Verifying Policy-Based Security for Web Services

Karthikeyan Bhargavan
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

ABSTRACT

WS-SecurityPolicy is a declarative configuration language for driving web services security mechanisms. We describe a formal semantics for WS-SecurityPolicy, and propose a more abstract link language for specifying the security goals of web services and their clients. Hence, we present the architecture and implementation of fully automatic tools that (1) compile policy files from link specifications, and (2) verify by invoking a theorem prover whether a set of policy files run by any number of senders and receivers correctly implements the goals of a link specification, in spite of active attackers. Policy-driven web services implementations are prone to the usual subtle vulnerabilities associated with cryptographic protocols; our tools help prevent such vulnerabilities, as we can verify policies when first compiled from link specifications, and also reverify policies against their original goals after any modifications during deployment.

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, Pi Calculus, XML Security

1. INTRODUCTION

Web services can protect SOAP [23] messages sent over insecure transports by embedding security headers. The WS-Security standard [17] defines how such headers may include signatures, ciphertexts, and a range of security tokens, such as tokens identifying particular principals. Relying on generic implementations in libraries, web service programmers can pick and mix headers for messages, depending on their security needs, thereby designing their own application-level protocols above SOAP.

Like all networked systems secured via cryptography, web services may be vulnerable to a class of attacks, first described by Needham and Schroeder [18] and first formalized by Dolev and Yao [11], where an attacker may intercept, compute, and inject messages, but without compromising the underlying cryptographic algorithms. In the setting of SOAP security, we refer to these as *XML rewriting attacks*, as opposed to attacks on web services implementations, such as buffer overruns or SQL injection.

CCS'04, October 25-29, 2004, Washington, DC, USA.

WS-SecurityPolicy [10], built on the WS-Policy [8] and WS-PolicyAssertion [9], is a declarative XML format for programming how web services implementations construct and check WS-Security headers. By expressing security checks as XML metadata instead of imperative code, policy-based web services conform to the general principle of isolating security checks from other aspects of message processing, to aid human review of security. Moreover, coding security checks as XML metadata aids interoperability since the metadata may easily be exchanged between different implementations on different platforms.

Still, driving web services security from WS-SecurityPolicy is no panacea. First, despite its name, WS-SecurityPolicy drives low-level mechanisms that build and check individual security headers; we need a way to relate policies to more abstract, application-level goals such as message authentication or secrecy. Second, the configuration files, including WS-SecurityPolicy files, of a SOAP-based system largely determine its vulnerability to XML rewriting attacks; WS-SecurityPolicy gives freedom to invent new cryptographic protocols, which are hard to get right, in whatever guise.

We propose a new language and two new tools to address these problems. Our high-level link specification language describes intended secrecy and authentication goals for messages flowing between SOAP processors; our link language is a simple notation, covering some common cases, and could easily be generated from a simple UI or a systems modelling tool. Our first tool compiles link specifications to WS-SecurityPolicy configuration files. In part because of the subtle semantics of policy files, it is safer to generate them from link specifications than write them directly. Our second tool is an analyzer to check (prior to execution) whether the security goals of a link specification are achieved by a given set of WS-SecurityPolicy files. Our analyzer works by constructing a formal model of a set of SOAP processors, together with the security checks they perform, in the TulaFale scripting language, a dialect of the pi calculus. We then run existing tools for TulaFale to check automatically whether the security goals of the formal model are vulnerable to any XML rewriting attacks.

We have implemented the techniques of this paper for a particular policy-driven implementation of Web Services security, the Web Services Enhancements toolkit (WSE) [15], and in the process found and corrected several security problems. In principle, our approach can easily be adapted to other systems based on WS-SecurityPolicy. For the sake of readability, we suppress most details of the XML wire format in this paper, and instead use an abstract notation for policies; nonetheless, our implementation directly consumes and produces the XML files used by WSE.

Our work builds on much recent research on developing automatic analyses of abstract descriptions of cryptographic protocols. Specifically, it is part of our attempt to give a formal semantics to

web services security [12, 2, 4]. We rely on previous models of WS-Security in the pi calculus, and compose these models to support declarative security policies. To the best of our knowledge, the tools described here are the first to check implementation files configuring SOAP security protocols for vulnerabilities to XML rewriting attacks. Having tools construct the formal model to be analyzed is advantageous as it eliminates any human error arising from constructing ad hoc models by hand. It also enables the systematic testing of the policy files used to deploy web services.

The paper is organized as follows. In Section 2, we review and discuss security policies for web services, and set up notations. Section 3 describes the architecture of our formal tools. In Sections 4 and 5, we explain their implementation, first as an operational semantics for policies expressed as TulaFale predicates, then as a translation of links to policies and to security checks. Section 6 explains the formal security results we can automatically derive for the policies generated from these links. Section 7 discusses some extensions of our basic results. Section 8 concludes. A technical report [3] provides detailed examples and the formal scripts used to verify their correctness.

2. SECURITY POLICIES FOR WEB SERVICES (REVIEW)

2.1 Web Services and their Configuration

We consider systems of SOAP [23] processors distributed across multiple machines. Each processor may send and receive SOAP envelopes for various services. The envelope format is processed by generic system libraries driven by declarative configuration files, whereas the envelope payload is processed by imperative application code. For instance, a simple (unprotected) envelope may be of the form:

```
<Envelope>
  <Header>
    <To>http://bobspetshop.com/service.asmx</To>
    <Action>http://petshop/premium</Action>
    <MessageId>uuid:5ba86b04...</MessageId></Header>
  <Body>
    <GetOrder><orderId>20</orderId></GetOrder></Body>
</Envelope>
```

(For the sake of brevity, we omit XML namespace information in this paper.) This envelope has a message body, representing a method call at the service, preceded by optional WS-Addressing [7] headers that provide the URIs of the target service and action and a unique message identifier. To return the result of `GetOrder(20)`, the service may send a response envelope that includes a header `<RelatesTo>uuid:5ba86b04...</RelatesTo>` instead of `<To>` and `<Action>` to route the response to the requester.

SOAP envelopes can be protected using a `<Security>` header containing security tokens [17]. For instance, message integrity may be protected by a token embedding an XML digital signature, whereas the identity of the sender may be passed as a second token embedding an X.509 certificate. Parts of the envelope may be encrypted, possibly using a third token to indicate how to derive the decryption key.

2.2 WS-Policy and WS-SecurityPolicy

Next, we define an abstract syntax for the policies considered in this paper. We omit the explicit choice of algorithms for canonicalization, secure hash, shared-key encryption, and so on, and assume a fixed algorithm for each purpose. We use the constructor List for ML-style lists, separated by commas and enclosed within brackets.

