Verifying Policy-Based Web Services Security*

Karthikeyan Bhargavan Cédric Fournet Andrew D. Gordon

Microsoft Research

November 2005
Revised November 2007

Abstract

WS-SecurityPolicy is a declarative language for configuring web services security mechanisms. We describe a formal semantics for WS-SecurityPolicy and propose a more abstract language for specifying secure links between web services and their clients. We present the architecture and implementation of 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. We can verify policies when first compiled from link specifications, and also re-verify policies against their original goals after any modifications during deployment. Moreover, we present general security theorems for all configurations that rely on compiled policies.

*An earlier, abridged version of this paper appears in the proceedings of the *11th ACM Conference on Computer and Communications Security*, pages 268–277, Washington DC, October 25–29, 2004.

Contents

1 Introduction **1**

2 Security Policies for Web Services **3**
- 2.1 Web Services and XML Rewriting Attacks . . . 3
- 2.2 WS-Security . . . 4
- 2.3 WS-Policy and WS-SecurityPolicy . . . 5
- 2.4 Mapping Messages to Policies . . . 7
- 2.5 Discussion . . . 9

3 Architecture of Policy Tools **10**
- 3.1 TulaFale, a Security Tool for Web Services (Review) . . . 11
- 3.2 An Abstract Syntax for Link Specifications . . . 12
- 3.3 Our Generator and Compiler Tools . . . 13
- 3.4 Verified Security Properties . . . 14

4 Generating Configurations from Links **16**

5 Compiling Configurations to Scripts **18**
- 5.1 Managing Principals and their Credentials . . . 19
- 5.2 Generic Senders and Receivers . . . 21
- 5.3 Operations on Envelopes . . . 23
- 5.4 Compiling Policies to Predicates . . . 23
- 5.5 Compiling Configurations to Clauses . . . 29
- 5.6 Embedding Security Goals . . . 29

6 Verifying Link-Based Scripts **32**
- 6.1 Security Properties, Formally . . . 33
- 6.2 Verifying Example Configurations . . . 34
- 6.3 Validating All Link-Generated Configurations . . . 35
- 6.4 Validating Partial Configurations . . . 38
- 6.5 Proof of Theorem 4 . . . 39
- 6.6 Proof of Theorem 5 . . . 48

7 Correlation **49**

8 Conclusions and Related Work **51**

1 Introduction

Web services can protect SOAP [36] messages sent over insecure transports by embedding security headers. The WS-Security standard [30] defines how such headers may include signatures, ciphertexts, and a range of security elements, 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 using WS-Security may be vulnerable to a class of attacks, first described by Needham and Schroeder [31] and first formalized by Dolev and Yao [19], 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*, to differentiate them from other classes of attack, such as buffer overruns or SQL injection.

WS-SecurityPolicy [18], with WS-Policy [16] and WS-PolicyAssertion [17], 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, when building secure systems, 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, in principle, the metadata may easily be exchanged between different implementations on different platforms.

Still, driving web services security from declarative policies 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 policy 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. As evidence, during security reviews of handwritten policies, we encountered vulnerabilities to a range of attacks, including redirection, impersonation, and replay attacks.

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; this link language is a simple notation, covering some common cases, and could easily be generated from a simple user interface 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 significantly 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 verify these security goals—or to detect any vulnerabilities to XML rewriting attacks.

We have implemented the techniques of this paper for a particular policy-driven im-

plementation of Web Services security, a version of Web Services Enhancements 2.0 (WSE) [28]. In principle, our approach can easily be adapted to other implementations of WS-SecurityPolicy and, more generally, to other systems based on declarative security configurations. Still, it is important to focus on the semantics of a particular system to develop tools and to obtain precise results.

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 [23, 4, 9]. We rely on previous models of WS-Security in the pi calculus, develop models for WS-SecurityPolicy, 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. Section 2 reviews, discusses, and sets up notations for web services security policies. Section 3 describes the architecture of our formal tools, introduces our link specification language, and outlines our security properties. Section 4 describes the generation of policy configurations from link specifications, while Section 5 describes our semantics of policy configurations via TulaFale predicates and processes. Section 6 describes formal security results that we can automatically derive for the policies generated from these links. Theorems 1 and 2 are authenticity, secrecy, and basic liveness results for specific policy configurations generated from example links. Theorem 3 is the more general result that for every link specification, the generated policy configuration possesses the expected authenticity and secrecy properties. (This theorem implies the first two, at the cost of significantly longer run time in the theorem prover.) Turning to policy configurations only partially generated from link specifications, Theorem 4 establishes authenticity properties of any policy configuration, so long as its policies for processing incoming messages conform to the link specification, even if those for outgoing messages do not. Theorem 5 is the converse result for policy configurations where the policies for outgoing messages conform to the link specification. Section 7 outlines an extension of our semantics to correlate requests and responses; Theorem 6 establishes message correlation for one of our examples. Section 8 concludes.

Additional materials are available online, as part of the TulaFale tool [9], in directory `ws-policies`. The directory contains all files mentioned in this paper, including sample policy files, as well as the TulaFale scripts used to establish our formal results.

This article is a revised version of a technical report [5]. WS-SecurityPolicy and related specifications have been developed and refined since the research in this paper was completed. When we refer to WS-SecurityPolicy in this paper we mean WS-SecurityPolicy 1.0 [18], as implemented by the version of WSE used as our research platform. Neither the research in this paper nor our discussion of WS-SecurityPolicy applies directly to WS-SecurityPolicy 1.1 [25], the most recent version, which adopts a significantly different style compared to previous versions.

2 Security Policies for Web Services

2.1 Web Services and XML Rewriting Attacks

We consider systems of SOAP [36] processors distributed across multiple machines. The primitive message pattern is asynchronous communication of a single SOAP message from one processor to another. A common derived message pattern is a request-response protocol between a client and a server. A single SOAP processor (for example, a web server) may act both as a client and a server. For example, acting as a server, it may receive a request and then, acting as a client of another server, send a new request and use the subsequent response to construct its response to the original request.

Each SOAP message conforms to an XML schema for an *envelope*, comprising an optional *header* element for routing, security, and other metadata, plus a mandatory *body* element containing the message payload. For instance, here is a simple (unprotected) envelope.

```
<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>
```

This envelope has a message body, representing a method call at the service, preceded by optional WS-Addressing [15] headers that provide the URIs of the target service and action and a unique message identifier. To return the result of `GetOrder(20)`, the server may send a response envelope with a header `<RelatesTo>uuid:5ba86b04... </RelatesTo>` instead of `<To>` and `<Action>` to route the response to the requester.

We treat SOAP faults as ordinary SOAP responses. For the sake of readability, our presentation omits many details of the XML wire format, such as XML namespace information, and uses an abstract syntax for policies and configurations. Our formalism retains many details of the XML syntax, as they matter for security; for example, an XML signature may cover some chosen subset of the SOAP headers, so our model needs to represent the various headers. Even though our formalism hides some of the XML details, our tools directly consume and produce the XML file formats used by WSE.

There is a risk, of course, that SOAP messages may be read in transit by a passive attacker able to read network traffic. Moreover, SOAP messages may be created, modified, and replayed by an active attacker able to inject messages into the network. (In fact, the flexibility and explicitness of SOAP messages also make such attacks easier to set up.) The usual solution is to secure messages with cryptography. The details are subtle. The incorrect use of cryptography may leave vulnerabilities open to attack, as described by Needham and Schroeder [31]. The conservative recommendation of Needham and Schroeder is that the security goals of protocols based on cryptogra-

phy should be established even in the presence of a demanding adversary: one who is in control of the network, has the use of some of the cryptographic keys belonging to principals of the system, and may employ these keys in attacks against other principals.

WS-Security [30] standardizes message-based cryptographic mechanisms to protect the confidentiality and integrity of SOAP-based communications. As mentioned in the introduction, we refer to those attacks on web services that fall in the general class described by Needham and Schroeder as *XML rewriting attacks*. Our experience with actual implementations of WS-Security [8] suggests that vulnerabilities to XML rewriting attacks indeed exist in misconfigured policy-driven uses of WS-Security. Our goal in this paper is to establish that certain policy-driven applications of WS-Security effectively protect communications in the face of the conservative threat model introduced by Needham and Schroeder.

2.2 WS-Security

WS-Security specifies how to protect SOAP envelopes using a `<Security>` header containing security elements such as *security tokens*, which identify principals or sessions.

- For integrity protection, parts of the envelope may be bound together by an XML digital signature [21], with a security token indicating the identity of the signer. (Following the terminology adopted for Web Services security, a "signature" refers to any kind of authenticator, not necessarily based on public keys.)
- For confidentiality protection, parts of the envelope may be encrypted [20], then annotated with XML references indicating how to retrieve the decryption key.

As we discuss elsewhere [4], WS-Security provides a precise grammar and default processing for elements in the security header, but prescribes no fixed protocol itself. Hence, compared to traditional transport security, WS-Security is more flexible, yet more costly in terms of performance and complexity.

In this paper, we consider two representative sorts of security token; the standard defines several others. An *X.509 token* supports XML encryption and signature based on public-key cryptography. It is an XML element whose content is an encoding of an X.509 certificate, a binary format including a subject name and a public key, jointly signed by a private key of some certification authority.

A *username token* supports XML signature based on password sharing—but not XML encryption. It is an XML element that always includes a username and may include other elements such as a timestamp and a nonce. Provided sender and receiver know the password associated with the username, they can derive a shared symmetric key from the token by hashing the password with the timestamp and nonce, and then use this key to produce and verify the MAC embedded in the signature.

We assume a population of *known principals*, principals able to authenticate their identity via passwords or public keys. For simplicity, principals are identified by their name, as it appears in security tokens: the subject field in X.509 certificates, and the username element in username tokens. We let names range over arbitrary strings. We say a security token is *known* if it is recognised by the system. An X.509 token is known

if the underlying X.509 certificate is issued by an acceptable certification authority. A username token is known if the corresponding username and password are registered in the password database.

Our view of web services security emphasises these known principals, the principals corresponding to WS-Security tokens. Typically, each SOAP processor is associated with such a principal. For example, a client may be associated with a username and password, while a server may be associated with a signing key. There are often other identities associated with each SOAP processor, such as network addresses, machine names, and identities of various software components, but these identities are not directly supported by WS-Security tokens. Hence, we phrase our security properties in terms of the known principals, because these are the identities actually used as keys in WS-Security headers, and suppress the other identities. Relating these other identities to the known principals is important in practice, but is highly dependent on specific hardware and software configurations, and is beyond the scope of our model.

2.3 WS-Policy and WS-SecurityPolicy

We conceive of each SOAP processor as consisting of some application components that communicate with the network via services provided by a generic SOAP stack. A SOAP protocol stack typically provides a range of generic libraries for processing SOAP envelopes, especially their headers. Configuration files set parameters, known as policies, that govern how these libraries process messages. The application components in a SOAP processor are primarily concerned with the bodies of SOAP messages, and delegate the task of processing security headers to the SOAP stack.

WS-Policy and related specifications define the declarative XML format that configures Microsoft WSE and other libraries that implement WS-Security; in this format, a policy is a propositional formula defining a predicate on a SOAP message. WS-Policy [16] defines a syntax of formulas consisting of disjunctions and conjunctions built from any set of *base assertions*. WS-SecurityPolicy [18] defines a particular set of base assertions describing the confidentiality and integrity guarantees provided by encryptions and signatures embedded within a message according to WS-Security. The combined format is sufficiently expressive that, in many situations, all application-specific uses of WS-Security can be determined by configuration data, as opposed to imperative program code.

For this paper we adopt the following abstract syntax, which amounts to a subset of WS-Policy and WS-SecurityPolicy. Our syntax has constructors named after element tags appearing in the relevant XML formats; it also has constructors for ML-style lists, separated by commas and enclosed within brackets.

Message Parts, Token Descriptions, and Policies:

Pt : Part ::=	Message Part
Header(*tag* : string)	SOAP Header with tag *tag*
Body	SOAP Body
Tk : Token ::=	Token Description
X509	X.509 Token, any subject

X509(sub : string)	X.509 Token with subject *sub*
Username	User/Password Token, any user
Username(u : string)	User/Password Token with user *u*
Pol : Policy ::=	Policy
All($Pols$: List(Policy))	Conjunction of list of policies
OneOrMore($Pols$: List(Policy))	Disjunction of list of policies
Integrity(Tk : Token, Pts : List(Part))	Base assertion: integrity
Confidentiality(Tk : Token, Pts : List(Part))	Base assertion: confidentiality

A *message part* identifies an element of the SOAP message. The part Header(*tag*) identifies an element *tag* that is a well-defined descendant of the SOAP `<Header>` element. The part Body identifies the SOAP `<Body>` element.

A *token description* is a predicate satisfied by tokens in the SOAP `<Security>` header. Any known X.509 token satisfies X509, whereas only a known token with subject name *sub* satisfies X509(*sub*). Similarly, any known username token satisfies Username, whereas only a known token with username *sub* satisfies Username(*sub*).

A *policy* is a predicate on messages. A message satisfies All($[Pol_1, \ldots, Pol_n]$) if and only if it satisfies Pol_i for all $i \in 1..n$. A message satisfies OneOrMore($[Pol_1, \ldots, Pol_n]$) if and only if it satisfies Pol_i for some $i \in 1..n$. A message satisfies Integrity($Tk, [Pt_1, \ldots, Pt_n]$) if and only if the `<Security>` header contains a token satisfying Tk and a signature using a key derived from this token and binding at least message elements Pt_1, ..., Pt_n. A message satisfies Confidentiality($Tk, [Pt_1, \ldots, Pt_n]$) if and only if there is a token in the `<Security>` header that satisfies Tk and the message elements Pt_1, ..., Pt_n are all encrypted in place using a key derived from this token. (Since the token must support encryption, it must be an X.509 token.)

Here is an example policy:

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

A message satisfies this policy if its `<Security>` header contains a username token, an X.509 certificate for subject `"BobsPetShop"`, and a signature derived from the username token and that covers the message body as well as the `<To>`, `<Action>`, and `<MessageId>` headers, and moreover, the body of the message is encrypted with a key derived from the X.509 token.

Our syntax omits some features of WS-Policy seldom used for security, such as the ExactlyOne operator and the Rejected and Optional modifiers. We omit the explicit choice of cryptographic algorithms for canonicalization, secure hash, shared-key encryption, and so on, and assume a fixed algorithm for each purpose. We also omit the possibility of identifying message elements using XPath [35] expressions.

When used to configure an implementation of WS-Security, policies govern processing of both outgoing and incoming messages. On the receiver side, an incoming SOAP envelope is accepted as valid, and passed to the application, if the envelope satisfies the *receive policy*. Conversely, on the sender side, the implementation attempts to generate SOAP envelopes that satisfy the *send policy*. (In the WSE implementation, filtering incoming messages according to a receive policy is called *enforcement*,

while attempting to add security elements to outgoing messages to meet a send policy is called *verification*.) Typically, the send policy should be at least as demanding as the receive policy. This may be achieved by exchanging and comparing policies beforehand, using auxiliary protocols.

Our formal semantics of policy-based processing of incoming and outgoing messages, given in Section 5.4, resolves some important details missing from the informal semantics of this section and indeed the published specification [16]. In particular, it fixes an order for processing policy assertions: the lexical order in the policy is the processing order for the receiver, and is the opposite of the processing order for the sender. Hence, our example policy first encrypts the message body, then signs the encrypted message body. Following WSE, we limit which elements may be signed or encrypted: our semantics allows Header(*tag*) in an integrity assertion only if $tag \in \{$`To`, `Action`, `From`, `Created`, `MessageId`, `RelatesTo`$\}$, and disallows this construct in confidentiality assertions. We could easily extend our semantics to associate additional tags with particular envelope elements.

