s_2 \wedge rc_1 \wedge rc_2$, where:

$$\begin{array}{lll} s_2 & \triangleq & K_{\mathrm{WS}_M} \ \textit{says}\ M \ \textit{says}\ s_1 \\ rc_2 & \triangleq & K_M \ \textit{says}\ \mathrm{WS}_M \Rightarrow M \end{array}$$

By the trust transformation rules in Fig. 9, we have:

$$[\![s_2 \wedge rc_1 \wedge rc_2]\!] = M \ \textit{says}\ D \ \textit{says}\ \mathbf{priv}_{MDB}$$

and WS_{MDB} will base its (successful) online authorization decision upon this assertion.

The assertion $[\![s_2 \wedge rc_1 \wedge rc_2]\!]$ can be obtained by WS_{MDB} in a variety of ways. For example, all certificates and extrapolated assertions can be propagated from Joe to WS_M to WS_{MDB}, and WS_{MDB} itself can perform the trust transformation. The implementation of this trust transformation on wire format assertions is defined in Fig. 10; note in particular that the function does not perform any cryptographic procedures in the transformation. We may formally assert correctness of this implementation of the trust transformation specified in Fig. 9 as follows:

LEMMA 2. *For any φ_1 and φ_2, we have* $[\![\llparenthesis \varphi_1 \rrparenthesis \wedge \llparenthesis \varphi_2 \rrparenthesis]\!] = \llparenthesis [\![<\texttt{Request}>\varphi_1</><\texttt{RoleCerts}>\varphi_2</>]\!] \rrparenthesis$.

The pre-transformation assertion communicated from WS_M to WS_{MDB} can then be defined as follows. Assuming $\hat{M} =$

$$
\begin{array}{rcl}
\llbracket \texttt{<Request>}\varphi\texttt{</><RoleCerts>}\varphi'\texttt{</>}\rrbracket & = & \llbracket\varphi\rrbracket \\
\\
\llbracket \texttt{<Prin>}\hat{\mathbf{priv}}\texttt{</>}\rrbracket & = & \texttt{<Prin>}\hat{\mathbf{priv}}\texttt{</>} \\
\llbracket \texttt{<Says>}\varsigma,\varphi\texttt{</>}\rrbracket & = & \llbracket\varphi\rrbracket \\
\llbracket \texttt{<Says>}\alpha,\varphi\texttt{</>}\rrbracket & = & \texttt{<Says>}\alpha,\llbracket\varphi\rrbracket\texttt{</>}
\end{array}
$$

Figure 10: Wire Format Trust Transformation

MedServ and $\hat{\mathrm{WS}}_M = \texttt{WS(M)}$, let:

$$
\begin{array}{lll}
\varphi_3 & \triangleq & \texttt{<Says>} \\
 & & \quad \texttt{<Prin>MedServ</>} \\
 & & \quad \texttt{<Says>} \\
 & & \quad\quad \texttt{<CertSig>} \\
 & & \quad\quad\quad \mathbf{base64}(\mathit{cert_1}) \\
 & & \quad\quad\quad \mathbf{base64}(\mathit{sig_1}) \\
 & & \quad\quad \varphi_1 \\
 \\
\varphi_2 & \triangleq & \texttt{<Speaksfor>} \\
 & & \quad \texttt{<Prin>WS(M)</>} \\
 & & \quad \texttt{<Prin>MedServ</>}
\end{array}
$$

and observe $(\!| \varphi_3 |\!) = M\ \mathit{says}\ K_J\ \mathit{says}\ D\ \mathit{says}\ \mathbf{priv}_{MDB}$ and $(\!| \varphi_4 |\!) = \mathrm{WS}_M \Rightarrow M$. Let:

$$
\begin{array}{lll}
\mathit{cert_3} & \triangleq & \mathbf{x509}(sk_{CA}, \texttt{WS(M)}, \texttt{rsa-sha1}, pk_{\texttt{WS(M)}}) \\
\mathit{cert_4} & \triangleq & \mathbf{x509}(sk_{CA}, \texttt{MedServ}, \texttt{rsa-sha1}, pk_{\texttt{MedServ}})
\end{array}
$$