Policies:

part : Part ::=	Message Parts
Header(tag : string)	SOAP Header
Body	SOAP Body
tk : Token ::=	Token Descriptions
X509	X.509 Cert
X509(sub : string)	with subject sub
Username	User/Password
Username(u : string)	with user u
pol : Pol ::=	Policies
None	Empty, true
All(ps : List(Pol))	Conjunction
OneOrMore(ps : List(Pol))	Disjunction
Integrity(tk : Token, pts : List(Part))	Integrity
Confidentiality(tk : Token, pts : List(Part))	Confidentiality

WS-Policy structures policy files as logical formulas over *base assertions* that can be composed using operators for conjunction, All[...], and disjunction, OneOrMore[...]. In the following, we omit other features of WS-Policy seldom used for security, such as the ExactlyOne[...] operator and the Rejected and Optional modifiers—our assertions are all implicitly Required.

WS-SecurityPolicy defines two base assertions for integrity and confidentiality. Each assertion refers to a key, either from an X.509 certificate or derived from a shared secret associated with the client. In SOAP envelopes, this is implemented by embedding either an X.509 token or a username token in the security header. Although the actual key is provided at runtime from a local database, the assertion may specifically request a subject name. Each assertion is also parameterized by a list of parts, denoting target elements of the envelope to be encrypted or jointly signed. Each part may be specified by its header name, or more generally using an XPath [22] expression. For each integrity assertion, a XML digital signature token is embedded in the security header. For each encrypted part, the target element is replaced with its encryption.

On the receiver side, a SOAP envelope is accepted as valid, and passed to the application, if its policy is satisfied for this envelope. Conversely, on the sender side, the protocol stack generates SOAP envelopes that satisfy its policy. Normally, the sender policy should be at least as demanding as the receiver policy. This may be enforced by exchanging and comparing policies beforehand, using auxiliary protocols.

As an example, the following policy may be used to secure the envelope shown in Section 2.1, by encrypting its message body using the service's X.509 public encryption key, and by signing all its elements using a shared secret associated with the client.

```
All [ Integrity(Username, [Header("To"),Header("Action"),
                           Header("MessageId"),Body]),
      Confidentiality(X509("BobsPetShop"),[Body]) ]
```

2.3 Policy Maps (in WSE)

Since a SOAP processor may host (and interact with) many services with diverse security requirements, it is essential to specify how policies are associated with services and envelopes. In this paper, we select the policies for processing SOAP envelopes via two partial maps from SOAP endpoints to individual policies, for incoming and outgoing envelopes, respectively. We use the following abstract syntax for policy configurations. (It is based on the local configuration format in a preliminary version of WSE 2.0, and differs a little from the format used in the released version [15].)

Configurations:

uri : URI ::= `anyLegalXmlUri`	Set of URIs
addr : Addr ::=	SOAP Endpoints
Default	Any service, any action
ToDefault(suri : URI)	Service suri, any action
ToAction(suri : URI, ac : URI)	Service suri, action ac
map : Polmap ::=	Policy Maps
Send(addr : Addr, pol : Pol)	Send Policy for addr
Receive(addr : Addr, pol : Pol)	Receive Policy at addr
cfg : Config ::= polmaps : List(Polmap)	Configurations

As an example, we give a configuration for the client that supports the request and response for http://bobspetshop.com. The configuration consists of a send policy map for generating requests, and a receive policy map for checking responses:

```
clientConfig =
  [ Send(ToAction("http://bobspetshop.com/service.asmx",
                  "http://petshop/premium"),
         Integrity(Username,Req)),
    Receive(Default,Integrity(X509("BobsPetShop"),Resp)) ]

Req = [Header("To"),Header("Action"),Header("MessageId"),
       Header("Created"),Body]
Resp = [Header("From"),Header("RelatesTo"),Header("MessageId"),
        Header("Created"),Body]) ]
```

The configuration maps requests with `<To>"http://bobspetshop.com/service.asmx"</>` and `<Action>"http://petshop/premium"</>` to a policy that signs the message body, relevant WS-Addressing headers, and a creation timestamp header. The receive policy maps all responses to a similar policy that ensures that the relevant response message parts are signed using an X.509 certificate assigned to `"BobsPetShop"`. The message parts, *Req* and *Resp*, in this configuration represent the minimum set that need to be signed for safety; hence they appear several times in this paper.

2.4 Discussion

Policies can be rather weak: for the receiver, the policy Integrity [X509("Alice"),[Body]] only guarantees that a client with an Alice certificate sent an envelope with the received message body, to some service, at some point. It provides neither message authentication nor replay protection, as an attacker can rewrite anything else in intercepted envelopes. As another example, the policy All[Integrity(t,[Header("MessageId")]),Integrity(t,[Body])] for a token t is weaker than Integrity(t,[Header("MessageId"),Body]) since the former accepts an envelope with separate signatures for the message identifier and contents.

The choice of a policy usually depends on the service and its implementation; for instance, authentication of the `<To>` and the `<Action>` elements matters if the same certificate is used for different services and actions. Similarly, elements used to implement replay protection or message correlation (typically the message identifier and the sender's timestamp) should be authenticated by default, even if the application ignores them. More generally, headers trusted by the application, say for transaction management, should be authenticated. Conversely, given intermediate SOAP processors, a service should not expect all headers to be signed.

Independently, the implementation of policies in a SOAP protocol stack is non-trivial. For instance, the ordering of encryption and signing operations obviously matters, but is left unspecified.

Finally, one cannot realistically hope to capture all security needs with a simple declarative syntax, so it is important to understand how basic needs expressible in policies can be supplemented with ad hoc mechanisms, relying for instance on custom security tokens. For instance, an essential limitation of the core policy language is that it is stateless, that is, its interpretation does not depend for example on previously-received messages. This calls for extension mechanisms for properties that concern series of messages, such as correlation between successive requests to the same service.

3. ARCHITECTURE OF POLICY TOOLS

We present the design and implementation for our tools, leaving most details to the next two sections. Our general approach, depicted in Figure 1, is to develop an operational model for web services that (1) closely reflects their actual deployments and (2) supports automated verification of security properties. As well as running web services applications using WSE, we symbolically verify their security using TulaFale, a scripting language for expressing XML security protocols.

3.1 TulaFale, a Security Tool for Web Services

TulaFale [4] is a typed language based on the applied pi calculus [1] with support for XML processing, built on top of ProVerif [6, 5], a cryptographic protocol verifier. The language has terms, predicates, and processes.