Our semantics for the message parts listed in an integrity assertion is that all the parts must be present in the message. In fact, WS-SecurityPolicy allows the parts to be absent and requires them to be signed only if they are present in the message. A separate `<MessagePredicate>` assertion as defined by WS-PolicyAssertion [17] can be used to mandate the presence of particular message parts. In practice, the headers mentioned in integrity assertions typically coincide with those mandated by the `<MessagePredicate>` assertion. We adopt a simplified semantics for the sake of a clean presentation, and expect it would be straightforward to extend our semantics to exactly match WS-SecurityPolicy and WS-PolicyAssertion.

2.4 Mapping Messages to Policies

Since a SOAP processor may host (and interact with) many services with diverse security requirements, we need a mechanism to specify how send and receive policies are associated with services and envelopes. In our model, based on WSE, each SOAP processor has a policy configuration to determine the policy to apply to incoming and outgoing messages. In fact, we represent a whole system of SOAP processors by a single policy configuration that represents all the policy applications at any physical node in the system. We use the following abstract syntax for policy configurations. (The syntax is based on the local configuration format in a preliminary, "technical preview" version of WSE 2.0, and differs a little from the format used in later versions [28].)

Addresses, Policy Maps, and Configurations:

uri : URI ::= `anyLegalXmlUri`	Set of URIs
Addr : Address ::=	SOAP Endpoint Address
Default	Default service and action
ToDefault(*Suri* : URI)	Default action at service *Suri*
ToAction(*Suri* : URI, *Ac* : URI)	Action *Ac* at service *Suri*
Map : Polmap ::=	Policy Map
Send(*Addr* : Address, *Pol* : Policy)	Send map for *Addr*

Receive($Addr$: Address, Pol : Policy)	Receive map at $Addr$
C : Config ::=	Configuration
$Maps$: List(Polmap)	List of policy maps

A *uniform resource identifier* (URI) is a string used to name or identify a resource.

An *address* is a predicate on SOAP messages based on their WS-Addressing headers, that is, a service URI (*Suri*) and an action URI (*Ac*) for this service. Every message satisfies Default. A message satisfies ToDefault(*Suri*) if and only if its `<To>` header contains *Suri*. A message satisfies ToAction(*Suri*,*Ac*) if and only if it satisfies ToDefault(*Suri*) and its `<Action>` header contains *Ac*.

A *policy map* is a relation between incoming or outgoing messages and policies. A send map Send(*Addr*,*Pol*) relates outgoing messages satisfying *Addr* to the policy *Pol*. A receive map Receive(*Addr*,*Pol*) relates incoming messages satisfying *Addr* to the policy *Pol*.

A *configuration* $[Map_1, \ldots, Map_n]$ is the union of the relations given by the maps $Map_1, \ldots, Map_n$. (The WSE implementation interprets the list of policy maps sequentially, whereas our semantics ignores the ordering and may select any map in the list. Hence, our configurations are more permissive than those of WSE, which enable us to conservatively verify safety properties.)

For example, a client of a service `"http://BobsPetShop.com"` may have the following configuration. The configuration consists of a send policy map for outgoing requests, and a receive policy map for incoming responses:

```
Target = ToAction("http://BobsPetShop.com/service.asmx",
                  "http://petshop/premium")
CommonParts = [Header("MessageId"), Header("Created"), Body]
ReqParts = [Header("To"), Header("Action")] @ CommonParts
RespParts = [Header("From"), Header("RelatesTo")] @ CommonParts
Cc = [Send(Target, Integrity(Username, ReqParts)),
      Receive(Default, Integrity(X509("BobsPetShop"), RespParts))]
```

This configuration relates outgoing messages matching *Target* to a policy that signs the message body, relevant WS-Addressing headers, and the creation time. The configuration maps all responses to a policy that ensures that the relevant response message parts are signed using an X.509 certificate with subject `"BobsPetShop"`. The message parts *ReqParts* and *RespParts* represent the minimum parts that need to be signed for basic safety; hence they appear several times in this paper.

Conversely, a server may have the configuration below:

```
Cs = [Receive(Target, Integrity(Username, ReqParts)),
      Send(Default, Integrity(X509("BobsPetShop"), RespParts)) ]
```

In this paper, we focus on the security layer and not the application layer of SOAP processors. Hence, to model a complete distributed configuration, it suffices to concatenate the individual configurations. For example, the distributed configuration of a system with one client and one server may be represented by the concatenation $C_c @ C_s$. This relation represents all the incoming messages satisfying any receive policy and all the outgoing messages satisfying any send policy, at any SOAP processor anywhere

in the system. The whole concatenated policy could in principle be deployed at every SOAP processor in the system; if a node receives a message meant for another processor, security processing may fail (if the appropriate keys are not actually available) or eventually the message will be dropped at the application level.

There are further examples of abstract configurations in Section 4. (The files `c0.xml` and `c1.xml` [9] contain the corresponding concrete configurations, expressed in the XML format accepted by WSE.)

2.5 Discussion

In this section, we describe some limitations of the version of WS-SecurityPolicy studied in this paper. As we discuss, the flexibility of this policy language may be problematic for security. Our approach in this paper is to develop tools to address the limitations of WS-SecurityPolicy by generating usages that provably guarantee specific high-level security goals.

Policies can be rather weak. For example, from the receiver's viewpoint, the policy

Integrity[X509(`"Alice"`),[Body]]

guarantees only that a sender with a certificate for `"Alice"` sent an envelope with the same message body as the received message, to some service, at some point. Conversely, the policy provides no authentication for the rest of the message (and in particular no replay protection) as an attacker can rewrite any header in intercepted envelopes. As another example, the policy

All[Integrity(*Tk*,[Header(`"MessageId"`)]), Integrity(*Tk*,[Body])]

for a token *Tk* is weaker than

Integrity(*Tk*,[Header(`"MessageId"`),Body])

since the former accepts an envelope with separate signatures for the message identifier and contents. Most implementations accept as valid a signature that signs more elements than requested. Hence, valid signatures generated for two successive messages using the latter policy may be rewritten into a single envelope that meets the former policy and associates, for instance, the identifier of the first message with the body of the second message. This is especially problematic when message identifiers are used to correlate requests and responses.

The choice of adequate policies usually depends on the service and its implementation; for instance, authentication of the `<To>` and `<Action>` elements matters if the same token may be accepted for different services and actions. Similarly, elements used by the SOAP protocol stack to implement replay protection or message correlation (typically the message identifier and creation time) should be authenticated, even if the application ignores them. More generally, headers trusted by the application, say for transaction management, should also be authenticated. On the other hand, in the presence of intermediate SOAP processors, a service should not expect all headers to be jointly signed.

Apart from the choice of policies, the implementation of policies in a SOAP protocol stack is non-trivial. Although our semantics uses the lexical order of Integrity

and Confidentiality assertions to determine the order of encryption and signing operations, the order is left unspecified by WS-Policy. This order needs to be well-defined, as the wrong order may introduce vulnerabilities. With signing before encryption, for instance, the hash of a plaintext element within an (unencrypted) XML signature can leak information about this element, even if the element itself is later encrypted. Conversely, with encryption before signing, a signature of an encrypted message can be blindly overridden by an attacker aiming to take credit for the encrypted message. On the positive side, confidentiality is typically enforced using authenticated encryption (which visibly fails if the wrong decryption key is used), hence a successful decryption may ensure agreement on the identity of the receiver, even if this identity is not signed. Similar issues occur in case the policy has several overlapping Confidentiality assertions, as one needs to implement them as nested encryptions.

Although senders and receivers are expected to select compatible policies, successful communications typically do not ensure agreement on their respective policies. In principle, the sender application may independently perform encryptions or insert signatures before policy enforcement, which are accepted as part of policy enforcement by the receiver SOAP stack. Conversely, for instance, a sender policy may encrypt the message body, whereas a receiver policy with no confidentiality assertion may accept the message and pass the encrypted message body to the application. Even if we assume that applications never perform any security processing on their own, an active attacker may still, for instance, encrypt an intercepted message under the public key of the receiver. Hence, successful decryption and authentication as part of policy enforcement do not necessarily imply the authenticated sender treated the message contents as confidential.

Finally, one cannot realistically hope to capture all security needs with a simple 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 assertions. For instance, an essential limitation of the core policy language is that it is stateless, that is, its interpretation does not depend on previously-received messages. This suggests extension mechanisms for properties that concern series of messages, such as correlation between successive requests to the same service.

3 Architecture of Policy Tools

This section outlines the structure of our tools and the properties they verify, leaving most details to Sections 4–7. Our general approach, depicted in Figure 1, is to develop an operational model for web services that (1) closely reflects their actual deployments, as observed experimentally with WSE, and (2) supports automated verification of security properties. Thus, as well as running web services applications, we can symbolically verify their security using TulaFale, a scripting language for expressing XML security protocols.

Section 3.1 reviews the syntax and semantics of TulaFale. Section 3.2 introduces *link specifications*, our notation for the security goals of SOAP clients and servers. Section 3.3 introduces our tools to generate policy configurations from link specifications, and to compile configurations and link specifications to TulaFale scripts. Section 3.4

describes the target security properties to be verified in these scripts.

3.1 TulaFale, a Security Tool for Web Services (Review)

TulaFale [9] is a typed language based on the applied pi calculus [1] with support for XML processing, built on top of ProVerif [12, 13, 2], a cryptographic protocol verifier. The TulaFale language has terms, predicates, and processes. The following is a brief overview of TulaFale; we refer to previous work [9] for a complete description of the language, including a detailed script example.

- Terms combine XML and symbolic "black-box" cryptography, specified by a set of functions and 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, defined in the TulaFale file `library.tf` [9], gives a (simplified) account of a WS-Security username token, by describing how to build this XML token and compute the derived key from some given username *u*, secret *pwd*, nonce *n*, and timestamp *t*:

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

  The term *tok* illustrates our syntax for XML literals; we omit the element name in the end-element marker `</>`, and also omit namespace information. The key *k* is derived from *pwd*, *n*, and *t* using a keyed hash computation given in WS-Security.

- Processes express configurations of SOAP processors and auxiliary services 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. Processes can log events to mark progress through a protocol. There are two sorts of event: **begin** and **end** events. Typically, a **begin** event marks the transmission of a message, while an **end** event marks the acceptance of a message, that is, the successful checking of an incoming message.

- The attacker is modelled as an arbitrary process context, running in parallel with the system configuration, and thus able to mount any active attack combining communications, symbolic cryptography, and XML rewriting. The only restrictions are that freshly-generated names are not initially known by the attacker,

and that the attacker may not log or observe the events we use to track progress through a protocol.

To check formal security properties, we compile TulaFale scripts to the applied pi calculus, and then invoke ProVerif. Properties include confidentiality (some name remains secret for all runs) and authenticity (expressed as correspondences between the events logged 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. 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 prevented in practice.)

We can reflect a significant part of WS-Security as a TulaFale library (see file `library.tf` [9]), and can successfully apply TulaFale and ProVerif to handwritten scripts modelling a series of SOAP protocols [3, 4, 9]. Nonetheless, modelling in TulaFale remains delicate: only experts can be expected to write scripts and safely interpret the results of ProVerif. The present work aims to widen the accessibility of these tools by generating TulaFale scripts directly from the policy files that configure SOAP stacks.

3.2 An Abstract Syntax for Link Specifications

We propose a simple format for *secure links*, which express high-level security goals for message passing between SOAP processors hosting sets of principals acting as clients and servers. (To the best of our knowledge, no existing web services specification provides a format for this purpose.) Our link format mentions a few basic, strong security properties, such as message authentication, but is otherwise very limited. Our intent is that links be easier and safer to configure than policy maps. The link language is thus 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. In particular, the format is designed so that automatically-generated configurations avoid common pitfalls, thereby providing safe defaults for web services configurations.

An individual link defines security goals for messages sent to and from a particular service URI. The messages in each direction must be authenticated and may be encrypted. Authentication must cover at least the web service, the message body, and the message identifier. The formal syntax is as follows.

Links and Link Specifications:

secr : Secr ::=	Secrecy Flag
Clear	Cleartext Message Body
Encrypted	Encrypted Message Body
ps : PrincipalSet ::=	Set of Principals
Any	All known principals

pset : List(string)	Selection of known principals
link : Link ::=	Link
(*Suri* : URI,	Service URI
Actions : List(URI),	Enabled actions
Clients : PrincipalSet,	Authorized clients
Services : PrincipalSet,	Authorized servers
Secrecy : Secr)	Body secrecy
L : *LinkSpec* ::= List(Link)	Link specification

A *secrecy flag* is either 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.

A *principal set* is a predicate on known principals. Every known principal satisfies Any. A known principal with identity u_i satisfies $[u_1,\ldots,u_n]$ if $i \in 1..n$.

A *link* is a structure (*Suri*, *Actions*, *Clients*, *Services*, *Secrecy*) that expresses confidentiality and integrity goals for all requests sent to the service URI *Suri*, and all responses sent from it. The list *Actions* enumerates the allowed SOAP actions for requests. The principal set *Clients* specifies the principals authorized as clients of the service, while *Services* specifies the principals authorized to implement the service. Finally, *Secrecy* is the secrecy flag for both directions.

A *link specification* is a collection of links, with the constraint that at most one link exists for each URI. A link specification specifies all the allowable actions in a system. The same principal may act both as a server and a client. As a first example, the singleton link specification

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

declares a service at a given URI, offering a single action, with clients that act on behalf of any known principal, and services that act only on behalf of "BobsPetShop". All messages are authenticated, but encryption is not required. The specification below additionally requires encryption for all requests and responses:

L_1 = [("http://BobsPetShop.com/service.asmx",
["http://petshop/premium"], Any, ["BobsPetShop"], Encrypted)]

3.3 Our Generator and Compiler Tools

Our two new tools appear on the right-hand side of Figure 1. Given a link specification L, the *configuration generator* returns a policy configuration $\mathscr{C}(L)$ to implement L. Given a policy configuration C and a link specification L, the *configuration compiler* returns a TulaFale script $\mathscr{S}(C,L)$ that models the semantics of C and includes security assertions that reflect the goals expressed in L.

Starting from a link specification L, we can check that the generated configuration $\mathscr{C}(L)$ is correct by producing the script $\mathscr{S}(\mathscr{C}(L),L)$ and running the TulaFale verifier. (The verification of a particular $\mathscr{C}(L)$ is entirely automatic; more generally, Theorem 3

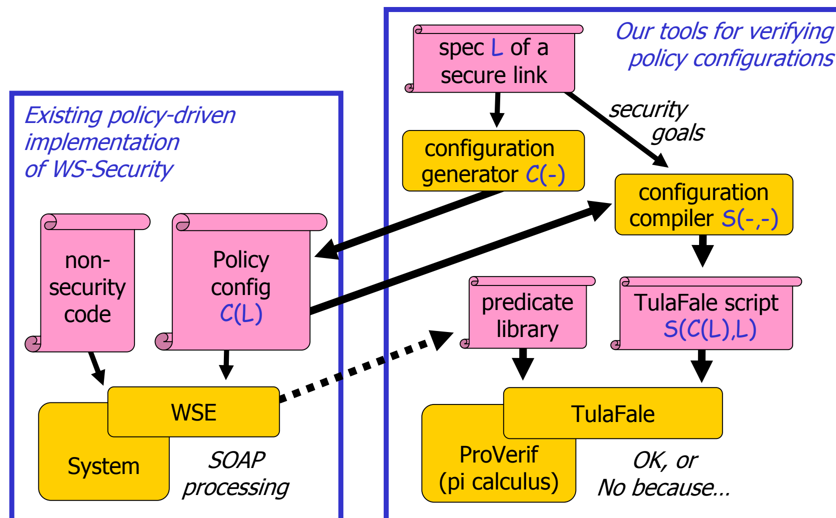

Figure 1: Generating and Checking Web Services Security Policies

asserts that $\mathscr{C}(L)$ is correct for all L, but the proof of this theorem requires a combination of hand proof and automation.) In addition to checking a generated configuration, we may consider a different configuration C' meant to enforce L, such as a configuration obtained by editing $\mathscr{C}(L)$. We can check that C' still meets the original security goals by verifying the script $\mathscr{S}(C',L)$. In both cases, TulaFale scripts are automatically produced and verified, thereby providing formal security guarantees without the need to reconsider the model. For instance, verification may be triggered whenever the configuration is updated, before committing the changes to a live system.