which are the x509-certified public keys of WS(M) and MerServ, and let:

$$
\begin{array}{lll}
\mathit{sig_3} & \triangleq & \mathbf{rsa.sha1}(\mathbf{c14n}(\varphi_3), sk_{\texttt{WS(M)}}) \\
\mathit{sig_4} & \triangleq & \mathbf{rsa.sha1}(\mathbf{c14n}(\varphi_4), sk_{\texttt{MedServ}})
\end{array}
$$

Then the item i_{auth}, defined as follows, can be included as the AuthInfo in the Header of WS_Ms SOAP message request to WS_{MDB}, when the former invokes the latter on behalf of Joe:

$$
\begin{array}{l}
\texttt{<Request>} \\
\quad \texttt{<Says>} \\
\quad\quad \texttt{<CertSig>}\mathbf{base64}(\mathit{cert_3}), \mathbf{base64}(\mathit{sig_3})\texttt{</>} \\
\quad\quad \varphi_3 \\
\texttt{<RoleCerts>} \\
\quad \texttt{<Conj>} \\
\quad\quad \texttt{<Says>} \\
\quad\quad\quad \texttt{<CertSig>}\mathbf{base64}(\mathit{cert_2}), \mathbf{base64}(\mathit{sig_2})\texttt{</>} \\
\quad\quad\quad \varphi_2 \\
\quad\quad \texttt{<Says>} \\
\quad\quad\quad \texttt{<CertSig>}\mathbf{base64}(\mathit{cert_4}), \mathbf{base64}(\mathit{sig_4})\texttt{</>} \\
\quad\quad\quad \varphi_4
\end{array}
$$

Observe that $\llbracket i_{auth} \rrbracket$ is equivalent to the following:

$$
\begin{array}{l}
\texttt{<Request>} \\
\quad \texttt{<Says><Prin>MedServ</>}\varphi_1\texttt{</>} \\
\texttt{<RoleCerts></>}
\end{array}
$$

and:

$$(\!| \llbracket i_{auth} \rrbracket |\!) = M\ \mathit{says}\ D\ \mathit{says}\ \mathbf{priv}_{MDB}$$

Alternatively, trust transformation can be performed eagerly, eliminating the need for Joe and WS_M to sign authorization information for online transactions. In this case, the wire formatted assertion $\llbracket i_{auth} \rrbracket$ would be communicated from WS_M to WS_{MDB}.

4.2.6 Logging and Auditing

For auditing, it is necessary to reconstruct the signed statements and role certificates of extrapolated forms. In general, we imagine that auditing will be driven by the logging of signed statements and certificates in the online stage; although they're not used in online checking, they're saved and retrieved for offline verification. The details constitute a significant engineering problem and are beyond the scope of this paper.

Firstly, assuming that signed statements and signatures are logged, *who* exactly does the logging is another question. For example, everyone could be required to log their own statements, or the request statements they receive, or the source or target machine could be required to log all relevant statements, or perhaps a distinguished machine could be established as a log server, etc. We propose a model wherein these details are abstract; we posit *log locations* ι, and a lookup function that, given a location ι, returns the statements and certificates at that location. The concrete form and definition of locations ι and lookup, as well as logging conventions, determine the particulars for a given implementation.

Furthermore, noting that requests are single statements involving possibly multiple intermediaries, to obtain maximal flexibility in logging we would like the ability to "break up" statements into those parts made by each intermediary, in case each is independently responsible for logging. Thus, rather than statements such as $K_{\mathrm{WS}}\ \mathit{says}\ R\ \mathit{says}\ s$, we introduce *indexed* statements as defined in Fig. 8, allowing auditing to "follow the trail" to reconstruct extrapolated statements from multiple log locations.