Terms combine XML and symbolic "black-box" cryptography, parameterized by a set of rewrite rules. For instance, we define AES symmetric encryption and decryption in TulaFale as follows:

```
constructor AES(bytes,bytes):bytes.
destructor decryptAES(bytes,bytes):bytes
    with decryptAES(k,AES(k,b)) = b.
```

Prolog-style predicates operate on terms; they are used to reflect the syntax and informal semantics of web services specifications. For instance, the following predicate gives a (simplified) account of a WS-Security username token, by describing how to build this XML token and compute a derived key from username *u*, secret *pwd*, timestamp *t*, and nonce *n*:

```
predicate mkUserTokenKey (tok:item,u,pwd,t:string,n:bytes,k:bytes) :–
  tok = <UsernameToken>
          <Username> u </>
          <Password Type="None"></>
          <Nonce> base64(n) </>
          <Created> t </> </>,
  k = psha1(pwd,concat(n,utf8(t))).
```

Processes express configurations of principals that send, receive, and transform terms using these predicates. Processes can generate names modelling secrets, nonces, and message identifiers; pi calculus scoping tracks knowledge of freshly generated names.

We model the attacker as some arbitrary process context, running in parallel with the system configuration, and thus able to mount any active attack combining communications, cryptography, and XML rewriting. The only restriction is that fresh names are not initially known by the attacker.

To check formal security properties, we compile our TulaFale scripts to the applied pi calculus, and then invoke ProVerif. For each property, either ProVerif succeeds, and establishes the property for all runs, in any context, or it fails with a trace that we can (usually) decompile into a TulaFale counterexample that describes an attack, or it diverges. (With a little user adjustment of scripts, divergence can usually be avoided in practice.) Properties include confidentiality (some name remains secret for all runs) and authenticity (expressed as correspondences between special events performed by processes to mark their progress). Since TulaFale scripts define processes, the general theory of the pi calculus can also be usefully applied, for instance to prove complementary properties by hand, or to generalize automatically-proved properties.

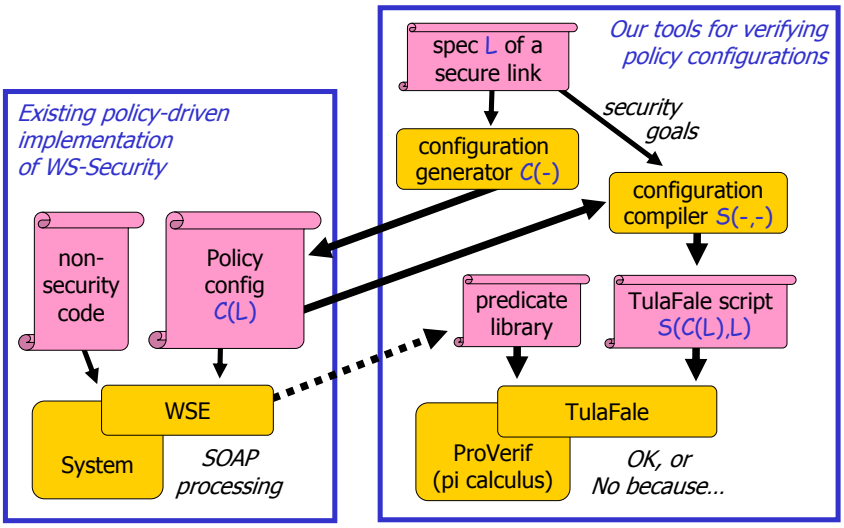

Figure 1: Generating and Checking Web Services Security Policies

Although we successfully applied TulaFale to verify a series of SOAP configurations, and reflected a significant part of WS-Security as a TulaFale library, modelling in TulaFale remains delicate: only experts can be expected to write scripts and safely interpret the results of ProVerif.

3.2 Compiling Policies to TulaFale Scripts

To verify declarative SOAP configurations, we introduce a new tool that compiles these configurations to TulaFale scripts, thereby giving a precise operational semantics to their specifications. The core of our "configuration compiler" (see Figure 1) consists of a translation from WS-SecurityPolicy formulas to TulaFale predicates on envelopes relying on (our existing model of) WS-Security. Pragmatically, our tool also collects the policy maps of a WSE implementation and automatically generates its TulaFale script. From that point, one can handwrite relatively short security properties for the configuration and verify them using TulaFale. More superficially, the tool can also detect and report common errors in policy configurations (often apparent in TulaFale), such as unauthenticated routing information.

Crucially, our tools and the actual web service runtime take as input the same policy configurations. Hence, we can directly determine web services vulnerabilities caused by misconfiguration of policy files. In contrast, in previous work, protocol verifiers work on ad hoc, handwritten, abstract descriptions of security protocols, and the gap between the handwritten description and the running code can lead to errors, and is tedious to check and to maintain. In other words, many formal techniques for verifying cryptographic protocols are now available, but their systematic application here to reflect actual distributed deployment of protocols is new.

3.3 Generating Security Goals and Policies for Abstract Configurations

In the absence of an existing XML schema for writing high-level security goals, we design our own simple format for *secure links* between SOAP endpoints hosting sets of principals acting as clients and servers. This format can mention a few basic security properties, such as message authentication, but is otherwise very limited: indeed, our goal here is that links be much easier and safer to configure than policy maps.

The link language is considerably more abstract (and less expressive) than policy maps, so that reviewing the security of a link specification is much easier than understanding the security implications of every detail in a configuration. For instance, they can be designed so that automatically-generated configurations avoid common pitfalls, thereby providing "secure by default" web services configurations.

Both our new tools take a link specification L as input. The first ("configuration generator" in Figure 1) generates WSE policy configurations $\mathcal{C}(L)$ to implement L. The second ("configuration compiler" in Figure 1) generates a TulaFale script $\mathcal{S}(C,L)$, which consists of a formal model of the policy configuration C, plus security goals extracted from L.

For any L, we can check correct generation of $\mathcal{C}(L)$ by compiling to the script $\mathcal{S}(\mathcal{C}(L),L)$, and running the TulaFale verifier. Alternatively, we can use a different (or a modified) configuration C', for instance by handwriting some of the policies, and check that the amended configuration still meets the original security goals, by verifying the script $\mathcal{S}(C',L)$. In this case, we automatically verify formal security guarantees, without the need to manipulate TulaFale scripts. For instance, one could run the verifier whenever the configuration is edited, before committing the changes to a live system.