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. Many formal techniques for verifying cryptographic protocols are now available, but their systematic application here to reflect an existing security configuration language is new.

3.4 Verified Security Properties

Before giving precise definitions and theorems for the generator $\mathscr{C}(-)$ and the compiler $\mathscr{S}(-,-)$ implemented in our tools, we outline the security properties we can verify on the resulting configurations. (Sections 6 and 7 give corresponding formal statements.)

As explained above, properties in TulaFale are based on events that mark the transmission or acceptance of messages, as well as the compromise of principals. These events are invisible to our SOAP processors; they are used only to specify proof goals.

Our target properties should hold for all runs, despite the presence of an active attacker that controls the network and some of the principals. To allow the possibility of key compromise and insider attacks, we differentiate between *safe* and *unsafe* principals. We explicitly leak the secrets of unsafe principals to the attacker, allowing it to

attempt insider attacks. Accordingly, we introduce a **begin** event *Leak*(u) to log that the secrets of a principal u have been leaked. This event corresponds, for example, to an attacker compromising a machine to obtain passwords, or a human insider resolving to abuse their credentials. They are invisible to the SOAP processor. (Section 5.1 describes our management of principals and their partial compromise.)

Ideally, our properties should also hold irrespective of application code. To allow the possibility of arbitrary application code for the safe principals, we define generic senders and receivers, and let the attacker control their actions, as follows:

- The attacker (as an arbitrary application) can initiate any message send for any envelope, by calling our SOAP sender. At this point, the fact that the application intends to send this envelope is recorded using a **begin** event, then the envelope is cryptographically protected, according to the policy embedded in the SOAP sender, and finally given to the attacker (as the public network).
- At the other end, our SOAP receiver takes any envelope from the attacker (as the network), attempts to process it to obtain a plain envelope, according to its embedded policy. If successful, the fact that the application accepts the resulting envelope as genuine is recorded using an **end** event.

We are interested in the possible series of events, as the attacker triggers any number of runs for our senders and receivers, sequentially or in parallel, for any choice of principals and envelopes.

Authenticity is expressed as a correspondence between these events: whenever an **end** event is logged, a corresponding **begin** event must have been logged, with the same content, unless its authenticated sender is compromised. This property clearly depends on the details of what is logged: the more precise the data logged in events, the stronger the property. To this end, the recorded data is extracted from data present in the envelope and the link policy, as follows.

- For unencrypted requests (with *Secrecy*=Clear in the link), the event comprises:
  - the Request and Clear tags;
  - the message identifier and timestamp (for anti-replay protection);
  - the `<To>` and `<Action>` headers (for receiver authentication);
  - the sender name (for sender authentication); and
  - the message body (for payload authentication).
- For encrypted requests, the event comprises the same data, except for a distinct Encrypted tag and, in addition, the receiver name used for encryption (for extended receiver authentication).
- For responses, the events comprise the same data, except for a distinct Response tag and, instead of the `<To>` and `<Action>` headers, a `<From>` header (for extended sender authentication) and a `<RelatesTo>` header (for correlation, leading to receiver authentication).

(Section 5.6 defines the predicates that build these events from envelopes and links.)

Confidentiality generally depends on application code (which may leak secrets irrespective of our protocol), so we establish a property specific to our generic senders, as follows. Whenever the link specifies encryption, we substitute a fixed value, B, for the message body. Then, provided that this value B is initially secret, the attacker should not obtain B, in spite of triggering and observing any series of messages carrying B on encrypted links. (Section 5.6 defines the predicate that injects B depending on the secrecy flag.)

Correlation provides extended authentication for series of related messages, rather than for individual sends and receives. We focus on correlating requests and responses:

- We use generic clients and servers (processing two messages), rather than generic senders and receivers (processing a single message).
- Before sending and after receiving the second message, we log more extensive events containing elements from both the request and the response.

Thus, by establishing a correspondence property for these more precise events, we show that, whenever a client accepts a message as the response to a pending request, this message was indeed sent as a response to this request, unless its authenticated sender is compromised. (Section 7 develops our model for correlation.)

4 Generating Configurations from Links

The configuration generator $\mathscr{C}(-)$ translates links to configurations. We discuss and illustrate its definition below; the definition appears in the next display.

To prevent alteration of its content and to allow replay protection, every message includes a signature covering the body, the message identifier, and the timestamp (the headers `<MessageId>` and `<Created>`). Additionally, to prevent redirection of messages, the signature on a request covers the target service (the headers `<To>` and `<Action>`), and the signature on a response covers the source service (the header `<From>`). To allow correlation of requests and responses, the signature on a response covers the related request identifier (the header `<RelatesTo>`). Therefore, as in the examples of Section 2.3, *ReqParts* and *RespParts* are the lists of message parts signed in requests and responses, respectively. The function *LinkPol*(*Sndr*,*Rcvr*,*Secrecy*,*Parts*) returns a policy for a message from a sender to a receiver, identified by tokens *Sndr* and *Rcvr*, with the message body encrypted for the receiver according to *Secrecy*, and the message parts *Parts* signed by the sender.

We allow each principal mentioned in a link description to be identified at the SOAP level either by a username token or an X.509 token. Thus, the auxiliary function *Toks* sends a set of principals *ps* to a set of token descriptions *Toks*(*ps*). Each principal P in the input corresponds to a username token and an X.509 token description in the output. (If we were modelling a richer set of WS-Security tokens we might be more selective, and allow servers, for example, to be identified by Kerberos or SAML tokens, but not by username tokens.)

Generation of a Policy Configuration from a Link Specification L: $\mathscr{C}(L)$

CommonParts = [Header("MessageId"), Header("Created"), Body]
ReqParts = [Header("To"),Header("Action")] @ *CommonParts*
RespParts = [Header("From"),Header("RelatesTo")] @ *CommonParts*

LinkPol(*Sndr*,*Rcvr*,Clear,*Parts*) = Integrity(*Sndr*,*Parts*)
LinkPol(*Sndr*,*Rcvr*,Encrypted,*Parts*) = All[Integrity(*Sndr*,*Parts*), Confidentiality(*Rcvr*,Body)]

Toks(Any,*Secrecy*) =
$\{$X509$\}$ $\cup$ $\{$Username | *Secrecy* = Clear$\}$
Toks([$P_1, \ldots, P_n$],*Secrecy*) =
$\{$X509(P_1), ..., X509(P_n)$\}$ $\cup$ $\{$Username(P_1), ..., Username(P_n) | *Secrecy* = Clear$\}$

$\mathscr{C}$(*Suri*,*Actions*,*Clients*,*Services*,*Secrecy*) =
let *CSs* = [(*Ctok*,*Stok*) | *Ctok* $\in$ *Toks*(*Clients*,*Secrecy*), *Stok* $\in$ *Toks*(*Services*,*Secrecy*)] **in**
let *ReqPol* = OneOrMore[*LinkPol*(*Ctok*,*Stok*,*Secrecy*,*ReqParts*) | (*Ctok*,*Stok*) $\in$ *CSs*] **in**
let *RespPol* = OneOrMore[*LinkPol*(*Stok*,*Ctok*,*Secrecy*,*RespParts*) | (*Ctok*,*Stok*) $\in$ *CSs*] **in**
let $\mathscr{C}_C$ = [Send(ToAction(*Suri*,*Ac*),*ReqPol*) | *Ac* $\in$ *Actions*] @ [Receive(Default,*RespPol*)] **in**
let $\mathscr{C}_S$ = [Receive(ToAction(*Suri*,*Ac*),*ReqPol*) | *Ac* $\in$ *Actions*] @ [Send(Default,*RespPol*)] **in**
$\mathscr{C}_C$ @ $\mathscr{C}_S$

For example, recall from Section 3.2 the link specification L_0, which is the singleton [(*Suri*, *Actions*,*Clients*, *Services*, *Secrecy*)] given by:

Suri = "http://BobsPetShop.com/service.asmx"
Actions = ["http://petshop/premium"]
Clients = Any
Services = ["BobsPetShop"]
Secrecy = Clear

Here are the token sets corresponding to *Clients* and *Services*:

Toks(*Clients*,*Secrecy*) = $\{$Username,X509$\}$
Toks(*Services*,*Secrecy*) = $\{$Username("BobsPetShop"),X509("BobsPetShop")$\}$

The policy configuration $\mathscr{C}(L)$ given by L has a request policy map for each allowed action and each pair of clients and servers authorized in L; it also has a response policy for each pair of client and server principals. The overall configuration results from $\mathscr{C}_C$, the client-side maps, and $\mathscr{C}_S$, the service-side policy maps. Intuitively, the service configuration is installed at the web service and the client configuration is distributed to all clients.

On the service side, for each allowed action *Ac*, a receive map relates incoming messages to the service URI and action *Ac* to a disjunction of policies *LinkPol*(*Ctok*, *Stok*, *Secrecy*, *ReqParts*), for each authorized pair (*Ctok*, *Stok*). Additionally, a send map relates any outgoing message to a disjunction of policies *LinkPol*(*Stok*, *Ctok*, *Secrecy*, *RespParts*), for each authorized pair (*Ctok*, *Stok*). The client side policy configuration $\mathscr{C}_C(L)$ is the opposite of $\mathscr{C}_S(L)$: each send policy becomes a receive policy; each receive policy becomes a send policy.

For example, we calculate the configuration $\mathscr{C}(L_0)$ generated for the unencrypted link L_0 from Section 3.2 as follows, up to removing duplicate disjuncts in *reqPol* and

respPol, and where *Target* (defined in Section 2.4) indicates the action and service.

CSs_0 = [(Username,Username("BobsPetShop")),(Username,X509("BobsPetShop")),
(X509,Username("BobsPetShop")),(X509,X509("BobsPetShop"))]
$reqPol_0$ = OneOrMore [Integrity(Username,*ReqParts*),Integrity(X509,*ReqParts*)]
$respPol_0$ = OneOrMore [Integrity(Username("BobsPetShop"),*RespParts*),
Integrity(X509("BobsPetShop"),*RespParts*)]
$\mathscr{C}_{C0}$ = [Send(*Target*,$reqPol_0$),Receive(Default,$respPol_0$)]
$\mathscr{C}_{S0}$ = [Receive(*Target*,$reqPol_0$),Send(Default,$respPol_0$)]
$\mathscr{C}(L_0) = \mathscr{C}_{C0}$ @ $\mathscr{C}_{S0}$

(The file `c0.xml` [9] is the corresponding WSE configuration generated for L_0.) This configuration is similar to but more permissive than example C_c @ C_s in Section 2.4; $\mathscr{C}(L_0)$ requires authentication of (at least) the basic parts of requests and responses, and enables signatures keyed using either X.509 certificates or shared client passwords.

Similarly, here is the configuration $\mathscr{C}(L_1)$ for the encrypted link L_1 from Section 3.2, up to removing a singleton OneOrMore in *reqPol* and *respPol*:

CSs_1 = [(X509,X509("BobsPetShop")]
$reqPol_1$ = All[Integrity(X509,*ReqParts*),Confidentiality(X509("BobsPetShop"),Body)]
$respPol_1$ = All[Integrity(X509("BobsPetShop"),*ReqParts*),Confidentiality(X509,Body)]
$\mathscr{C}_{C1}$ = [Send(*Target*,$reqPol_1$),Receive(Default,$respPol_1$)]
$\mathscr{C}_{S1}$ = [Receive(*Target*,$reqPol_1$),Send(Default,$respPol_1$)]
$\mathscr{C}(L_1) = \mathscr{C}_{C1}$ @ $\mathscr{C}_{S1}$

(The file `c1.xml` [9] is the corresponding WSE configuration generated for L_1.) 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.

5 Compiling Configurations to Scripts

The configuration compiler creates scripts representing distributed systems of SOAP processors, with an arbitrary number of senders and receivers. These scripts are our formal semantics. They can be read as concurrent programs coded in the pi calculus, and make precise the informal explanations of Section 2.

The script generated from a configuration and a link is defined as follows:

Translation of Policy Configuration C and Link L to a Script: $\mathscr{S}(C,L)$

$\mathscr{S}(C,L)$ = {
import "library.tf".
$\mathscr{F}(C)$
$\mathscr{A}(L)$
UsernameGenerator() | *X509Generator*() |
GenericSender() | *GenericReceiver*() }

The script implements a single, generic SOAP processor, dependent on predicates defined by the clauses $\mathscr{F}(C)$ and $\mathscr{A}(L)$; it first includes a shared TulaFale library file `library.tf` [9] that implements various SOAP and WS-Security constructions, as

well as predicates and processes whose definition does not depend on C or L. (The scripts generated for our running configuration examples L_0 and L_1 appear in files `s0.tf` and `s1.tf`; they have about 300 generated lines each, plus 300 lines from `library.tf`.)

The generic SOAP processor consists of the composition of four subprocesses, with the active attacker being an implicit process running in parallel. Two subprocesses, *UsernameGenerator* and *X509Generator*, manage a database of principals and their cryptographic secrets; these processes model our assumptions on principals, trust, and insider attacks; they are independent of C and L. The subprocess *GenericSender* sends SOAP envelopes on behalf of a client principal; the subprocess *GenericReceiver* receives SOAP envelopes on behalf of a service principal; these processes rely on predicates defined by $\mathscr{F}(C)$ and $\mathscr{A}(L)$.

The clauses $\mathscr{F}(C)$ define predicates that generate or check envelopes according to configuration C; intuitively, they describe an implementation of a SOAP messaging stack. The clauses $\mathscr{A}(L)$ define predicates that collect data used in security assertions for link L; they are used to specify our formal goals, but they do not contribute to message processing.

In an actual SOAP stack, application-level code controls the content of each sent envelope and the action triggered by each received (accepted) envelope. Our generic SOAP processor omits such application-level details. Instead, each of the four subprocesses is under the control of the implicit attacker in that each implements an abstract *control interface*. This interface consists of public channels on which the implicit attacker can send instructions to simulate arbitrary application-level events such as creation of a principal or transmission of a particular message.

Our explanations of the translation divide up as follows. Section 5.1 describes *UsernameGenerator* and *X509Generator*. Section 5.2 describes *GenericSender* and *GenericReceiver*. Section 5.3 introduces auxiliary predicates for manipulating envelopes, defined in our TulaFale library. Sections 5.4 and 5.5 explain the clauses $\mathscr{F}(C)$ that enforce policies and configurations, respectively. Section 5.6 explains the clauses $\mathscr{A}(L)$ that specify our security goals.