Thus, given these definitions, we impose the following discipline:

1. If a user J wishes to make a request for a privilege **priv** in role R when invoking web service WS, the statement $R\ \mathit{says}\ \mathbf{priv}$ will be sent to WS, and the statement:

   $$(K_J\ \mathit{says}\ R\ \mathit{says}\ \mathbf{priv}) \wedge (K_R\ \mathit{says}\ J \Rightarrow R)$$

   will be logged at a fresh location ι; this location will be determined by or communicated to WS.

2. If a web service WS wishes to propagate a request s to a web service WS' in role R, and the log location associated with s is ι, then WS sends the statement $R\ \mathit{says}\ s$ to WS', and the indexed statement:

   $$(K_{\mathrm{WS}}\ \mathit{says}\ R\ \mathit{says}\ \iota) \wedge (K_R\ \mathit{says}\ \mathrm{WS} \Rightarrow R)$$

   will be logged at a fresh location ι'; this location will be determined by or communicated to WS'.

$$
\begin{array}{rcl}
\mathrm{aud}(R\ \mathit{says}\ \mathbf{priv}, \iota) & = & \text{let } \hat{s} \wedge \mathcal{R} = \mathrm{lookup}(\iota) \text{ in} \\
 & & \mathrm{assert}(\mathcal{R} = K_R\ \mathit{says}\ J \Rightarrow R) \text{ for} \\
 & & \mathrm{assert}(\hat{s} = K_J\ \mathit{says}\ R\ \mathit{says}\ \mathbf{priv}) \text{ for} \\
 & & \hat{s} \wedge \mathcal{R} \\
\mathrm{aud}(R\ \mathit{says}\ s, \iota) & = & \text{let } \hat{s} \wedge \mathcal{R} = \mathrm{lookup}(\iota) \text{ in} \\
 & & \mathrm{assert}(\mathcal{R} = K_R\ \mathit{says}\ J \Rightarrow R) \text{ for} \\
 & & \mathrm{assert}(\hat{s} = K_J\ \mathit{says}\ R\ \mathit{says}\ \iota') \text{ for} \\
 & & \text{let } s' \wedge \mathcal{R}' = \mathrm{aud}(s, \iota') \text{ in} \\
 & & (K_J\ \mathit{says}\ R\ \mathit{says}\ s') \wedge (\mathcal{R} \wedge \mathcal{R}')
\end{array}
$$

Figure 11: An Auditing Technique

If a web service WS receives a request s, with associated log location ι, online checking will be done with respect s, which is in trusted form. Auditing will use s together with location ι to reconstruct the extrapolated form of s; in particular, given such s and ι, in a security context containing ACL $\mathcal{A}$ and CCL $\mathcal{C}$, we define:

$$\mathrm{audit}(s \wedge \mathcal{C} \wedge \mathcal{A}) = \mathrm{aud}(s, \iota) \wedge \mathcal{C} \wedge \mathcal{A}$$

where aud is defined as in Fig. 11. In the definition of aud, the function assert(P) blocks execution iff the predicate P is not valid. Therefore, aud reconstructs an authorized extrapolated statement from an authorized trusted contexts, and logged indexed statements, satisfying Condition 2; the result follows by induction on the trusted request:

Lemma 3. *Suppose s is a trusted context, and audit(s) returns s'; then $[\![s']\!] = s$.*

Furthermore, aud fails if it cannot reconstruct an extrapolated statement that is not authorized for the relevant privilege. Note that this property is implicit in the definition of aud; it is not necessary to actually perform automated ABLP theorem proving on the extrapolated statement:

Lemma 4. *Suppose s is a trusted context, $s \vdash \mathbf{priv}$ is valid, and audit(s) returns s'; then $s' \vdash \mathbf{priv}$ is valid.*

The key to this result is to note that audit$(s) \vdash s$, since aud only reconstructs valid role certifications and signed statements, via the assertions embedded in its definition.