4. FROM POLICIES TO PROCESSES

From policy configurations, the configuration compiler of Figure 1 generates scripts representing distributed systems of SOAP processors, with an arbitrary number of senders and receivers. These scripts provide our formal semantics; they can be read as concurrent programs coded in the pi calculus. This section explains important parts of these scripts, partly by example. We also experimented with variants of these scripts with richer models of processors and attackers, discussed in Section 7.

A script essentially implements a single, generic SOAP processor. It consists of the composition of four subprocesses, with the active attacker being an implicit process running in parallel. Two of these subprocesses, UsernameGenerator and X509Generator, code an abstract interface for managing secrets; they model our assumptions on principals, trust, and insider attacks; they do not depend on the configuration. The two remaining subprocesses, GenericSender and GenericReceiver, code SOAP processors that send and receive envelopes on behalf of principals; they rely on predicates compiled from policy maps.

4.1 Principals, Trust, and Insider Attacks

For simplicity, principals are identified by their (string) name, as they appear in authentication tokens: their subject field in X.509 certificate, and their username in username tokens. In WSE, principals provide code describing which envelopes to send and what to do with received envelopes. These details of the application are best left implicit in our model. Instead, we implement a control interface that enables the attacker to make these decisions. In addition, we provide a control interface enabling the attacker to trigger the generation of certificates and shared secrets for arbitrary principals, and to control whether they are leaked.

The generation of shared passwords is modelled as follows:

```
process UsernameGenerator() =
 (!in genUPChan(u);
   new pwdu;
   let entry = <UserPassword>
                 <Username>u</><Password>pwdu</></> in
   (!out dbChan(entry)))
|(!in genLeakUPChan(u);
   new pwdu;
   let entry = <UserPassword>
                 <Username>u</><Password>pwdu</></> in
   ((begin Leak(u); out publishChan(pwdu)) | (!out dbChan(entry))))
```

This process has two replicated inputs (**!in**) on channels genUPChan and genLeakUPChan. Whenever the environment sends the name u of a principal on channel genUPChan, this message is received by the first replicated input, a fresh secret password pwdu is generated (**new** pwdu), and the username and password are recorded as a replicated message entry sent on channel dbChan (**!out**). As opposed to genUPChan, channel dbChan is private to the SOAP processor: the entry may be read by the senders and receivers detailed in Section 4.2 below, but not by the environment (including the attacker).

The channel genLeakUPChan implements a similar service that models passwords leaked to the attacker. As above, each message on genLeakUPChan triggers the generation of a fresh secret password for u and its recording on dbChan. In addition, the password is sent on public channel publishChan, and can thus be read by the environment. Before leaking the password, however, an event Leak(u) is issued, indicating that principal u can no longer be trusted. Such events are invisible to the SOAP processor; they are used only to specify proof goals that account for leaked secrets.

Our model assumes that passwords are cryptographically strong, that is, are not subject to guessing attacks; we leave extensions to the web services policy framework to protect weak user-memorable passwords as future work.

The generation of certificates is similar: X509Generator in the technical report [3] implements a single certification authority with two public channels genXChan and genLeakXChan.

4.2 Generic Senders and Receivers

Our SOAP processors act on behalf of principals by reading their entries on channel dbChan. Without loss of generality, our script can thus include a single generic sender and a single generic receiver. (Formally, we can show that a configuration with multiple SOAP processors is observationally equivalent to a configuration with a single generic processor hosting all principals.)

The SOAP sender, illustrated below in a simple case, repeatedly inputs an envelope env from the environment on channel initChan and then instantiates the sending and receiving principals by reading their entries, sid and rid, from dbChan.

```
process GenericSender() =
 !in initChan(env);
  in dbChan(sid); in dbChan(rid);
  new freshid;
  filter mkConformant(env,[sid],[rid],[freshid],outenv) → outenv in
  filter linkAssert(sid,rid,env,a) → a in begin (Log,a); //* cf. Section 5
  out (httpChan, outenv)
```

This process attempts to enforce the send policy for messages between these principals by rewriting env into a policy-compliant envelope outenv, then sending this new envelope on httpChan, a public channel representing the network. To this end, it first creates a fresh message identifier, freshid. Then, it uses the **filter**...**in** construct to call TulaFale predicates mkConformant and linkAssert before actually sending the new envelope.

Policy enforcement depends on the configuration, via the predicate mkConformant, defined below. This predicate picks a send policy and attempts to enforce it for some set of principals by performing cryptographic operations on the input envelope.

Symmetrically, our SOAP receiver takes an envelope from the attacker on channel httpChan, instantiates the sending and receiving principals, and checks the receive policy for the intended destination before accepting the envelope.

```
process GenericReceiver() =
 !in httpChan(env);
  in dbChan(sid); in dbChan(rid);
  filter isConformant(env,[sid],[rid],outenv) → outenv in
  filter linkAssert(sid,rid,outenv,a) → a in end (Log,a); //* cf. Section 5
  done
```

This process uses the predicate isConformant, defined below, that picks a receive policy and checks that the envelope conforms to it for some set of senders and receivers.

4.3 Compiling Policies to Predicates

The policy configuration of the SOAP system is enforced using two predicates, mkConformant and isConformant, whose clauses are compiled from send and received policies, respectively. We assume that each policy has a unique identifier that can be used as its name. We present the clauses generated from the client side configuration clientConfig given in Section 2. Our configuration has a single send policy (`"ClientToService"`) requiring a signature on five message parts keyed using a password-based key. This yields the predicate:

```
predicate hasSendPolicyClientToService(env:item, sids, rids, fresh:items,
                                       outenv:item) :–
  sids = [user @ _ ],
  isUserPassword(user,u,p),
  fresh = [NewMessageId n t @ _ ],
  hasRequestParts(env,Toitm,Actionitm,MessageIditm,
                  CreatedItm,Bodyitm),
  MessageIditm = <MessageId>NewMessageId</>,
  mkUserTokenKey(utok,u,p,n,t,k),
  mkSignature(sig,"hmacsha1",k,
              [Toitm,Actionitm,MessageIditm,Createditm,Bodyitm]),
  outenv = <Envelope>
             <Header>
               Toitm Acitm MessageIditm
               <Security>
                 <Timestamp>Createditm</>
                 utok sig </></>
             Bodyitm </>
```

The predicate first extracts a username and password (u,p) for the signing principal from the sids database. Here, sids is a list of items, the operator @ denotes list concatenation, and _ is an unnamed variable standing for the tail. The predicate then extracts three fresh names: the message identifier for the envelope, a nonce, and a timestamp for generating a password-based key. Next, it parses the envelope to extract the five message parts corresponding to *Req*; here

we use a predicate call to hasRequestParts as shorthand for the five calls. The predicate then creates a new `<MessageId>` element with the fresh message identifier. The mkUserTokenKey predicate generates a new username token utok and password-based key k for the principal u using the password p and the fresh nonce and timestamp. The mkSignature predicate uses the key k to construct an XML signature sig that signs the five message parts. Finally, the predicate constructs an output envelope outenv with the signed input message parts, new message identifier, and the new username token and signature.