5.1 Managing Principals and their Credentials

We model the names and credentials of all principals known to the system of SOAP processors using a private channel. As outlined in Section 3.4, we differentiate between *safe* and *unsafe* principals, with a **begin** event *Leak*(u) to log that the secrets of a principal u have been leaked. We provide a control interface (in `library.tf` [9]) enabling the attacker to populate this database of known principals by generating passwords and certificates, and to control whether those principals are safe or unsafe.

Database of Known Principals The private channel *dbChan* records the credentials of all principals known to the system. Each entry in the database is a replicated output of a message on this channel. The channel *dbChan* is private to the SOAP processor: the entry may be read by the senders and receivers detailed in Section 5.2 below, but not by the environment (including the attacker). On the other hand, the public channel

publishChan is for publishing data to the environment, such as the leaked credentials of unsafe principals. In TulaFale, these channels are declared as follows:

```
private channel dbChan(item)
channel publishChan(item)
```

Password Generation We now detail the generation of shared passwords associated with username tokens. The control interface consists of a pair of public channels.

```
channel genUPChan(string)
channel genLeakUPChan(string)
```

The following process implements this control interface. It consists of the parallel composition of a pair of processes listening for messages on the control channels.

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

Whenever the environment sends the name *u* of a principal on *genUPChan*, the message is received by the first replicated input (**!in** *genUPChan*(*u*)). A fresh secret password *pwdu* is generated (**new** *pwdu*:string), and the username and password are recorded as a replicated output (**!out** *dbChan*(*entry*)).

Similarly, whenever the environment sends the name *u* on *genLeakUPChan*, the message is received by the second replicated input. A fresh password is generated, and the username and password are recorded on *dbChan* as before, except that additionally, the password is sent on the public channel, *publishChan*, and can thus be read by the environment. Before leaking the password, however, we log *Leak*(*u*) to mark that the principal *u* is unsafe.

In this paper, we assume that passwords are strong secrets and, in particular, that they are not subject to guessing attacks; various formal analyses now exist to detect vulnerabilities to such attacks [26, 14] and could likely be applied to our scripts.

X.509 Certificate Generation Similar to the control interface for username generation, the X.509 control interface consists of a pair of channels.

```
channel genXChan(string)
channel genLeakXChan(string)
```

and the implementation consists of a composition of processes that create a new X.509 private-public key pair, construct a public-key certificate signed by the root authority, and record the certificate, private key, and root public key in the *dbChan* database:

```
process X509Generator() =
 new sr:bytes;
 let kr = pk(sr);
 ( (out publishChan(base64(kr)))
 | (!in genXChan(u);
      new su:bytes;
      let ku = pk(su);
      let xu = x509(sr,u,"rsasha1",ku);
      let entry =
          <X509CertSecret>
            <Principal>u</>
            <BinarySecurityToken ValueType="X509v3">base64(xu)</>
            <SecretKey>base64(su)</>
            <CAPubKey>base64(kr)</></>;
      ((out publishChan(u);
        out publishChan(base64(ku));
        out publishChan(base64(xu))) |
       (!out dbChan(entry)))))
 | (!in genLeakXChan(u); begin Leak(u); ...
```

For simplicity, all X.509 certificates in our model are issued by a single certification authority. The key *sr* is its private certification key, while *kr* is the corresponding certificate-verification key, also made available to the attacker on *publishChan*.

We omit the final part of the code, which similarly creates X.509 certificates for unsafe principals, and additionally leaks the private key as well as the public certificate to the environment. In the context of a message *u* on *genLeakXChan*, the event **begin** *Leak*(*u*) records that principal *u* is unsafe.

5.2 Generic Senders and Receivers

Our SOAP processors act on behalf of principals by reading their entries from 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 the 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*.

```
channel initChan(item)
channel httpChan(item)

process GenericSender() =
 !in initChan(env);
  in dbChan(sid); in dbChan(rid);
  new freshid:string; new n:string; new t:string;
  filter freshId (env,freshid,envId) → envId;
  filter secretEnvelope (envId,B,envB) → envB;
  filter linkAssert(sid,rid,envB,a) → a;
```

```
begin Log(a);
filter mkConformant(envB,[sid],[rid],[n t],env′) → env′;
out httpChan(env′)
```

This process attempts to enforce the send policy for messages between these principals by rewriting *env* into a policy-compliant envelope *env′*, then sending this new envelope on *httpChan*, a public channel representing some unprotected network for transporting SOAP messages. To this end, it first creates fresh names *freshid*, *n*, and *t* used as identifier, nonce, and timestamp for the message. Then, it successively calls the TulaFale predicates freshId, secretEnvelope, linkAssert, and mkConformant using the **filter**...**in** construct, as detailed below:

- The first predicate freshId replaces the message identifier in *env* with the fresh identifier *freshid*, resulting in the envelope *envId*.
- The predicate secretEnvelope, defined in Section 5.6, is used to set up our secrecy goals. The resulting envelope *envB* can be either *env* unchanged, or *env* with a (plaintext) secret value *B* substituted for the message body supplied by the environment. Informally, a configuration will preserve secrecy if the attacker cannot obtain *B* despite its usage in *envB* and *env′*.
- The predicate linkAssert, defined in Section 5.6, selects elements from the envelope *envB* to log a **begin** event, depending on the link specification (but not on the configuration).
- Finally, policy enforcement depends on the configuration, via mkConformant defined in Section 5.5. This predicate picks a send policy and attempts to enforce it for some set of principals by performing cryptographic operations on the input envelope. This yields a modified envelope, *env′*, with a security header complying with the policy. To enforce the send policy, the generic sender passes one sending principal *sid*, one receiving principal *rid* and some fresh values to be used as nonces, timestamps, keys, etc. For the simple configuration $\mathscr{C}(L_0)$, two fresh values are needed: a nonce and a timestamp.

In full generality, our generic sender depends on the maximal size of the policies in the configuration (but not on their content) in two ways. First, the generic sender must read entries for at least as many sending and receiving principals as those used in the policies; second, it must generate at least as many fresh values as those consumed by the policies.

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],env′) → env′;
  filter linkAssert(sid,rid,env′,a) → a;
  end Log(a)
```

This process uses the predicate isConformant, defined in Section 5.5, to select a receive policy (depending on the configuration) then check that the received envelope conforms to that policy. If this check succeeds, the predicate linkAssert selects elements from the accepted envelope, and a corresponding **end** event is logged (depending on the link specification). Instead of just logging an event, an implementation would then pass the authenticated contents of *env*′ to the application.

5.3 Operations on Envelopes

We rely on a series of auxiliary "structural" predicates for reading and updating the various elements of envelopes that are relevant to policy enforcement; their definitions appear in `library.tf` [9]. As examples, we give a predicate for reading the message identifier in an envelope, and another predicate for updating its message identifier:

```
predicate hasHeaderMessageId (env,id,idval:item) :−
  env = <Envelope><Header>to ac id sec @ _</> @ _ </>,
  id = <MessageId>idval</>.