To implement log locations and auditing, we first of all extend the XML encoding of ABLP terms in Fig. 6, to include wire formatted locations:

$$
\begin{array}{rcll}
\varphi & ::= & <\!\texttt{Says}\!>\alpha, \ell<\!/\!> & \textit{formulae} \\
\ell & ::= & <\!\texttt{LogLoc}\!>\hat{\iota}<\!/\!> & \textit{log locations}
\end{array}
$$

where $\hat{\iota}$ is some standardized representation of the location ι. We also extend the format of `AuthInfo` items with location information `LogLoc`, providing the first link in the auditing discovery chain. Thus, `AuthInfo` items have the form:

$$
\begin{array}{l}
<\!\texttt{Request}\!> \\
\quad \varphi_1 \\
<\!\texttt{RoleCerts}\!> \\
\quad \varphi_2 \\
<\!\texttt{LogLoc}\!> \\
\quad \ell
\end{array}
$$

Example 4.1. *(continued)* We assume that the transaction involving Joe, WS$_M$ and WS$_{MDB}$ adhere to the trust transformation and logging discipline described above. Let $s = M\ \mathit{says}\ D\ \mathit{says}\ \mathbf{priv}_{MDB}$. Now, it is clearly the case that:

$$s \wedge \mathcal{C}_{MDB} \wedge \mathcal{A}_{MDB} \vdash \mathbf{priv}$$

so online checking succeeds. Furthermore, if WS$_{MDB}$ audits this statement, it will obtain:

$$
\begin{array}{c}
\mathrm{audit}(s \wedge \mathcal{C}_{MDB} \wedge \mathcal{A}_{MDB}) \\
= \\
(K_{\mathrm{WS}_M}\ \mathit{says}\ M\ \mathit{says}\ K_J\ \mathit{says}\ D\ \mathit{says}\ \mathbf{priv}_{MDB}) \\
\wedge\ (K_D\ \mathit{says}\ J \Rightarrow D) \wedge (K_M\ \mathit{says}\ \mathrm{WS}_M \Rightarrow M) \\
\wedge\ \mathcal{C}_{MDB} \wedge \mathcal{A}_{MDB}
\end{array}
$$

and we note that $[\![\mathrm{audit}(s)]\!] = s \wedge \mathcal{C}_{MDB} \wedge \mathcal{A}_{MDB}$, and audit$(s) \vdash \mathbf{priv}$, as indicated by Lemma 3 and Lemma 4.

5. DISCUSSION

In this section we conclude with a discussion of future and related work, and a brief summary of the main points of the paper.

5.1 Authentication and Auditing

In Sect. 4, we showed how signed ABLP formulae could be implemented, but we did not address messaging authentication. For example, although signatures assure the original source of an authorization assertion, we did not specify to check the authenticated identity of Web Service users against these sources, providing no defense against spoofing and replay attacks. However, in addition to providing an appealing XML syntax with cryptography, the system in [7] embeds this syntax in the applied Pi-calculus [3]. This setting allows verification of SOAP message authentication protocols in a Dolev-Yao model, and several such protocols are verified. By expressing our messages in their syntax, we immediately obtain techniques for authenticating messages between principals. Of course, this allows only point-to-point authentication, whereas several linked intermediaries can be involved in authorization decisions in our TbV model. In [12], techniques for verifying authorization safety in a Dolev-Yao model are developed, that apply to these more complicated authorization decisions. Adapting these techniques to a TbV setting is future work.

Another interesting topic for future work is auditing. In particular, we have only sketched a technique for logging and retrieving assertions during offline verification, but we have not considered *who* is to be held responsible for providing information during auditing, nor what the consequences will be for failure. This is an important topic that needs to be considered at both high (conceptual) levels and low (implementation) levels.

5.2 Related Work

Our system is a novel web services application of an idea that has been explored in other settings, namely previous authorization frameworks based on ABLP logic. Most closely related in this regard is proof carrying carrying authorization (PCA) [5, 4], a framework for specifying and enforcing webpage access policies (though the logic used there is not ABLP, but an application specific variant). However, that system comprises a general framework for webpage access control, so expressible policies are potentially more complicated than those we propose for web services. Furthermore, there is no distinction between online and offline checking in that framework as currently conceived, though our trust-but-verify approach could be adapted to it.