Each send policy P is thus translated to a different predicate hasSendPolicyP. Each send policy map, Send(addr,P), is then translated to a clause of mkConformant that invokes hasSendPolicyP if the destination service and action of an envelope matches addr. For instance, the send policy map in clientConfig translates to:

```
predicate mkConformant(env:item,sids, rids, fresh:items,outenv:item) :−
  hasHeaderTo(env,Toitm,Toval),
  hasHeaderAction(env,Actionitm,Actionval),
  Toval = "http://bobspetshop.com/service.asmx",
  Actionval = "http://petshop/regular",
  hasSendPolicyClientToService(env,sids,rids,fresh,outenv).
```

The second policy (ServiceToClient) in clientConfig is a receive policy that checks that five message parts (*Resp*) in the response message are signed with an X.509 certificate issued to the principal `BobsPetShop`. The corresponding receive policy map (for the Default address) is translated to a clause of the isConformant predicate that simply invokes hasReceivePolicyServiceToClient.

```
predicate hasReceivePolicyServiceToClient(env:item, sids, rids:items,
                                          outenv:item) :−
  hasResponseParts(env,Fromitm,RelatesToitm,MessageIditm,
                   CreatedItm,Bodyitm),
  hasSecurityHeader(env,toks),
  xtok in toks, sig in toks,
  isX509Token(xtok,"BobsPetShop",k,sids),
  isSignature(sig,"rsasha1",k,
              [Fromitm,RelatesToitm,MessageIditm,Createditm,
               Bodyitm]),
  outenv = env.

predicate isConformant(env:item, sids, rids:items, outenv:item) :−
  hasReceivePolicyServiceToClient(env,sids,rids,outenv).
```

The hasReceivePolicyServiceToClient predicate extracts the message parts corresponding to *Resp* and checks that the envelope has an X.509 token, for `BobsPetShop`, and has a signature keyed with this token that covers all the response message parts.

5. FROM LINKS TO POLICIES

A *link* defines high-level security goals for SOAP sessions between clients and servers, for a given service. In the simplest case, each session has a single message. Although this is the only case considered in this section, our tools and methodology extend to more complicated sessions; for example, Section 7 shows how to address request-response correlation. Each link specifies basic, strong security properties for the request messages from client to service, and for the response messages from service to client. The messages in each direction must be authenticated and optionally encrypted. A signature must cover the web service, the message body, and a message identifier. The formal syntax is as follows:

Links and Link Specifications:

secr : Secr ::= Clear | Encrypted	Secrecy level
ps : PrincipalSet ::= Any | pset : List(string)	Set of principals
link : Link ::=	Links
(suri : URI,	Service URI
actions : List(URI),	Enabled actions
clientPrin : PrincipalSet,	Authorized clients
servicePrin : PrincipalSet,	Service principals
secrLevel : Secr)	Body secrecy
L : LinkSpec ::= List(Link)	Link specification

We allow at most one link for each service URI in any link specification. Each link consists of the web service URI (suri), the set of allowed actions (actions), the set of principals that can act as clients (clientPrin) or as the web service (servicePrin), and the secrecy level (secrLevel) for both directions. Recall that a principal name is the username in a user/password combination or the subject name in an X.509 certificate. The secrecy level can either be Clear, meaning no encryption, or Encrypted, meaning that both requests and responses must have encrypted bodies. For encryption, both the client and server principal must use X.509 certificates. As an example, consider:

L_0 = [("http://bobspetshop.com/service.asmx",
["http://petshop/premium"], Any, ["BobsPetShop"], Clear)]

This says service `http://bobspetshop.com/service.asmx` offers an action `http://petshop/premium`. Its clients can act on behalf of any trusted principal, but the service acts only on behalf of `BobsPetShop`. All messages are authenticated, but encryption is not required.

5.1 Generating Policy Configurations

We now describe the configuration generator $\mathcal{C}(-)$ of Figure 1, that translates a list of links to a configuration consisting of a list of policy maps. We give the translation for the example L_0. The companion technical report [3] deals with the general case, and provides detailed rules for generating policy configurations.

P_{req} = OneOrMore[Integrity(Username,*Req*), Integrity(X509,*Req*)]
P_{resp} = OneOrMore[Integrity(Username("BobsPetShop"),*Resp*),
Integrity(X509("BobsPetShop"),*Resp*)]
A = ToAction("http://bobspetshop.com/service.asmx",
"http://petshop/premium")
C_c = [Send(A,P_{req}), Receive(Default,P_{resp})]
C_s = [Receive(A,P_{req}), Send(Default,P_{resp})]
$C_0 = \mathcal{C}(L_0) = C_c \ @ \ C_s$

Our request policy P_{req} requires a digital signature on (at least) the message parts *Req* signed by a principal using one of their X.509 certificates or shared client passwords. Our response policy P_{resp} requires a similar signature on *Resp*, signed specifically by `BobsPetShop`.

The policy configuration at the service C_s consists of a receive policy map for requests to A and a send policy map for responses. Conversely, at the client, the configuration C_c consists of a send policy map for requests to A and a receive policy map for responses. In general, $\mathcal{C}(L)$ generates request policy maps for each (action, client principal, server principal) tuple, and response policy maps for each (client principal, server principal) pair in L. We model the configuration of the distributed system by the concatenation of the client and server configurations. For our example link L_0, this configuration is called C_0.

5.2 Embedding Security Goals

The configuration compiler, $\mathcal{S}(-,-)$ of Figure 1, makes explicit the security goals of a link as proof obligations in TulaFale, us-

ing special *event messages* that mark important operations made by principals and key generators. Hence, before emitting a constructed envelope on httpChan, the generic sender of Section 4.2 emits **begin** Log(a) to mark its intention. Similarly, the generic receiver emits **end** Log(a) when it accepts an envelope as valid.

Informally, whenever **end** Log(a) is emitted, we would expect **begin** Log(a) to have been emitted, with matching arguments (a). The arguments to these events are automatically extracted from link descriptions, via the generated predicate linkAssert. (Crucially, this extraction is independent from the lower-level policy maps implemented in our scripts.)