predicate freshId(env,idval′,env′ : item) :−
  env = <Envelope><Header>to ac id sec @ o</>body</>,
  id′ = <MessageId>idval′</>,
  env′ = <Envelope><Header>to ac id′ sec @ o </>body</>.
```

(In the first formula, the pattern `<Header>`*to ac id sec* @_`</>` matches any XML element named `Header` whose content is a list starting with the four items *to*, *ac*, *id*, and *sec*. The @ operator is list concatenation, and the _ symbol is an anonymous variable matching any term.)

Predicate hasHeaderMessageId extracts both the message identifier element *id* and its value *idval* from envelope *env*. Predicate freshId decomposes *env*, then it creates a new identifier element *id*′ and builds a new envelope *env*′ with an updated identifier element *id*′ instead of *id*.

By definition, these predicates assume that the sought element (`<MessageId>`, for example) occurs at one fixed location within the envelope *env*, both for reading and updating this element. This assumption prevents confusion when determining, for example, what is "the" message identifier of an envelope: other `<MessageId>` elements may occur in *env*, but they are consistently ignored during processing. A more involved implementation may instead normalize (or reject) ill-formed envelopes received from the environment.

5.4 Compiling Policies to Predicates

We are now ready to describe the generation of clauses that model the policy configuration of the SOAP system. The configuration is enforced by calling one of the two predicates mkConformant and isConformant, whose clauses are compiled from send and receive policy maps, respectively. Those clauses rely in turn on pairs of predicates compiled from each individual policy. In order to name auxiliary predicates for policies, we associate a unique identifier with every policy appearing in a configuration. (In

practice, WSE configuration files embed such identifiers as fragment URIs.) Hence, for policy X, we generate predicates hasSendPolicy$X$ and hasReceivePolicy$X$.

We begin by example, with the clauses compiled from the client configuration C_c of Section 2.4, before dealing with the general case.

Clauses Compiled From Send Policies Our example configuration has a single send policy *Request* = Integrity(Username,*ReqParts*) that requires a signature on the five message parts listed in *ReqParts* keyed using a password-based key. This policy yields the predicate:

predicate hasSendPolicyRequest(*env*:item, *sids*,*rids*,*fresh*:items, *env*′:item) :−
user **in** *sids*,
isUserPassword(*user*,*u*,*p*),
fresh = [*n t*],
mkUserTokenKey (*tok*,*u*,*p*,*n*,*t*,*k*),
hasHeaderTo(*env*,*to*,_),
hasHeaderAction(*env*,*action*,_),
hasHeaderMessageId(*env*,*id*,_),
hasHeaderCreated(*env*,*created*,_),
hasBody(*env*,*body*,_),
mkSignature(*sig*,"hmacsha1",*k*,[*to action id created body*]),
replaceSecurityHeader(*env*,[*tok sig*],*env*′).

This predicate enforces policy *Request* for sending the envelope *env*, by rewriting it to *env*′, as explained below:

- Parameters *sids* and *rids* provide lists of entries representing sending and receiving principals, respectively (read from channel *dbChan* in *GenericSender*). Here, we need just one sending entry for building a `<UsernameToken>`, selected by *user* **in** *sids*.

  In general, a policy may need several principals for each role. However, for our simple link specification, we only ever use one sending and one receiving principal; that is, both *sids* and *rids* have exactly one entry each. Hence, no sender process implements a policy that, for instance, asks for two signatures on behalf of different principals. Still, we define a general translation function for all policies; we assume that *sids* and *rids* have as many entries as needed, so that our compiled scripts can easily be adapted to more complex specifications.

- Parameter *fresh* provides a list of distinct, freshly-generated values; here, we need two of them for generating a new username token, for its nonce n and timestamp t.

- The call to isUserPassword checks that *user* is a `<UserPassword>` entry, with some embedded username u and password p.

- The call to mkUserTokenKey then builds a new username token *tok* (from u, n and t) and computes the key k associated with the token (from p, n, and t).

- Parameter *env* represents the envelope before applying the policy. Each of the following five calls read from *env* an element listed in *ReqParts*, in preparation for signing.
- The call to mkSignature constructs an XML signature *sig* that signs these five message parts, using the generated key *k*.
- Finally, replaceSecurityHeader produces the resulting envelope, env' with its original content as well as an extended security header that embeds *tok* and *sig*.

We now deal with the general case (in the next display) by describing the rules for translating a named policy into a script fragment containing a sequence of predicate clauses that generate compliant envelopes—additional examples of the translation appear in files `s0.tf` and `s1.tf` [9], for every policy used in C_0 and C_1, respectively.

The translation functions in the display (and in the rest of the section) use the following notation for generated TulaFale fragments.

- Capitalized italicized identifiers, such as *Tag* or *X*, that appear as inputs to the translation denote meta-variables; they are inserted as variables or within predicate names in the target script.
- Curly parentheses, $\{\ldots\}$, denote fragments of formulas or scripts.
- Space separated fragments, F_1 F_2, denote concatenation.

The translation $\mathscr{F}_S$ relies on four auxiliary functions. The function *PartFormula* takes a list of message parts, such as those that appear in policies, and returns a fragment of a formula that extracts these parts from an envelope *env*; for instance, if the list contains Header(`"To"`), *PartFormula* calls the library predicate hasHeaderTo(*env*,*To*, *ToVal*) that parses the SOAP envelope *env* to extract the `<To>` header as the variable *To* and its contents as the variable *ToVal*; the function *PartItems* collects the list of TulaFale variables (*To*, *b*) corresponding to these extracted message parts. The function *Depth* computes the depth of a policy and the function *FreshVars* estimates the number of fresh values that will be needed to construct a message according to a policy; since each base assertion uses at most two fresh values (a nonce and a timestamp), *FreshVars*(*P*) is conservatively estimated as twice the depth of *P*.

The first three rules of $\mathscr{F}_S$ compile base assertions to predicate clauses. Each rule covers two subcases, depending on whether or not the assertion has an explicit principal name *U* (or *S*). When no name is supplied, *U* (or *S*) is handled as a fresh TulaFale variable that can match any name appearing in the entries *sids* and *rids*. The first translation rule $\mathscr{F}_S$(Integrity(Username(*U*),*Pts*),*X*) corresponds to the predicate hasSendPolicyRequest detailed above; the second rule is analogous except that it uses the private key corresponding an X.509 certificate to construct the signature; the third rule implements the assertion Confidentiality(X509(*S*)) by encrypting the message body with the public key of *S*, using the library predicate mkCipherValue.

The last two rules compile disjuctions and conjunctions of policies to the corresponding disjunctions and conjunctions of TulaFale predicates. The rule for disjunction generates one predicate with multiple clauses, one for each disjunct. The rule for

Translation of Send Policy *Pol* with Name *X* to Clauses: $\mathscr{F}_S(Pol, X)$

PartFormula(Body) = { hasBody(*env*, *b*, *bVal*), }
PartFormula(Header("*Tag*")) = { hasHeader*Tag*(*env*, *Tag*, *TagVal*), }
PartFormula([Pt_1,. . .,Pt_n]) = *PartFormula*(Pt_1) . . . *PartFormula*(Pt_n)

PartItems(Body) = *b*
PartItems(Header("*Tag*")) = *Tag*
PartItems([Pt_1,. . .,Pt_n]) = [*PartItems*(Pt_1) . . . *PartItems*(Pt_n)]

Depth(Integrity(*T*,*Pts*)) = 1
Depth(Confidentiality(*T*,*Pts*)) = 1
Depth(All[Pol_1,...,Pol_n]) = *Depth*(Pol_1) + ... + *Depth*(Pol_n)
Depth(OneOrMore[Pol_1,...,Pol_n]) = *max* {*Depth*(Pol_1), ... , *Depth*(Pol_n)}
FreshVars(*P*) = 2∗*Depth*(*P*)

$\mathscr{F}_S$(Integrity(Username,*Pts*),*X*) =
$\mathscr{F}_S$(Integrity(Username(*U*),*Pts*),*X*) = {
    predicate hasSendPolicy*X*(*env*,*sids*,*rids*,*fresh*,*env′*) :−
        sid **in** *sids*, isUserPassword(*sid*,*U*,*p*),
        fresh = [*n t*],
        mkUserTokenKey(*tok*,*U*,*p*,*n*,*t*,*k*),
        PartFormula(*Pts*)
        mkSignature(*sig*,`"hmacsha1"`,*k*,*PartItems*(*Pts*)),
        replaceSecurityHeader(*env*,[*tok sig*],*env′*). }

$\mathscr{F}_S$(Integrity(X509,*Pts*),*X*) =
$\mathscr{F}_S$(Integrity(X509(*S*),*Pts*),*X*) = {
    predicate hasSendPolicy*X*(*env*,*sids*,*rids*,*fresh*,*env′*) :−
        sid **in** *sids*,
        isX509CertSecret(*sid*,*S*,*tok*,*sk*,*cak*),
        isX509Token(*tok*,*cak*,*S*,`"rsasha1"`,*k*),
        PartFormula(*Pts*)
        mkSignature(*sig*,`"rsasha1"`,*sk*,*PartItems*(*Pts*)),
        replaceSecurityHeader(*env*,[*tok sig*],*env′*). }

$\mathscr{F}_S$(Confidentiality(X509,[Body],*X*) =
$\mathscr{F}_S$(Confidentiality(X509(*S*),[Body],*X*) = {
    predicate hasSendPolicy*X*(*env*,*sids*,*rids*,*fresh*,*env′*) :−
        hasBody(*env*,*b*,*bVal*),
        rid **in** *rids*,
        isX509CertSecret(*rid*,*S*,*tok*,_,*cak*),
        isX509Token(*tok*,*cak*,*S*,`"rsasha1"`,*k*),
        mkCipherValue(*bEnc*,`"rsa"`,*k*,*b*),
        replaceBody(*env*,*bEnc*,*env′*). }

$\mathscr{F}_S$(OneOrMore[Pol_1,...,Pol_n],*X*) = { $\mathscr{F}_S$(Pol_1,*X*) . . . $\mathscr{F}_S$(Pol_n,*X*) }

$\mathscr{F}_S$(All[Pol_1,...,Pol_n],*X*) = {
    $\mathscr{F}_S$(Pol_1,X_1) . . . $\mathscr{F}_S$(Pol_n,X_n)
    predicate hasSendPolicy*X*(env_0,*sids*,*rids*,*fresh*,env_n) :−
        fresh = [$fresh_{1,1}$ . . . $fresh_{n,f_n}$],
        hasSendPolicyX_n(env_0,*sids*,*rids*,[$fresh_{n,1}$ . . . $fresh_{n,f_n}$],env_1), . . .
        hasSendPolicyX_1(env_{n-1},*sids*,*rids*,[$fresh_{1,1}$ . . . $fresh_{1,f_1}$],env_n). }
    where f_i abbreviates *FreshVars*(Pol_i)

conjunction generates different predicates for each conjunct and combines them in a new predicate that distributes fresh values among all the conjuncts and then calls them in the reverse order. As discussed in Section 2.5, the order in which conjuncts are interpreted is unspecified in WS-Policy. In our translation, we interpret receive policies in the exact order of conjuncts; hence, the corresponding send policies must be interpreted in reverse order so that, for example, an encrypted-then-signed message is first verified, then decrypted. Our choice of ordering conforms to the implementation choices of WSE.

Clauses Compiled From Receive Policies The second policy (*Response*) in C_c is a receive policy that checks that five message parts (*RespParts*) in the response message are signed with an X.509 certificate issued to the principal `"BobsPetShop"`. It yields the predicate:

predicate hasReceivePolicyResponse(*env*:item, *sids*, *rids*:items, *env*′:item) :−
hasSecurityHeader(*env*,*toks*),
tok **in** *toks*, *sig* **in** *toks*, *sid* **in** *sids*,
isX509CertSecret(*sid*,`"BobsPetShop"`,*tok*,_,*cak*),
isX509Token(*tok*,*cak*,`"BobsPetShop"`,`"rsasha1"`,*k*),
hasHeaderFrom(*env*,*from*,_),
hasHeaderRelatesTo(*env*,*related*,_),
hasHeaderMessageId(*env*,*id*,_),
hasHeaderCreated(*env*,*created*,_),
hasBody(*env*,*body*,_),
isSignature(*sig*,`"rsasha1"`,*k*,[*from related id created body*]),
env′ = *env*.

This predicate returns *env*′ = *env* after checking that *env* complies with policy *Response*. (In general, *env*′ may differ from *env*, for instance after decrypting the message body.) To this end, it extracts the message parts corresponding to *RespParts*, then it 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.

The translation function $\mathscr{F}_R$ for receive policies (in the next display) translates a named policy into a script fragment containing a sequence of predicate clauses that check received envelopes for compliance to the policy. It has the same structure as the translation function $\mathscr{F}_S$ for send policies. The first three rules of $\mathscr{F}_R$ compile base assertions and are duals of the first three rules of $\mathscr{F}_S$. The first rule implements the assertion Integrity(Username(U),*Pts*); it verifies a signature based on a username token, after extracting the same message parts that are signed in the first rule of $\mathscr{F}_S$. The second rule corresponds to the predicate hasReceivePolicyResponse detailed above. The third rule decrypts the message body using a private key and returns a modified envelope *env*' by replacing the body with its decrypted version. As in $\mathscr{F}_S$, the last two rules of $\mathscr{F}_R$ compile disjunctions and conjunctions of policies to the corresponding disjunctions and conjunctions of TulaFale predicates. The main differences are in the translation of conjunction: the generated predicate does not need to distribute fresh values and it calls each conjunct in order, whereas the predicate generated by $\mathscr{F}_S$ calls conjuncts in reverse order.

Translation of Receive Policy *Pol* with Name *X* to Clauses: $\mathscr{F}_R(Pol,X)$

$\mathscr{F}_R$(Integrity(Username,*Pts*),*X*) =
$\mathscr{F}_R$(Integrity(Username(*U*),*Pts*),*X*) = {
    predicate hasReceivePolicy*X*(*env*,*sids*,*rids*,*env*′) :−
        hasSecurityHeader(*env*,*toks*),
        tok **in** *toks*, *sig* **in** *toks*, *sid* **in** *sids*,
        isUserPassword(*sid*,*u*,*p*),
        isUserTokenKey (*tok*,*u*,*p*,*n*,*t*,*k*),
        PartFormula(*Pts*)
        isSignature(*sig*,`"hmacsha1"`,*k*,*PartItems*(*Pts*)),
        env′ = *env*. }

$\mathscr{F}_R$(Integrity(X509,*Pts*),*X*) =
$\mathscr{F}_R$(Integrity(X509(*S*),*Pts*),*X*) = {
    predicate hasReceivePolicy*X*(*env*,*sids*,*rids*,*env*′) :−
        hasSecurityHeader(*env*,*toks*),
        tok **in** *toks*, *sig* **in** *toks*, *sid* **in** *sids*,
        isX509CertSecret(*sid*,*S*,*tok*,_,*cak*),
        isX509Token(*tok*,*cak*,*S*,`"rsasha1"`,*k*),
        PartFormula(*Pts*)
        isSignature(*sig*,`"rsasha1"`,*k*,*PartItems*(*Pts*)),
        env′ = *env*. }

$\mathscr{F}_R$(Confidentiality(X509,[Body],*X*) =
$\mathscr{F}_R$(Confidentiality(X509(*S*),[Body],*X*) = {
    predicate hasReceivePolicy*X*(*env*,*sids*,*rids*,*env*′) :−
        hasSecurityHeader(*env*,*toks*),
        tok **in** *toks*, *rid* **in** *rids*,
        isX509CertSecret(*rid*,*S*,*tok*,*sk*,*cak*),
        isX509Token(*tok*,*cak*,*S*,`"rsasha1"`,*k*),
        hasBody(*env*,*b*,*bVal*),
        isCipherValue(*bVal*,`"rsa"`,*sk*,*bDec*),
        bDec = `<Body>`*bDecVal*`</>`,
        replaceBody(*env*,*bDecVal*,*env*′). }

$\mathscr{F}_R$(OneOrMore[Pol_1,...,Pol_n],*X*) = { $\mathscr{F}_R$(Pol_1,*X*) … $\mathscr{F}_R$(Pol_n,*X*) }

$\mathscr{F}_R$(All[Pol_1,...,Pol_n],*X*) = {
    $\mathscr{F}_R$(Pol_1,X_1) … $\mathscr{F}_R$(Pol_n,X_n)
    predicate hasReceivePolicy*X*(env_0,*sids*,*rids*,env_n) :−
        hasReceivePolicyX_1(env_0,*sids*,*rids*,env_1), …
        hasReceivePolicyX_n(env_{n-1},*sids*,*rids*,env_n). }

5.5 Compiling Configurations to Clauses

From a given configuration, we generate clauses that compose policy-enforcement predicates, according to the policy maps listed in the configuration. Each send policy map Send(*Addr*,*Pol*) with policy *Pol* named *X* is translated to a clause of mkConformant that invokes hasSendPolicy*X* when the destination service and action of an envelope match the address *Addr*. For instance, the send policy map in C_c yields the clause:

predicate mkConformant(*env*:item,*sids*, *rids*, *fresh*:items,*env*′:item) :−
  hasHeaderTo(*env*,_,`"http://BobsPetShop.com/service.asmx"`),
  hasHeaderAction(*env*,_,`"http://petshop/premium"`),
  hasSendPolicyRequest(*env*,*sids*,*rids*,*fresh*,*env*′).

This clause first matches the contents of the `<To>` and `<Action>` headers of the source envelope *env* against the values specified in the policy map, and then it calls the predicate hasSendPolicyRequest on the envelope *env* to yield a signed envelope *env*′ that complies with the policy named *Request*.

To enforce each send policy map, Send(*Addr*,*Pol*), the predicate mkConformant may require as many as *FreshVars*(*Pol*) fresh values. Accordingly, to enforce the send policies in a configuration *C*, the translated generic sender process generates and provides a list *fresh* consisting of $\max\{FreshVars(P_i) | \mathrm{Send}(A_i, P_i) \in C\}$ elements.

Conversely, each receive policy map is translated to a clause of isConformant that invokes receive policy predicates for matching service and actions. For instance, the translation of the receive policy map in C_c with Default address simply invokes the predicate hasReceivePolicyResponse:

predicate isConformant(*env*:item, *sids*, *rids*:items, *env*′:item) :−
  hasReceivePolicyResponse(*env*,*sids*,*rids*,*env*′).

In the general case, as defined in the next display, the translation generates a script fragment containing a sequence of predicate clauses, containing clauses for each policy and a clause for each policy map; the predicates mkConformant and isConformant are defined as disjunctions of the clauses for send and receive policy maps, respectively.

5.6 Embedding Security Goals

The configuration compiler finally makes explicit our security goals, formally defined in Section 6. To this end, $\mathscr{S}(C,L)$ includes $\mathscr{A}(L) = \mathscr{G}\ \mathscr{A}_A(L)$, where $\mathscr{G}$ lists our proof obligations and the generated clauses $\mathscr{A}_A(L)$ specialize those obligations for *L*. Crucially, $\mathscr{A}(L)$ does not depend on the configuration *C*, and is coded independently from the lower-level policy maps implemented in our scripts.

Authenticity The processes that represent principals and key generators mark their significant operations, such as the transmission and acceptance of SOAP messages, by logging events. For example, before emitting its envelope on *httpChan*, the generic sender of Section 5.2 logs the event **begin** *Log*(*a*). Similarly, the generic receiver logs **end** *Log*(*a*) to mark its intent to accept an envelope as valid. We state authenticity goals as correspondences between such events. Informally, in the absence of unsafe principals, we would expect that whenever **end** *Log*(*a*) is logged, **begin** *Log*(*a*) has also

Translation of Policy Maps, Configurations to Clauses: $\mathscr{F}(map)$, $\mathscr{F}(Cfg)$

AddrFormula(Default) = {}
AddrFormula(ToDefault(*Suri*)) = { hasHeaderTo(*env*,_,*Suri*), }
AddrFormula(ToAction(*Suri*,*Action*)) = {
    hasHeaderTo(*env*,_,*Suri*),
    hasHeaderAction(*env*,_,*Action*), }
$\mathscr{F}$(Send(*Addr*,Pol_X)) = {
    $\mathscr{F}_S(Pol_X,X)$
    predicate mkConformant(*env*,*sids*,*rids*,*fresh*,env') :−
      AddrFormula(*Addr*)
      hasSendPolicy$X$(*env*,*sids*,*rids*,*fresh*,env'). }
$\mathscr{F}$(Receive(*Addr*,Pol_X)) = {
    $\mathscr{F}_R(Pol_X,X)$
    predicate isConformant(*env*,*sids*,*rids*,env') :−
      AddrFormula(*Addr*)
      hasReceivePolicy$X$(*env*,*sids*,*rids*,env'). }
$\mathscr{F}([Map_1, \ldots, Map_n]) = \{ \mathscr{F}(Map_1) \ldots \mathscr{F}(Map_n) \}$

been logged with matching arguments a. The arguments a can be used to correlate our intentions with the envelope contents; they are automatically extracted from envelopes, using a predicate linkAssert that is compiled from link descriptions.

Before dealing with the general case, we illustrate the definition of linkAssert for our example link specification: L_0 yields two clauses, for marking requests and responses, respectively. We detail the clause for requests:

predicate linkAssert(*sid*, *rid*:item, *env*:item, *a*:items) :−
  hasHeaderTo(*env*,_,*to*),
  hasHeaderAction(*env*,_,*action*),
  hasHeaderMessageId(*env*,_,*id*),
  hasHeaderCreated(*env*,_,*t*),
  hasCleartextBody(*env*,_,*body*),
  hasName(*sid*,*sender*),
  to = `"http://BobsPetShop.com/service.asmx"`,
  action = `"http://petshop/premium"`,
  a = [`"Request"` `"Clear"` *sender to action id t body*].

In this clause, the first six predicates extract data relevant to the request: elements from the source envelope *env*, and the name of the sending principal *sid*. The predicate hasCleartextBody extracts the message body and also checks that it is unencrypted. (As our model supports nested encryption, this check prevents any ambiguity on the intended message body, as discussed in Section 2.5.) The tests on *to* and *action* check the values found in the envelope against those specified in the link. The resulting list a summarizes the message; it includes tags marking *env* as an unencrypted request, the sender name *sender*, and data extracted from *env*.

The linkAssert predicate may be called both by the generic sender, to record its genuine intent to send *env* on behalf of *sender*, and by the generic receiver, to record its intent to accept *env* as an authentic envelope from *sender*. The (omitted) linkAssert

Translation of a Link L to Clauses used in Logged Events: $\mathscr{A}_A(L)$

PartVal(Body) = *bVal*
PartVal(Header("*Tag*")) = *TagVal*
PartVal([Pt_1,...,Pt_n]) = { [*PartVal*(Pt_1) ... *PartVal*(Pt_n)] }

NameFormula(*E*,Any,*N*) = { hasName(*E*,*N*), }
NameFormula(*E*,[P_1, ..., P_n],*N*) = { hasName(*E*,*N*), *N* **in** [P_1 ... P_n], }

PrinFormula(*Senders*,*Receivers*,Clear) = {
  NameFormula(*sid*,*Senders*,*sn*)
  principals = [*sn*], }
PrinFormula(*Senders*,*Receivers*,Encrypted) = {
  NameFormula(*sid*,*Senders*,*sn*)
  NameFormula(*rid*,*Receivers*,*rn*)
  principals = [*sn rn*], }

$\mathscr{A}_A$(*Suri*,*Actions*,*Clients*,*Services*,*Secrecy*) = {
  predicate linkAssert(*sid*,*rid*:item,*env*:item,*a*:items) :−
    PartFormula(*ReqParts*)
    toVal = *Suri*,
    actionVal **in** *Actions*,
    PrinFormula(*Clients*,*Services*,*Secrecy*)
    a = ["Request" *Secrecy* @ *principals* @ *PartVal*(*ReqParts*)].
  predicate linkAssert(*sid*,*rid*:item,*env*:item,*a*:items) :−
    PartFormula(*RespParts*)
    fromVal = *Suri*,
    PrinFormula(*Services*,*Clients*,*Secrecy*)
    a = ["Response" *Secrecy* @ *principals* @ *PartVal*(*RespParts*)]. }

$\mathscr{A}_A$([L_1, ..., L_n]) = { $\mathscr{A}_A(L_1)$ ... $\mathscr{A}_A(L_n)$ }

clause for responses is similar to the one detailed above, except that it collects the parts *RespParts* from the envelope and checks that the URI and the principal name for the service coincide with those specified in L_0.

In general, we generate a pair of linkAssert clauses for each link, using the rules of the next display. These clauses define the precise structure of the events logged by senders and receivers. These rules rely on auxiliary functions: *PartFormula* (defined in Section 5.4) extracts selected elements from the envelope. *PartVal* collects the variables bound to the content of these elements. *NameFormula* binds principal names for each role, with two cases: when the link specifies a restrictive list of principal names, the name obtained from the entry used to process the envelope (*sid* or *rid*) must also appear in that list. *PrinFormula* prepares the names to be recorded in the event, with two cases: when some parts are both encrypted and signed, the intended recipient of the message is also authenticated, hence both the sender and receiver names are recorded; otherwise, only the sender name is recorded.

Confidentiality Recall that our generic sender takes envelopes from the attacker and sends them out after adding some security headers and (optionally) encrypting its mes-

sage body. If a link requires secrecy, then the message content must be protected from the attacker. To model this, whenever the generic SOAP sender is asked to send an envelope, it first calls a predicate secretEnvelope that may replace the body supplied by the attacker by a single value B. Informally, our confidentiality goal is that, provided the value B is initially secret, the attacker cannot obtain B, in spite of triggering and observing series of messages carrying B on encrypted links.

The predicate secretEnvelope is defined by two clauses, as part of `library.tf` [9]. Both clauses are enabled for any input envelope. The first clause replaces the body of the envelope with a body containing B in cleartext. The second clause passes the envelope unchanged to policy enforcement, including the (arbitrary) envelope body provided by the attacker—otherwise, our confidentiality goal would weaken our authenticity goal.

predicate secretEnvelope(*env*, *B*, *env*′: item) :− *replaceCleartextBody*(*env*,*B*,*env*′).
predicate secretEnvelope(*env*, *B*, *env*′: item) :− *env*′ = *env*.

6 Verifying Link-Based Scripts

We are now ready to state and prove security properties for our scripts. We first formally define the security goals for scripts generated by our configuration compiler. We then establish two kinds of security results.

First, given an arbitrary policy configuration C, we can check whether it guarantees the security goals of a link specification L, by verifying the script $\mathscr{S}(C,L)$. We demonstrate this method on three examples, two of which guarantee the security goals (Theorems 1 and 2), whereas the third does not. The analysis is fully automated and makes use of the TulaFale tool and ProVerif theorem prover.

Second, we show that every policy configuration $\mathscr{C}(L)$ computed by our configuration generator guarantees the security goals of the original link specification L. We establish this correctness result for the generator by proving that all scripts of the form $\mathscr{S}(\mathscr{C}(L),L)$ satisfy their security goals (Theorem 3). Moreover, we show that policy configurations composed of some link-generated components and other arbitrary components still guarantee some security goals (Theorems 4 and 5). The proofs of these results are beyond the reach of automated tools because they express properties about an unlimited number of configurations.

Our proofs use several novel techniques that are of independent interest. Lemma 1 introduces a notion of logical refinement that enables logical abstraction to be applied to a script before automated verification. This powerful technique can be used to elide irrelevant detail to make verification feasible, and is applicable to any TulaFale or ProVerif script. We use this lemma extensively in our proofs; for example, Lemma 4 demonstrates a single, automatically verified script that abstractly represents every link-generated configuration. A second technique, used in the proof of Lemma 6 enables the verification of a script where a process contains a loop that may execute an arbitrary number of times before returning. With each iteration of the loop, the process may create fresh values and use them to create bigger messages. ProVerif fails to prove properties about such examples; we show how to use process equivalences and redun-

dant events to chain together several correspondences to prove our security goals. This technique is applicable to other ProVerif scripts containing these kinds of loops.

6.1 Security Properties, Formally

In our formal statements, scripts *S* are treated as processes, with fixed definitions for terms and predicates and a reduction-based semantics for processes adapted from the applied pi calculus [14]. Automated verification in TulaFale safely approximates the semantics of *S* running in parallel with an *attacker process*, that is, an arbitrary process that logs no events and where the name *B* does not occur. Our target properties for authentication, secrecy, and adequacy are as follows.

Authentication (Robust Safety):

A script *S* is *robustly safe* when, for any run with any attacker process, if **end** *Log*([*tag secr sender* @ *contents*]) occurs, then either **begin** *Log*([*tag secr sender* @ *contents*]) or **begin** *Leak*(*sender*) previously occurred.

The definition requires that a correspondence holds between **begin** and **end** events, for any terms *tag*, *secr*, *sender*, and *contents*. (In all our scripts, *tag* is either `"Request"` or `"Response"`, and *secr* is either `"Clear"` or `"Encrypted"`.) Our correspondence property reflects our model of principals, with possibly compromised sender principals. In a simpler model with no insider attacks (obtained for instance by restricting channels *genLeakUPChan* and *genLeakXChan* of Section 5.1), we would define authentication with a simpler correspondence property [37, 22]: for any term *a*, if **end** *Log*(*a*) occurs, then **begin** *Log*(*a*) previously occurred.

Secrecy:

The attacker obtains B when the attacker process can send the message *B* on a public channel. A script *S preserves secrecy* when, for any run with any attacker process, if the attacker obtains *B*, then either **begin** *Log*([_ `"Clear"` _ _ _ _ _ *B*]) previously occurred, or **begin** *Log*([_ `"Encrypted"` _ *receiver* _ _ _ _ *B*]) and **begin** *Leak*(*receiver*) previously occurred.

Hence, our notion of secrecy focuses on a single secret *B* potentially used as a message body in any envelope. The attacker may obtain *B* in only two cases: *B* is sent on a cleartext link; or *B* is sent to an unsafe principal (as the attacker may use compromised keys for decryption). In other words, the attacker cannot obtain message bodies that are sent to safe principals over encrypted links.

Functional Adequacy:

A script *S* is *adequate for a* when, for some run in some context, **end** *Log*(*a*) occurs.

Adequacy is a functional requirement used to provide assurance that our scripts can successfully send and receive messages. It is defined as a reachability property for **end**

events with some particular contents. We typically check adequacy for a few common values of a, for requests and responses, with or without encryption:

$aReq_0$ = ["Request" "Clear" "Alice" ...]
$aResp_0$ =["Response" "Clear" "BobsPetShop" ...]
$aReq_1$ =["Request" "Encrypted" "Alice" "BobsPetShop" ...]
$aResp_1$ =["Response" "Encrypted" "BobsPetShop" "Alice" ...]

6.2 Verifying Example Configurations

Recall that $\mathscr{C}(L)$ is the policy configuration generated from a link specification L, and that $\mathscr{S}(C,L)$ is the TulaFale script (essentially a pi calculus process, with some embedded assertions) compiled from a policy configuration C and a link specification L. Hence, $C_0 = \mathscr{C}(L_0)$ and $S_0 = \mathscr{S}(C_0, L_0)$ are the policy configuration and pi calculus process, respectively, for our example link.

Continuing with our running example, C_0 and L_0, our first theorem illustrates the verification of a policy configuration against a link specification.

Theorem 1 *Let $S_0 = \mathscr{S}(C_0, L_0)$. We have (1) S_0 is robustly safe; (2) S_0 is adequate for values of the forms $aReq_0$ and $aResp_0$.*

Proof: By verification of script `s0.tf` derived as $\mathscr{S}(C_0, L_0)$. □

In this paper, all scripts are verified by invoking TulaFale then ProVerif. Hence, our proofs rely on the correctness of the algorithms underlying these tools [9, 13] as well as the correctness of their implementation. On a workstation with two 3.6 GHz Pentium 4 processors and 2 GB of memory, verification for all the goals stated in Theorem 1 succeeds in a few minutes.

In addition to the properties listed in Theorem 1, our next theorem also states that the policy configuration C_1 generated from L_1 preserves the secrecy goals of L_1. (Technically, C_0 also trivially preserves secrecy, using the first case of the definition.)

Theorem 2 *Let $S_1 = \mathscr{S}(C_1, L_1)$. We have (1) S_1 is robustly safe; (2) S_1 preserves secrecy; (3) S_1 is adequate for values of the forms $aReq_1$ and $aResp_1$.*

Proof: By verification of script `s1.tf` derived as $\mathscr{S}(C_1, L_1)$. □

Conversely, if a configuration has a weak policy for an endpoint, the link specification may be violated. For example, the following configuration C_2 modifies C_0 by adding a new receive policy map for `CheapService.asmx` with a policy that requires signatures only on the body of the message:

Target′ = ToAction("http://BobsPetShop.com/CheapService.asmx",
"http://petshop/regular")
$C_2 = C_0$ @
[Send(*Target′*, Integrity(Username, *ReqParts*)),
Receive(*Target′*, Integrity(Username, [Body]))]

(In contrast, the send policy still requires a more extensive signature.) The link specification for this new configuration has two links:

L_2 = [("http://BobsPetShop.com/service.asmx",
["http://petshop/premium"], Any, ["BobsPetShop"], Clear),
("http://BobsPetShop.com/CheapService.asmx",
["http://petshop/regular"], Any, ["BobsPetShop"], Clear)]

Let S_2 be the script $\mathscr{S}(C_2,L_2)$ (appearing in file `s2.tf` [9]). When we try to automatically verify S_2, the proof fails for L_2, and ProVerif returns a counter-example: the attacker can modify the destination of the messages to mount a redirection attack and, since the receive policy does not check that the addressing headers are signed, this attack succeeds. (Experimentally, a similar attack succeeds against WSE configured with C_2.) Conversely, the configuration C_2 is safe for the original service: $\mathscr{S}(C_2,L_0)$ is automatically verified as robustly safe.

6.3 Validating All Link-Generated Configurations

We now extend our automated proof technique to general classes of policy configurations and link specifications. We present three results in this direction, Theorems 3, 4, and 5. Their proofs are more complex than the results in Section 6.2; they involve auxiliary scripts and (human-proved) lemmas. The automated parts of the proofs of Theorem 3 and Theorem 4 take several hours.

The following theorem validates our intent that *all* configurations generated from link specifications be safe:

Theorem 3 *For any link specification L, let $S = \mathscr{S}(\mathscr{C}(L),L)$. We have (1) S is robustly safe; (2) S preserves secrecy.*

The proof mixes automated verification of scripts with manual arguments. We first recall a proof technique for relating the properties of scripts that differ in their definitions of predicates:

Lemma 1 (Logical Refinement) *We say that a script S logically refines S' when (1) S and S' differ only in their predicate clauses, and (2) all instances of formulas appearing in* **filter***s that hold with the clauses of S also hold with the clauses of S'.*

Logical refinement preserves all safety properties, including our notions of authentication and secrecy.

Intuitively, S' has more traces than S, so if S' is secure, so is S. The proof of Lemma 1 follows from the operational semantics of filters; it is detailed in [4] for a similar process calculus.

In preparation for logical refinement, we define a small variant $\mathscr{S}'(\mathscr{C}(L),L)$ of the script $\mathscr{S}(\mathscr{C}(L)),L)$ by rearranging filters in *GenericSender* and *GenericReceiver*. For any link specification L, the script $\mathscr{S}'(\mathscr{C}(L),L)$ (on the right) differs from $\mathscr{S}(\mathscr{C}(L),L)$ (on the left) only as follows:

process *GenericSender*() = ... **filter** linkAssert(*sid*,*rid*,*envB*,*a*) $\rightarrow a$; **begin** *Log*(*a*); **filter** mkConformant (*envB*,[*sid*],[*rid*],[*n t*],*env*′) $\rightarrow$ *env*′; **out** *httpChan*(*env*′).	**process** *GenericSender*() = ... **filter** mkConformantAssert (*envB*,*sid*,*rid*,[*n t*],*a*,*env*′) $\rightarrow$ *a*,*env*′; **begin** *Log*(*a*); **out** *httpChan*(*env*′).
process *GenericReceiver*() = ... **filter** isConformant (*env*,[*sid*],[*rid*],*env*′) $\rightarrow$ *env*′; **filter** linkAssert(*sid*,*rid*,*env*′,*a*) $\rightarrow a$; **end** *Log*(*a*).	**process** *GenericReceiver*() = ... **filter** isConformantAssert (*envB*,*sid*,*rid*,*env*′,*a*) $\rightarrow$ *env*′,*a*; **end** *Log*(*a*).

where the new predicate mkConformantAssert conjoins linkAssert and mkConformant, and the new predicate isConformantAssert conjoins isConformant and linkAssert:

predicate mkConformantAssert(*env*,*sid*,*rid*:item,*fresh*,*a*:items,*env*′:item) :−
linkAssert(*sid*,*rid*,*env*,*a*), mkConformant(*env*,[*sid*],[*rid*],*fresh*,*env*′).
predicate isConformantAssert(*env*,*sid*,*rid*,*env*′:item,*a*:items) :−
isConformant(*env*,*sid*,*rid*,*env*′), linkAssert(*sid*,*rid*,*env*′,*a*).

Lemma 2 *If $\mathscr{S}'(\mathscr{C}(L),L)$ is robustly safe and preserves secrecy, so does $\mathscr{S}(\mathscr{C}(L),L)$.*

Proof: From any given run of $\mathscr{S}(\mathscr{C}(L),L)$ in any context, we build a similar run for $\mathscr{S}'(\mathscr{C}(L),L)$. By construction, the two processes perform matching reduction steps in any context, except when evaluating the modified filters. By definition of the new predicates, the two filters before the change both succeed with results *a* and *env*′ if and only if the new filter succeeds with the same results. If both filters succeed, they trigger the same process (using one less reduction step after the change). Similarly, if a first filter fails, the corresponding single filter also fails. If only a second filter fails, the corresponding single filter also fails, with the same effect, except for an additional **begin** *Log*(*a*), triggered with $\mathscr{S}(\mathscr{C}(L),L)$ but not with $\mathscr{S}'(\mathscr{C}(L),L)$. This additional event plays no role in the rest of the run.

Hence, the resulting process with the modified script is the same as with the original script, except that it may have less begin events. In particular, by definition of robust safety and secrecy, if $\mathscr{S}'(\mathscr{C}(L),L)$ is robustly safe and preserves secrecy, so does $\mathscr{S}(\mathscr{C}(L),L)$. □

The proof of Theorem 3 proceeds by exhibiting a single script *S* that is logically refined by $\mathscr{S}'(\mathscr{C}(L),L)$ for any *L*. We outline the content of *S*, and refer to the script `linkgen.tf` [9] for the complete definition. The differences between *S* and $\mathscr{S}'(\mathscr{C}(L),L)$ are as follows:

- For policies, instead of link-generated predicates, *S* has eight predicates implementing every combination of send and receive policies, for requests and responses, on encrypted and cleartext links. In contrast with link-generated predicates, these predicates do not check any principal names. For instance, the pred-

icate hasSendPolicyRequestE implements send policies for request messages on encrypted links.

- For policy maps, S has four predicates implementing send and receive policy maps for encrypted and cleartext links. In contrast with mkConformant and isConformant in $\mathscr{S}'(\mathscr{C}(L),L)$, these predicates do not check any service addresses or actions. For instance, the predicate isConformantC implements receive policy maps for cleartext links.
- For link assertions, S has two predicates, linkAssertC and linkAssertE, implementing assertions for encrypted and cleartext links. In contrast with linkAssert in $\mathscr{S}'(\mathscr{C}(L),L)$, these predicates do not check any principal names, services, or actions.
- Finally, S redefines the predicates mkConformantAssert and isConformantAssert of $\mathscr{S}'(\mathscr{C}(L),L)$ so that they combine the new predicates for policy maps and link assertions, with one clause for each choice of encryption:

  predicate mkConformantAssert(*env*,*sid*,*rid*:item,*fresh*,*a*:items,*env*′:item) :−
  linkAssertE(*sid*,*rid*,*env*,*a*), mkConformantE(*env*,[*sid*],[*rid*],*fresh*,*env*′).
  predicate mkConformantAssert(*env*,*sid*,*rid*:item,*fresh*,*a*:items,*env*′:item) :−
  linkAssertC(*sid*,*rid*,*env*,*a*), mkConformantC(*env*,[*sid*],[*rid*],*fresh*,*env*′).
  predicate isConformantAssert(*env*,*sid*,*rid*,*env*′:item,*a*:items) :−
  isConformantE(*env*,[*sid*],[*rid*],*env*′), linkAssertE(*sid*,*rid*,*env*′,*a*).
  predicate isConformantAssert(*env*,*sid*,*rid*,*env*′:item,*a*:items) :−
  isConformantC(*env*,[*sid*],[*rid*],*env*′), linkAssertC(*sid*,*rid*,*env*′,*a*).

We first verify that S meets the security goals of Theorem 3:

Lemma 3 *S is robustly safe and preserves secrecy.*

Proof: By verification of script `linkgen.tf`. □

Next, we show that, for every link L, the script $\mathscr{S}'(\mathscr{C}(L),L)$ logically refines S. A link-generated configuration $\mathscr{C}(L)$ consists of a set of send and receive policy maps: for each link (*Suri*, [ac_1,…,ac_m], [cp_1,…,cp_n], [sp_1,…,sp_o], Secr), it contains, on the service-side, one send and one receive policy map for each (ac_i, cp_j, sp_k) and, on the client-side, one send and one receive policy map for each (cp_j, sp_k). The policies themselves are of four kinds, request and response policies for clear and encrypted links, differing only in the names of the sending and receiving principals. Hence, the script generated from this configuration and the original link consists of predicates that fall under a few generic patterns and differ mainly in names of principals and service addresses. By deleting the clauses corresponding to addresses and principal names, we obtain the more abstract script S.

Lemma 4 *For any link specification L, the script $\mathscr{S}'(\mathscr{C}(L),L)$ logically refines S.*

Proof: We first successively establish logical refinements for policies, policy maps, and link assertions:

(1) For policies, we show that every link-generated predicate of $\mathscr{S}'(\mathscr{C}(L),L)$ refines one of the eight policy predicates in S. For instance, let P be the policy *LinkPol*(*Ctok*,*Stok*,Encrypted,*ReqParts*), for some *Ctok*, *Stok*. By comparing their clauses, we easily check that, whenever the predicate generated by $\mathscr{F}_S(L)$ holds, so does hasSendPolicyRequestE.

(2) For policy maps, we show that, whenever the link-generated predicate mkConformant holds, either mkConformantC holds or mkConformantE holds. Similarly, whenever the link-generated predicate isConformant holds, either isConformantC holds or isConformantE holds. For instance, consider the receive policy map $M =$ *Recv*(*Addr*,*LinkPol*(*Ctok*,*Stok*,Clear,*RespParts*)), for some *Addr*, *Ctok*, and *Stok*. We show that if the clause generated by $\mathscr{F}(M)$ holds, so does isConformantC.

(3) For link assertions, we show that, whenever the link-generated predicate linkAssert holds, either linkAssertC holds or linkAssertE holds. For instance, let L be any cleartext link. We show that if any of the clauses for linkAssert in $\mathscr{A}_A(L)$ holds, so does linkAssertC.

Each of these refinements is established by unfolding predicate definitions and applying simple rewrite rules: formula F refines G if G can be obtained by (a) deleting some clauses from F, by (b) replacing a variable in F by _, or by using (c) the disjunction $F_1 \ldots F_n$ refines G if each F_i refines G.

Next, we show that the predicates mkConformantAssert and isConformantAssert in $\mathscr{S}'(\mathscr{C}(L),L)$ logically refine the corresponding predicates in S. These refinements rely on an additional property of link specifications. Our link specifications have at most one link for each service URI. So, every envelope *envB* can be mapped to either an encrypted link or a cleartext link, but not both. If *envB* is mapped to an encrypted link, then the matching clauses of mkConformant refine mkConformantE and the matching clauses of linkAssert refine linkAssertE. If *envB* is mapped to a cleartext link, then the matching clauses of mkConformant refine mkConformantC and the matching clauses of linkAssert refine linkAssertC. By combining these two refinements, we establish that the conjunction of linkAssert(*sid*,*rid*,*envB*,*a*) and mkConformant(*envB*,[*sid*],[*rid*],[*n t*],*env*′) in $\mathscr{S}'(\mathscr{C}(L),L)$ refines mkConformantAssert(*envB*,*sid*,*rid*,[*n t*],*a*,*env*′) in S. By a similar argument, we show the logical refinement for isConformantAssert.

We conclude by induction on L. □

Proof of Theorem 3: For a given L, we apply Lemma 1 with the logical refinement of Lemma 4 to show that the properties proved for S in Lemma 3 also hold for $\mathscr{S}'(\mathscr{C}(L),L)$. Finally, we apply Lemma 2 to establish these properties for $\mathscr{S}(\mathscr{C}(L),L)$. □

6.4 Validating Partial Configurations

In this section, we present two theorems regarding configurations where only some policies and policy maps are available for analysis. Even though some parts of the configuration are unknown, these results rely on simplifying assumptions on the behaviours of senders and receivers:

- Even if a sender's policy maps are unknown, it is assumed that the sender still enforces some policy on outgoing messages. In particular, a sender may not send a client password or secret key in the clear.

- A sender may use only one sending and one receiving principal identity to enforce its policy.

Our next theorem covers the case where only the receive policies are known, whereas sending principals implement some arbitrary, unknown policies. For instance, a sending principal may use its signing keys to generate multiple signatures over arbitrary message parts.

Theorem 4 (Safety of Link-Generated Receive Configurations) *For any link specification L and configuration C such that C and $\mathscr{C}(L)$ have the same* Receive *policy maps, the process $\mathscr{S}(C,L)$ is robustly safe.*

That is, receive policies suffice to validate link specifications as regards authentication properties, irrespective of the send policies used to generate envelopes. (Clearly, we cannot hope to retain functional adequacy, as for instance a weak send policy may yield envelopes with signatures rejected by the receiver.)

Dually to Theorem 4, our next result is on the secrecy of configurations where only the send policies are known:

Theorem 5 (Safety of Link-Generated Send Configurations) *For any link specification L and configuration C such that C and $\mathscr{C}(L)$ have the same* Send *policy maps, the process $\mathscr{S}(C,L)$ preserves the secrecy of B.*

6.5 Proof of Theorem 4

For the proofs in this section, we rely on a variant of robust safety. We say that a script is *robustly link-safe* when, for any run with any attacker process:

- if **end** *Log*([*tag* `"Clear"`*sender* @ *contents*]) occurs, then either
  begin *Log*([*tag* `"Clear"`*sender* @ *contents*]), or
  begin *Log*([*tag* `"Encrypted"`*sender receiver* @ *contents*]), or
  begin *Leak*(*sender*) previously occurred; and

- if **end** *Log*([*tag* `"Encrypted"`*sender receiver* @ *contents*]) occurs, then either
  begin *Log*([*tag* `"Encrypted"`*sender receiver* @ *contents*]), or
  begin *Log*([*tag* `"Clear"`*sender* @ *contents*]), or
  begin *Leak*(*sender*) previously occurred.

Intuitively, robust link-safety allows the sender and receiver to disagree on whether a link is cleartext or encrypted. This notion is weaker than robust safety, and is also easier to verify automatically. Nonetheless, the two notions coincide for all link specifications:

Lemma 5 *If $\mathscr{S}(C,L)$ is robustly link-safe, then it is also robustly safe.*

Proof: We use the global assumption that link specifications have at most one link for each service URI. By inspection of the linkAssert clauses generated from links, we obtain that, for any given *Suri*, the runs of $\mathscr{S}(C,L)|O$ may have events either of the form **begin** *Log*([_ "Clear" _ *Suri* @ _]) and **end** *Log*([_ "Clear" _ *Suri* @ _]), or of the form **begin** *Log*([_ "Encrypted" _ _ *Suri* @ _]) and **end** *Log*([_ "Encrypted" _ _ *Suri* @ _]), but not both.

Suppose that $\mathscr{S}(C,L)|O \to^* T \Downarrow$ **end** *L*([*tag secr s* @ *a*]). We show that **begin** *Log*([*tag secr s* @ *a*]) or **begin** *Leak*(*s*) previously occurred. If *secr* = "Clear", then by robust link-safety of $\mathscr{S}(C,L)$ we have either **begin** *Log*([*tag* "Clear"*s* @ *a*]), or **begin** *Log*([*tag* "Encrypted"*s r* @ *a*]), or **begin** *Leak*(*s*) must have previously occurred. Since **end** *L*([*tag* "Clear"*s* @ *a*] and **begin** *Log*([*tag* "Encrypted"*s r* @ *a*]) cannot both occur in the same run, the second disjunct is eliminated. Similarly, if *secr* = "Encrypted", then either **begin** *Log*([*tag* "Encrypted"*s* @ *a*]), or **begin** *Leak*(*s*) must have previously occurred. □

The proof of Theorem 4 relies on an auxiliary script S^ω that lets the attacker implement arbitrary send policies. In this proof, we let $\mathscr{C}_R(L)$ range over configurations with the same Receive policy maps as in L. We automatically verify S^ω (Lemma 6), then we complete the proof by showing that, if S^ω is robustly link-safe, so is $\mathscr{S}(\mathscr{C}_R(L),L)$ for any L and $\mathscr{C}_R(L)$. To this end, we show that there exists a script S^n, for some n that depends on $\mathscr{C}_R(L)$, such that $\mathscr{S}(\mathscr{C}_R(L),L)$ logically refines S^n (Lemma 9) and that, if S^ω is robustly link-safe, so is S^n for any n (Lemma 12).

A Generic-Policy Sender The script S^ω models a configuration where receivers implement link-generated policies, much as in the script S used in the proof of Theorem 3, and senders allow the attacker to implement an arbitrary sequence of base policy assertions on the outgoing envelope. The definition of S^ω appears in `partialsend.tf` [9]. Hence, S^ω differs from S mainly in the definition of the process *GenericSender*:

```
private channel loopChan(item,item,item).
process GenericSender() =
 (!in initChan(env);
   in dbChan(sid); in dbChan(rid);
   new freshid:string;
   filter freshId (env,freshid,envId) → envId;
   filter secretEnvelope (env,B,envB) → envB ;
   filter linkAssert(sid,rid,envB,a) → a ;
   begin Log (a);
   out loopChan(sid,rid,envB)) |
 (!in loopChan(sid,rid,envB);
   new n:string; new t:string;
   filter hasSendPolicyA(envB,[sid],[rid],[n t],env′) → env′ ;
   out loopChan(sid,rid,env′)) |
 (!in loopChan(sid,rid,env′);
   out httpChan(env′))
```

This process (written G^ω in the proof) simulates the behaviour of a configuration with any send policies written as a conjunction of base policy assertions, as follows. It uses a new private channel *loopChan*.

The first sub-process starts much like the *GenericSender* process of Section 5.2—replacing the message identifier by a fresh value, possibly replacing the body with a secret value, and performing a begin assertion computed by linkAssert. Then, instead of invoking mkConformant with identities *sid* and *rid* on the envelope *envB*, it sends these three parameters on channel *loopChan*.

The second sub-process repeatedly inputs these parameters then invokes the predicate hasSendPolicyA (explained below) that enforces one arbitrary base policy assertion on this envelope. The resulting envelope env' is sent back on channel *loopChan*, with the same identities. Hence, this sub-process implements an iteration of a loop that applies a sequence of base assertions on the envelope.

The third sub-process also inputs these parameters from *loopChan*, then forwards the envelope onto channel *httpChan*, thus causing the loop to terminate at any point.

The process G^{ω} also uses hasSendPolicyA instead of mkConformant. The predicate hasSendPolicyA (defined in `partialsend.tf` [9]) implements a single, generic base policy assertion; it does not depend on any specific set of message parts or principal names and it does not implement conjunctions or disjunctions.

Before showing that, despite these differences, S^{ω} preserves all the runs of $\mathscr{S}(C,L)$, we verify that S^{ω} meets the security goals of Theorem 4:

Lemma 6 *S^{ω} is robustly link-safe.*

Proof: By verification of scripts `partialsend1.tf` and `partialsend2.tf` [9].

The script `partialsend1.tf` is obtained from S^{ω} (`partialsend.tf`) by annotating the middle processes in G^{ω} with additional begin events (indented below):