Other authorization systems founded in ABLP logic include that used in the Taos operating system [21], essentially a direct implementation of a subset of ABLP logic. Also, Wallach et al. have formalized the "security-passing style" of the Java stack inspection mechanism in a subset of ABLP [18, 20], which has served as a foundation for the SAFKASI programming language-based security architecture [19].

The SPKI/SDSI architecture [10, 16] is another authorization system for distributed communication. Their security model is similar to ABLP, but is based on a system of local names and emphasizes delegation. While SPKI/SDSI is a well-designed and appealing authorization framework, we conjecture that its emphasis on local namespaces may not be as salient in a web services context. In particular, SPKI/SDSI namespaces are based on *known* names, but a point we emphasize is that web services operate in the highly open and volatile environment of the web, so that web services generally cannot expect to know their invokers. Of course, our use of ABLP does not preclude the adoption of SPKI/SDSI principles later on, in future considerations of e.g. revocation, since the semantics of SPKI/SDSI has been shown to be embeddable within ABLP [15, 1].

In [17], a web services authorization system is defined, allowing specification of security policies in temporal logic, which are translated into reference monitors embedded in applications software. However, their approach is focused on complex policies for usage patterns similar to [4], and they make no online/offline checking phase distinction.

Related work on web services authentication includes an XML-based logic for web services authentication [7, 13], that is embedded within the applied Pi-calculus [11], allowing verification of web service security protocols via both human and machine proof techniques. It is likely that their approach will be relevant to future considerations of web services request authentication for our model.

5.3 Conclusion

In this paper, we have introduced a trust-but-verify framework for web services. We have used ABLP logic to establish a formal setting for framework design, and specified the conditions that any trust-but-verify implementation must satisfy. The central ideas we have presented are the separation of online and offline authorization phases, the notion of a trust transformation that establishes a meaningful relation between these phases, and a characterization of auditing for offline verification of online checking. We have specified an XML wire format for ABLP assertions in SOAP messages, and have described an example TbV system architecture.

6. REFERENCES

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Prog. Lang. Syst.*, 15(4):706–734, 1993.

[3] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM Press.

[4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, Singapore, Nov. 1999. ACM Press.

[5] L. Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton University, 2003.

[6] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in java. Technical Report TR-603-99, Princeton University, Computer Science Department, July 1999.

[7] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–209. ACM Press, 2004.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. W3C Note, May 2005. `http://www.w3.org/TR/SOAP/`.

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

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, Sept. 1999.

[11] C. Fournet and M. Abadi. Hiding names: Private authentication in the pi calculus. In *Proceedings of the International Symposium on Software Security*, number 2609 in LNCS, pages 317–338. Springer-Verlag, November 2003.

[12] C. Fournet, A. Gordon, and S. Maffeis. A type discipline for authorization policies. In *Proceedings of the 14th European Symposium on Programming (ESOP'05)*, 2005.

[13] A. Gordon, K. Bhargavan, C. Fournet, and R. Pucella. Tulufale: A security tool for web services. In Springer, editor, *Formal Methods for Components and Objects*, LNCS, 2003.

[14] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *Proceedings of the 2002 ACM workshop on XML security*, pages 18–29. ACM Press, 2002.

[15] J. Howell and D. Kotz. A formal semantics for SPKI. Technical Report 2000-363, Dartmouth College, 2000.

[16] R. Rivest and B. Lampson. SDSI — a simple distributed security infrastructure, 1996.

[17] E. G. Sirer and K. Wang. An access control language for web services. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 23–30. ACM Press, 2002.

[18] D. S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, 1999.

[19] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, 2000.

[20] D. S. Wallach and E. W. Felten. Understanding java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[21] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. Technical Report 117, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, Ca 94301, December 1993.