The technical report [3] defines general rules for embedding our security goals in TulaFale, including the clauses for linkAssert for authenticity. Here, we define linkAssert for the request envelopes of our example link L_0.

```
predicate linkAssert(sid, rid:item, env:item, a:items) :–
  hasUid(sid,sender), hasUid(rid,responder)
  hasHeaderTo(env,Toitm,to),
  hasHeaderAction(env,Actionitm,action),
  hasHeaderMessageId(env,MessageIditm,id),
  hasHeaderCreated(env,Createditm,t),
  hasBody(env,bitm,body),
  to = "http://bobspetshop.com/service.asmx",
  action = "http://premium",
  responder in ["BobsPetshop"],
  a = [sender responder "Request" to action id timestamp body].
```

The seven hasXyz predicates collect the arguments recorded in the event: two principal identifiers plus selected elements of the envelope. The following three equations test to, action, and responder against the parameters provided in the high-level link. If these tests succeed, the arguments are returned as the last, a. Arguments are similarly collected for the response envelopes of L_0, using a second clause for linkAssert (the technical report has full details).

6. VERIFYING LINK-BASED SCRIPTS

We can now state the formal security properties checked by our tools. Recall that $\mathcal{C}(L)$ is the policy configuration generated from a link specification L, and that $\mathcal{S}(C,L)$ is the TulaFale script (essentially a pi calculus process, with some embedded assertions) generated from a policy configuration C and a link specification L. Hence, $C_0 = \mathcal{C}(L_0)$ and $S_0 = \mathcal{S}(C_0,L_0)$ are the policy configuration and pi calculus process, respectively, for our example link.

Authentication and Adequacy Goals for Processes:

A process P is *robustly safe* when, for any run in any context, if **end** Log(a) occurs, then either **begin** Log(a) or **begin** Leak(u) with [a = u @ _] previously occurred.
P is *functionally adequate for a* when, for some run in some context, **end** Log(a) occurs.

These observational properties have a direct security interpretation for processes modelling web services configurations since, by construction, the emission of observed events (for instance, for robust safety **begin** Log([u] @ ass), **end** Log([u] @ ass), and **begin** Leak(u)) are carefully controlled (for instance, they are emitted only when a sender sends an envelope, a receiver accepts an envelope, and the password generator leaks a password, respectively).

Continuing with our running example C_0 and L_0, our first theorem illustrates how our tools can be used to verify the correctness of a fixed policy configuration against a link specification.

THEOREM 1. *The process $\mathcal{S}(C_0,L_0)$ is robustly safe, and functionally adequate for some* a *of the form* [_ "BobsPetShop" "Request"@ _] *and* ["BobsPetShop"_ "Response"@ _].

The proof is fully automated. All the theorems in this paper are proved by invoking ProVerif on a generic desktop computer with a 3GHz Pentium 4 processor and 1GB of memory. For this theorem, ProVerif takes around 3 minutes to prove the security goals for the 520 line generated TulaFale script.

Our proof technique can be extended to establish theorems on general classes of policy configurations and link specifications. We present two theorems in this direction, although we omit the more complex scripts and the (human-proved) lemmas involved in their proofs. While the automated part of the proof of Theorem 2 takes a few minutes, Theorem 3 takes about ninety minutes.

The following formalizes the intuition that *all* configurations generated from link specifications are safe:

THEOREM 2. *For any link specification L, process $\mathcal{S}(\mathcal{C}(L),L)$ is robustly safe.*

The next result covers the common case of remote principals trusted to implement some unknown security policy. For instance, they may use their signing keys to generate signatures requested in arbitrary policies.

THEOREM 3. *For any link specification L and configuration C such that C and $\mathcal{C}(L)$ have the same* Receive *policy maps, the process $\mathcal{S}(C,L)$ is robustly safe.*

This asserts: if server policies suffice to validate a link specification, Send policies are immaterial for authenticity. Clearly, we cannot hope to retain functional adequacy in this case. Dually, a more general result on secrecy depends only on the Send policies.

7. EXTENDED SECURITY MODELS

In this section, we present extensions of the technical results in Sections 4, 5, and 6. We present details on incorporating secrecy and encrypted links, we explain how message correlation is encoded, and we give a logical refinement relation on policies.

7.1 Secrecy

The link language allows encrypted links that require that message bodies in both directions be kept secret from the attacker. For instance, consider the following encrypted variation of our example link specification L_0.

```
L1 = [("http://bobspetshop.com/service.asmx",
      ["http://petshop/premium"],
      Any, ["BobsPetShop"], Encrypted)]
```

The following configuration implements this link specification:

```
Preq = All[ Integrity(X509,Req),
            Confidentiality(X509("BobsPetShop"),[Body])]
Presp = All[ Integrity(X509("BobsPetShop"),Resp)
             Confidentiality(X509,[Body])]
Cc = [Send(A,Preq), Receive(Default,Presp)]
Cs = [Receive(A,Preq), Send(Default,Presp)]
C1 = C(L1) = Cc @ Cs
```

Here, the request and response policies require the Body to be encrypted, and since username tokens do not support encryption, both principals must use X.509 certificates.

The script generated from this link, $\mathcal{S}(C_1,L_1)$, extends $\mathcal{S}(C_0,L_0)$ in two ways. First, the send and receive policies are now translated to predicates that also implement the encryption of the body. Second, whenever the generic sender inputs a request envelope from the attacker for recipient u, it may replace the body by a secret name B after issuing the event **begin** LogS(u). The secrecy goal is then that the attacker cannot capture or compute B unless u is untrusted. A new predicate mkLinkEnvelope implements the body replacement.

Secrecy Goal for Processes:

P preserves secrecy when, for any run in any context where B does not occur, if the context obtains B, then **begin** Leak(u) and **begin** KnowsSecret(u) previously occurred.

The following theorem states that the policy configuration C_1 generated from L_1 preserves the secrecy goals of L_1; it is automatically proved by ProVerif in a few minutes.

THEOREM 4. *The TulaFale script generated from L_1 and C_1 preserves the secrecy of B.*

We also establish the theorems of Section 6 for C_1 instead of C_0, and prove that all link-generated configurations preserve secrecy.

7.2 Correlation

For request-response exchanges, we require an additional authentication property. When the client accepts a response message from the web service, we want to guarantee that this message was generated in response to a particular earlier request. To formalize this property, we use modified TulaFale scripts $\mathcal{S}'(C,L)$: instead of generic senders and receivers, we model generic clients and services. A generic client sends a request and then waits for a response that matches the request. A generic service is symmetric.