```
...
(!in loopChan(sid,rid,envB);
        filter clearLinkAssert(sid,rid,envB,preC) →preC;
        filter encLinkAssert(sid,rid,envB,preE) →preE;
        begin Log1c(preC); begin Log1e(preE);
   new n:string; new t:string;
   filter hasSendPolicyA(envB,[sid],[rid],[n t],env′) →env′ ;
        filter clearLinkAssert(sid,rid,env′,postC) →postC;
        filter encLinkAssert(sid,rid,env′,postE) →postE;
        begin Log2c(postC); begin Log2e(postE);
   out loopChan(sid,rid,env′)) |
...
```

where clearLinkAssert and encLinkAssert are auxiliary predicates that succeed for all inputs *sid*, *rid*, and *env*. Since the variables bound by these predicates occur only in the contents of auxiliary begin events, `partialsend1.tf` and S^{ω} have the same runs, except for those auxiliary begin events. We automatically verify that the events for any run of `partialsend1.tf` with any attacker process have the following properties:

- if **begin** *Log1e*([*tag* `"Encrypted"`*s r* @ *a*]) occurs,
  then **begin** *Log2e*([*tag* `"Encrypted"`*s r* @ *a*]) previously occurred;
- if **begin** *Log2e*([*tag* `"Encrypted"`*s r* @ *a*]) occurs,
  then either **begin** *Log1e*([*tag* `"Encrypted"`*s r* @ *a*])
  or **begin** *Log1c*([*tag* `"Clear"`*s* @ *a*]) previously occurred;

- if **begin** *Log2c*([*tag* `"Clear"`*s* @ *a*]) occurs,
  then **begin** *Log1c*([*tag* `"Clear"`*s* @ *a*]) previously occurred; and
- if **begin** *Log1c*([*tag* `"Clear"`*s* @ *a*]) occurs,
  then either **begin** *Log2c*([*tag* `"Clear"`*s* @ *a*]),
  or **begin** *Log*([*tag* `"Clear"`*s* @ *a*]),
  or **begin** *Log*([*tag* `"Encrypted"`*s r* @ *a*]) previously occurred.

By composing these properties, we establish that, for any given run,

(1) if **begin** *Log1c*([*tag* `"Clear"`*s* @ *a*]) occurs,
then either **begin** *Log*([*tag* `"Clear"`*s* @ *a*])
or **begin** *Log*([*tag* `"Encrypted"`*s r* @ *a*]) previously occurred.

(2) if **begin** *Log1e*([*tag* `"Encrypted"`*s r* @ *a*]) occurs,
then either **begin** *Log*([*tag* `"Encrypted"`*s r* @ *a*])
or **begin** *Log*([*tag* `"Clear"`*s* @ *a*]) previously occurred.

The script `partialsend2.tf` [9] is obtained from `partialsend1.tf` as follows: *loopChan* becomes a public channel, allowing the attacker to send and receive messages, hence to effectively control the loop. In addition, *GenericSender* no longer relies on *loopChan* to convey *sid* or *rid* and instead inputs them directly from *dbChan* when needed. So all inputs on *loopChan* ignore *sid*, *rid* and all outputs send (empty) constants in the place of *sid*, *rid*. Hence, `partialsend2.tf` defines the process