As an example, we present the generic client for $\mathcal{S}'(C_0,L_0)$:

```
process GenericClient() =
 !in initChan (env);
  in dbChan (cid); in dbChan (sid);
  new freshid;
  filter mkConformant(env,[cid],[sid],[freshid],outenv) →outenv in
  filter linkAssert(cid,sid,env,aReq) →aReq in
  begin Log(aReq);
  out httpChan(outenv);

  in httpChan(respenv);
  filter isConformant(respenv,[sid],[cid],resp) →resp in
  filter hasCorrelator(resp,freshid,cid) →in
  filter hasLinkAssert(sid,cid,resp,aResp) →aResp in
  end Log(aResp);
  end LogCorr(aReq,aResp)
```

As in GenericSender, the client accepts an envelope from the environment and enforces the send policy on it by invoking the predicate mkConformant. It then issues **begin** Log(aReq), where aReq is the assertion for request messages, before sending the request message out on httpChan. Then, the client process waits for the response, checks that it conforms to a receive policy and that it is correlated to the request before issuing the events **end** Log(aResp) and **end** LogCorr(aReq,aResp). The symmetric GenericService process issues **begin** LogCorr(aReq,aResp) before generating the response.

Correlation Goal:

A process *P correctly correlates requests and responses* when, for any run in any context, if **end** LogCorr(a1,a2) occurs, then either **begin** LogCorr(a1,a2) or **begin** Leak(u) previously occurred with a1 = u @ _ or a2 = u @ _.

If no user secrets are leaked, then the key to correlation is checking that the RelatesTo field of the response echoes the fresh MessageId in the request. If insider attacks are allowed, however, this mechanism does not suffice. So, we extend responses (in mkConformant, isConformant) to additionally include and sign the user token used to authenticate the request. The predicate hasCorrelator checks that the response contains both the freshid and cid used in the request.

THEOREM 5. *The process $\mathcal{S}'(C_0,L_0)$ correctly correlates requests and responses.*

7.3 Towards a logical theory for Policies

One can treat security policies as logical formulas with integrity and confidentiality assertions as atomic propositions, combined using conjunction and disjunction. This leads to a natural notion of refinement: one policy refines another if any message that satisfies the latter also satisfies the former. Such a refinement can be used to develop rules for safely modifying policy configurations that have been proved to be secure. For instance, we can establish that refining a link-generated receive policy preserves robust safety.

Compilation to TulaFale provides an ad hoc model of the logic: we can check that the basic axioms hold, and can be pushed through process configurations (for instance, comparing parallel compositions of servers to disjunctions of policies); we can also exhibit additional laws that hold in our model, for instance, authentication without signature for username tokens, and transitivity of multiple signatures sharing a fresh name.

Receiver Policy Refinement for Link-Based Safety: $p \Rightarrow q$

We say that p refines q, written $p \Rightarrow q$ when, for all link specification L and policy configuration $C[\_]$ with a placeholder as a Receive policy, if $\mathcal{S}(C[q],L)$ is safe, then $\mathcal{S}(C[p],L)$ is also safe.

Similarly, for some given security goals, one can define a notion of refinement for sender policies, and for policy configurations. Refinement provides an abstract way of extending our results to policies that are apparently more demanding. We give some sample refinement properties, which can be established using logical refinement on TulaFale predicates generated for these policies.

Refinements of Receive Policies:

$$\frac{}{pol \Rightarrow pol} \qquad \frac{pol_1 \Rightarrow pol_2 \quad pol_2 \Rightarrow pol_3}{pol_1 \Rightarrow pol_3}$$

$$\frac{}{pol_i \Rightarrow \text{OneOrMore}[pol_1,\ldots,pol_k]}\ \forall i$$

$$\frac{\forall i : pol \Rightarrow pol_i}{pol \Rightarrow \text{All}[pol_1,\ldots,pol_k]} \qquad \frac{}{\text{All}[pol_1,\ldots,pol_k] \Rightarrow pol_i}\ \forall i$$

$$\frac{T \in \{\text{X509},\text{Username}\}}{\text{Integrity}(T(s),L) \Rightarrow \text{Integrity}(T,L)}$$

$$\frac{L' \subseteq L}{\text{Integrity}(t,L) \Rightarrow \text{Integrity}(t,L')}$$

$$\frac{T \in \{\text{X509},\text{Username}\},\ H = \text{Header}(\text{``MessageId''})}{\begin{array}{l}\text{All}[\text{Integrity}(T(s),[H]@L_1),\\ \quad \text{Integrity}(T(s),[H]@L_2)] \Rightarrow \text{Integrity}(T(s),[H]@L_1@L_2)\end{array}}$$

The first two rules express reflexivity and transitivity of refinement, while the next three encode disjunction and conjunction of policies. The last three rules express refinement properties of the Integrity assertion. Requiring a more specific token or signatures over more message parts leads to a stronger receive policy. Requiring two signatures that both sign the fresh message identifier is equivalent to requiring a single signature that covers both sets of message parts.

Each of these rules is established by manual proofs about the corresponding process translations. As an example of the last refinement rule, let C' be the configuration formed by replacing the response policy, Integrity(X509(`"BobsPetShop"`),*Resp*), in C_0 by:

```
All[Integrity(X509("BobsPetShop"),
              [Header("MessageId"),Body]),
    Integrity(X509("BobsPetShop"),
              [Header("MessageId"),Header("From"),
               Header("RelatesTo"),Header("Created")])]
```

THEOREM 6. *$\mathcal{S}(C',L_0)$ is robustly safe.*

8. CONCLUSIONS AND RELATED WORK

This paper makes two main contributions to the study of web services security. The first is a formal semantics and theory of SOAP security policies and configurations via the TulaFale scripting language. The second is the design of a simple high-level link language to express intended security properties, plus tools to generate message-level WS-SecurityPolicy files from link descriptions and to verify they correctly implement the intended security goals.

The tools build a formal model of a given SOAP configuration, with an unbounded number of senders and receivers, and show there are no vulnerabilities to XML rewriting attacks by an unbounded opponent. Inasmuch as WS-SecurityPolicy files directly control security processing for SOAP messages (as in WSE, for instance), our tools can thus verify properties of actual deployments. Although verification does not apply to the system libraries that interpret policies at run-time, it does detect or rule out errors in the policy files generated or customized by users for particular SOAP installations. Our tools would have caught errors we found by careful manual inspection, such as omission to sign or check significant headers, particularly WS-Addressing headers, in actual policies written for use with WSE.

Although there are several tools to check security policies at lower layers, such as IPSec [13], to the best of knowledge, ours are the first to check security policies at the SOAP layer.

There are other recent tools to generate WS-SecurityPolicy files from more abstract descriptions. WSE [15] itself includes a security settings wizard that gathers data analogous to our link specifications with a UI and from this generates policy files. Our tools demonstrate the feasibility of checking the correctness of the policies generated by this wizard, although we have not attempted this systematically. Tatsubori, Imamura, and Nakamura [21] describe another graphical tool for generating policies. They argue that users should specify policies in terms of application requirements (such as message flows), and that these should be separated from platform capabilities (such as availability of a PKI). Hence, they propose separate application and platform models, from which they generate WS-SecurityPolicy files. Their work places more emphasis on human usability issues than ours, but does not include any formal verification of generated policies.

Several recent systems [19, 16, 14, 20] generate implementation code from abstract protocol descriptions in Dolev-Yao [11] formalisms such as strand spaces, CAPSL, and the spi-calculus. Since the abstract protocol descriptions can be formally verified, confidence in the Java code generated by these systems is greater than in a handwritten implementation. Our configuration compiler works in the opposite direction from these systems, in that it generates a Dolev-Yao model (TulaFale) from implementation code. An advantage of this direction is that end users get the benefits of formal verification without needing to rewrite their protocols in a new language. (With our tools, end users still need to write security goals, either directly in TulaFale, or via our link language.)

Zheng, Chong, Myers, and Zdancewic [24] generate distributed cryptographic protocols to realize type-based integrity and confidentiality policies expressed via an extension of Java's type system. Their approach, rather different to ours, also avoids handwritten cryptographic protocols.

Web services security will certainly evolve; the current specifications are partly inadequate and their connection to a more abstract security management layer is delicate. Flexible policy languages seem useful and necessary as web services needs and architectures develop. Still, flexibility can be the enemy of security, and we may hope that standard policies and practices can be agreed. Our semantics and tools should help along the way.

Acknowledgements. Bruno Blanchet implemented several extensions to ProVerif needed for this work. Riccardo Pucella prototyped a first compiler from policies to TulaFale during an internship. Tuomas Aura, Daniel Stieger, and the anonymous reviewers made useful comments on earlier versions of the paper.

9. REFERENCES

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

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

[3] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. Technical Report MSR–TR–2004–84, Microsoft Research, 2004.

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

[5] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society Press, 2001.

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

[7] D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, Aug. 2004. W3C Member Submission, at `http://www.w3.org/Submission/ws-addressing/`.

[8] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web services policy framework (WS-Policy), May 2003.

[9] D. Box, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, P. Patrick, C. von Riegen, and J. Shewchuk. Web services policy assertions language (WS-PolicyAssertions), May 2003.

[10] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. Web services security policy language (WS-SecurityPolicy), Dec. 2002.

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

[12] 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.

[13] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. *International Journal of Information Security*. To appear.

[14] S. Lukell, C. Veldman, and A. C. M. Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunication Networks and Applications Conference (SATNAC)*, 2003.

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

[16] F. Muller and J. Millen. Cryptographic protocol generation from CAPSL. Technical Report SRI–CSL–01–07, SRI, 2001.

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

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

[19] A. Perrig, D. Song, and D. Phan. AGVI – automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, LNCS, pages 241–245. Springer, 2001.

[20] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, volume 1, pages 400–405, 2004.

[21] M. Tatsubori, T. Imamura, and Y. Nakamura. Best practice patterns and tool support for configuring secure web services messaging. In *International Conference on Web Services (ICWS'04)*, pages 244–251, 2004.

[22] W3C. *XML Path Language (XPath) Version 1.0*, 1999. W3C Recommendation, at http://www.w3.org/TR/xpath.

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

[24] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 236–250, 2003.

APPENDIX

A. SAMPLE POLICY CONFIGURATION

The following is the XML policy configuration corresponding to $C_1 = C(L_1)$. The first two policy maps form the client configuration C_c while the last two form the service configuration C_s. The overall format is based on the config files of WSE while the policy format conforms to WS-SecurityPolicy.

```
<PolicyMappings>
  <SendPolicy>
    <To>http://bobspetshop.com/service.asmx</To>
    <Action>http://premium</Action>
    <Policy Id="ClientToServer1">
      <All>
        <Confidentiality>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
              <Claims>
                <SubjectName>BobsPetShop</SubjectName>
              </Claims></SecurityToken></TokenInfo>
          <MessageParts>Body()</MessageParts>
        </Confidentiality>
        <Integrity>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
            </SecurityToken></TokenInfo>
          <MessageParts>Body() Header("To") Header("Action")
                Header("MessageId") Header("Created")
          </MessageParts></Integrity>
      </All></Policy></SendPolicy>
  <ReceivePolicy>
    <To>default</To>
    <Action>default</Action>
    <Policy Id="ServerToClient2">
      <All>
        <Integrity>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
              <Claims>
                <SubjectName>BobsPetShop</SubjectName>
              </Claims></SecurityToken></TokenInfo>
          <MessageParts>Body() Header("From")
             Header("RelatesTo") Header("MessageId")
             Header("Created")</MessageParts></Integrity>
        <Confidentiality>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
            </SecurityToken></TokenInfo>
          <MessageParts>Body()</MessageParts>
        </Confidentiality></All></Policy></ReceivePolicy>
  <ReceivePolicy>
    <To>http://bobspetshop.com/service.asmx</To>
    <Action>http://premium</Action>
    <Policy Id="ClientToServer3">
      <All>
        <Integrity>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
            </SecurityToken></TokenInfo>
          <MessageParts>Body() Header("To") Header("Action")
                Header("MessageId") Header("Created")
          </MessageParts></Integrity>
        <Confidentiality>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
              <Claims>
                <SubjectName>BobsPetShop</SubjectName>
              </Claims></SecurityToken></TokenInfo>
          <MessageParts>Body()</MessageParts>
        </Confidentiality></All></Policy></ReceivePolicy>
  <SendPolicy>
    <To>default</To>
    <Action>default</Action>
    <Policy Id="ServerToClient4">
      <All>
        <Confidentiality>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
            </SecurityToken></TokenInfo>
          <MessageParts>Body()</MessageParts>
        </Confidentiality>
        <Integrity>
          <TokenInfo>
            <SecurityToken>
              <TokenType>X509v3</TokenType>
              <Claims>
                <SubjectName>BobsPetShop</SubjectName>
              </Claims></SecurityToken></TokenInfo>
          <MessageParts>Body() Header("From")
             Header("RelatesTo") Header("MessageId")
             Header("Created")</MessageParts></Integrity>
      </All></Policy></SendPolicy></PolicyMappings>
```