```
channel loopChan(item,item,item).
process GenericSender() =
 (!in initChan(env);
   in dbChan(sid); in dbChan(rid);
   new freshid:string;
   filter freshId (env,freshid,envId) → envId;
   filter secretEnvelope (env,B,envB) → envB ;
   filter linkAssert(sid,rid,envB,a) → a ;
   begin Log (a);
   out loopChan("","",envB)) |
 (!in loopChan(_,_,envB);
   in dbChan(sid); in dbChan(rid);
   new n:string; new t:string;
         filter clearLinkAssert(sid,rid,envB,preC) → preC;
         filter encLinkAssert(sid,rid,envB,preE) → preE;
         begin Log1c(preC); begin Log1e(preE);
   filter hasSendPolicyA(envB,[sid],[rid],[n t],env′) → env′ ;
         filter clearLinkAssert(sid,rid,env′,postC) → postC;
         filter encLinkAssert(sid,rid,env′,postE) → postE;
         begin Log2c(postC); begin Log2e(postE);
   out loopChan("","",env′)) |
 (!in loopChan(_,_,env′);
   out httpChan(env′))
```

Both modifications enable more runs, but we can show that they preserve all previous runs of `partialsend1.tf`. By inspection of S^{ω}, the channel *dbChan* is private, is

never communicated, and all its outputs are replicated. By automated verification of `partialsend1.tf`, in any run with any attacker process, every **begin** *Log1c*(*preC*) is preceded by some **begin** *Log*(*a*). So, every **begin** *Log1c*(*preE*) (in the second sub-process) is preceded by a successful **in** *dbChan*(*sid*); **in** *dbChan*(*rid*); (in the first sub-process). Hence, since all outputs on *dbChan* are replicated, inserting **in** *dbChan*(*sid*); **in** *dbChan*(*rid*); in the second sub-process is safe, since these inputs can succeed with the same identities *sid*, *rid*.

By automated verification of `partialsend2.tf`, we obtain that in any run with any attacker process,

(3) if **end** *Log*([*tag* `"Encrypted"`*s r* @ *a*]) occurs,
then either **begin** *Log1e*([*tag* `"Encrypted"`*s r* @ *a*])
or **begin** *Leak*(*s*) previously occurred; and

(4) if **end** *Log*([*tag* `"Clear"`*s* @ *a*]) occurs,
then either **begin** *Log1c*([*tag* `"Clear"`*s* @ *a*])
or **begin** *Leak*(*s*) previously occurred.

Robust link-safety for S^{ω} follows from the combination of Properties (1)–(4). □

A Generic Base Assertion Predicate Recall that a base policy assertion requires either that the message contain a signature over a fixed number of message parts or that the message body be encrypted, and also specifies the token to use. More abstractly, we define a predicate hasSendPolicyA that simulates any base assertion but ignores its token type, principal, and specific message parts:

predicate hasSendPolicyA(*env*:item, *sids*,*rids*,*fresh*:items, *env′*:item) :−
  sid **in** *sids*,
  isUserPassword(*sid*,*u*,*p*),
  fresh = [*n t* @ _],
  mkUserTokenKey (*tok*,*u*,*p*,*n*,*t*,*k*),
  getParts(*env*,*parts*),
  mkSignature(*sig*,`"hmacsha1"`,*k*,*parts*),
  replaceSecurityHeader(*env*,[*tok sig*],*env′*).

predicate hasSendPolicyA(*env*:item, *sids*,*rids*,*fresh*:items, *env′*:item) :−
  sid **in** *sids*,
  isX509CertSecret(*sid*,*u*,*tok*,*sk*,*cak*),
  isX509Token(*tok*,*cak*,*u*,`"rsasha1"`,*k*),
  getParts(*env*,*parts*),
  mkSignature(*sig*,`"rsasha1"`,*sk*,*parts*),
  replaceSecurityHeader(*env*,[*tok sig*],*env′*).

predicate hasSendPolicyA(*env*:item,*sids*,*rids*,*fresh*:items,*env′*:item) :−
  hasBody(*env*,*b*,*bVal*),
  rid **in** *rids*,
  isX509CertSecret(*rid*,*u*,*tok*,_,*cak*),
  isX509Token(*tok*,*cak*,*u*,`"rsasha1"`,*k*),
  mkCipherValue(*bEnc*,`"rsa"`,*k*,*b*),
  replaceBody(*env*,*bEnc*,*env′*).

predicate hasSendPolicyA(*env*:item,*sids*,*rids*,*fresh*:items,*env'*:item) :−
env' = *env*.

This predicate consists of four clauses mirroring the structure of clauses generated by the translation of send policies ($\mathscr{F}_S$ in Section 5.4). The first clause logically abstracts the clauses generated by $\mathscr{F}_S$(Integrity(Username(*U*),*Pts*),*X*) for any principal names *U*, message parts *Pts*, and policy name *X*. It ignores the names and uses an auxiliary predicate, getParts, that iterates through the possible headers of the envelope and extracts an arbitrary subset *parts* of these headers plus the message body. (The full definition of getParts appears in `partialsend.tf` [9].) The next two clauses similarly abstract the clauses generated from $\mathscr{F}_S$(Integrity(X509(*S*),*Pts*),*X*) and $\mathscr{F}_S$ (Confidentiality(X509(*S*),[Body],*X*) for any *S*, *Pts*, and *X*. The last clause represents the case of an empty assertion.

We also let the predicate hasSendPolicyA^n perform a sequence of *n* invocations of predicate hasSendPolicyA:

predicate hasSendPolicyA^n(*env*:item,*sids*,*rids*,*fresh*:items,*env'*:item) :−
fresh = [n_1 t_1 n_2 t_2 ... n_n t_n @ _],
hasSendPolicyA(*env*,*sids*,*rids*,[n_1 t_1],env^1),
hasSendPolicyA(env^1,*sids*,*rids*,[n_2 t_2],env^2), ...
hasSendPolicyA(env^{n-1},*sids*,*rids*,[n_n t_n],*env'*).

We have a logical refinement between hasSendPolicyA^l and hasSendPolicyA^m:

Lemma 7 *For all $l \leq m$ and all lists fresh with at least $2m$ elements,* hasSendPolicyA^l (*env*,*sids*,*rids*,*fresh*,*env'*) *logically refines* hasSendPolicyA^m(*env*,*sids*,*rids*,*fresh*,*env'*).

Proof: Predicate hasSendPolicyA^l implements a special instance of hasSendPolicyA^m where only the first $2l$ elements of *fresh* are used, and the last $m-l$ invocations of hasSendPolicyA are performed using the fourth clause (for an empty assertion). □

Next, we show that hasSendPolicyA^n logically abstracts the send policy enforcement predicates generated for any policy of depth *n*:

Lemma 8 *Let P be a policy of depth $n = \mathsf{Depth}(P)$ and* hasSendPolicyX *be the predicate generated by $\mathscr{F}_S(P,X)$. Let fresh be a list of at least $2n$ elements.*

For all env, sids, rids, env', hasSendPolicyX(*env*, *sids*,*rids*,*fresh*,*env'*) *logically refines* hasSendPolicyA^n($env, sids, rids, fresh, env$').

Proof: By induction on the structure of *P*. For brevity, we omit predicate arguments, unless we need to track changes in their variables, writing for instance hasSendPolicyX instead of hasSendPolicyX(*env*, *sids*, *rids*, *fresh*, *env'*).

- (Base Case) $P = \text{Integrity}(t,p)$ or $\text{Confidentiality}(t,p)$, with $n = 1$. By definition of $\mathscr{F}_S(P,X)$, hasSendPolicyX refines one clause of hasSendPolicyA: it has additional checks on principal names, and it uses *PartFormula*(*Pts*) and *PartItems*(*Pts*) in place of getParts(*env*, *parts*) and *parts*. We verify that *PartFormula*(*Pts*), *parts* = *PartItems*(*Pts*) refines getParts(*env*, *parts*). Hence, hasSendPolicyX refines hasSendPolicyA^1.

- (Inductive Step) $P = \text{OneOrMore}[P_1, \ldots, P_k]$, with $n = \max\{\mathit{Depth}(P_i) | i = 1..n\}$. By induction hypothesis for each P_i with $n_i = \mathit{Depth}(P_i)$, hasSendPolicyX(*env*, *sids*,*rids*,*fresh*,*env′*) in $\mathscr{F}_S(P_i, X)$ refines hasSendPolicyA^{n_i}(*env*,*sids*,*rids*,*fresh*,*env′*) for *fresh* with at least $2n_i$ elements.

  By Lemma 7, each hasSendPolicyA^{n_i} refines hasSendPolicyA^n for all *fresh* with at least $2n$ elements. By transitivity of refinement, each hasSendPolicyX$_i$ in $\mathscr{F}_S(P_i,X_i)$ refines hasSendPolicyA^n for all *fresh* with at least $2n$ elements. Finally, hasSendPolicyX in $\mathscr{F}_S(\text{OneOrMore}[P_1, \ldots, P_k], X)$ is a disjunction of the predicates hasSendPolicyX in $\mathscr{F}_S(P_i, X)$ for each i. Since each disjunct refines hasSendPolicyA^n, so does the full predicate.

- (Inductive Step) $P = \text{All}[P_1, \ldots, P_k]$. By induction hypothesis, for each P_i with $n_i = \mathit{Depth}(P_i)$, predicate hasSendPolicyX$_i$ in $\mathscr{F}_S(P_i,X_i)$ refines hasSendPolicyA^{n_i} for all *fresh* with at least $2n_i$ elements, and $n = \mathit{Depth}(P) = n_1 + \cdots + n_k$.

  The predicate hasSendPolicyX in $\mathscr{F}_S(\text{All}[P_1, \ldots, P_k], X)$ first extracts $2n$ (that is, $\mathit{FreshVars}(P)$) variables from *fresh* and then computes the conjunction of $\mathscr{F}_S(P_i,X_i)$ for $i = 1 \ldots k$, giving each conjunct $\mathit{FreshVars}(P_i) = 2n_i$ variables. By inlining each conjunct, we obtain hasSendPolicyA^{n_i}, and thus hasSendPolicyX refines the predicate hasSendPolicyA^n for all *fresh* with at least $2n$ elements. □

As a corollary, we obtain a refinement between $\mathscr{S}(C,L)$ and S^n:

Lemma 9 *Let S^n be the script obtained from S by replacing the call to* mkConformant *within GenericSender by a call to* hasSendPolicyA^n*, and by inserting the definitions of* hasSendPolicyA^n *and* getParts*.*

For every link specification L and configuration $\mathscr{C}_R(L)$, there exists $n \geq 0$ such that $\mathscr{S}(\mathscr{C}_R(L),L)$ logically refines S^n.

Proof: S^n is obtained from $\mathscr{S}(C,L)$ by replacing the isConformant predicate by the one in S, and by replacing the mkConformant predicate by hasSendPolicyA^n, hence erasing the checks on *to* and *action*. In the proof of Lemma 4, we established that the isConformant predicate in S abstracts the one in $\mathscr{S}(C,L)$, sowe only need to show that the mkConformant predicate in $\mathscr{S}(C,L)$ refines hasSendPolicyA^n for some n. Let $\{\text{Send}(A_1,P_1), \ldots, \text{Send}(A_k,P_k)\}$ be the send policy maps of C, and $n = \max\{\mathit{Depth}(P_i) \, | i = 1..k\}$. The generic sender process in $\mathscr{S}(C,L)$ creates a *fresh* list with $2n$ elements and invokes mkConformant(*env*,*sids*,*rids*,*fresh*,*env′*). Erasing the checks on *to* and *action* in mkConformant yields a more abstract predicate that implements the disjunction of predicates hasSendPolicyX$_i$ generated by $\mathscr{F}_S(P_i,X_i)$. By Lemma 8, each hasSendPolicyX$_i$ refines hasSendPolicyA$_i^n$ for *fresh* with at least $2n_i$ elements, where $n_i = \mathit{Depth}(P_i)$. By Lemma 7 and transitivity of refinement, each hasSendPolicyX$_i$ also refines hasSendPolicyA^n for *fresh* with at least $2n$ elements. Since each disjunct refines hasSendPolicyA^n, so does the full predicate mkConformant in $\mathscr{S}(C,L)$. Hence, $\mathscr{S}(C,L)$ refines S^n. □

Looping from n to ω To prove that $\mathscr{S}(\mathscr{C}_R(L),L)$ is robustly link-safe, we must now prove that S^n is robustly link-safe for all $n \geq 0$. The only part of S^n that depends on n is the generic sender process:

```
process GenericSender() =
 !in initChan(env);
  in dbChan(sid); in dbChan(rid);
  new freshid:string;
  filter freshId (env,freshid,envId) → envId;
  filter secretEnvelope (envId,B,envB) → envB;
  filter linkAssert(sid,rid,envB,a) → a;
  begin Log(a);
  new n1:string; new t1:string;
  ...
  new nn:string; new tn:string;
  filter hasSendPolicyA^n(envB,[sid],[rid],[n1 t1 ... nn tn],env1) → env1;
  out httpChan(env')
```

We set up auxiliary notations, then give a lemma that relates S^n to S^ω. The lemma relies on observational equivalence, $\approx$, an equivalence relation on processes that is closed by application of evaluation contexts and that preserves all observations of events [4].

$$
\begin{array}{rcl}
A[-] & = & \textbf{!in}\ \mathit{initChan}(\mathit{env}); A'[-] \\
A'[-] & = & \textbf{new}\ \mathit{freshid}\text{:string}; \\
 & & \textbf{filter}\ \text{freshId}\ (\mathit{env},\mathit{freshid},\mathit{envId}) \rightarrow \mathit{envId}; \\
 & & \textbf{filter}\ \text{secretEnvelope}\ (\mathit{envId},B,\mathit{envB}) \rightarrow \mathit{envB}; \\
 & & \textbf{in}\ \mathit{dbChan}(\mathit{sid});\ \textbf{in}\ \mathit{dbChan}(\mathit{rid}); \\
 & & \textbf{filter}\ \text{linkAssert}(\mathit{sid},\mathit{rid},\mathit{envB},a) \rightarrow a; \\
 & & \textbf{begin}\ \mathit{Log}(a);[-] \\
L[-] & = & \textbf{new}\ n\text{:string};\ \textbf{new}\ t\text{:string}; \\
 & & \textbf{filter}\ \text{hasSendPolicyA}(\mathit{envB},[\mathit{sid}],[\mathit{rid}],[n\ t],\mathit{env}') \rightarrow \mathit{env}'; \\
 & & \textbf{let}\ \mathit{envB} = \mathit{env}';[-] \\
M & = & \textbf{out}\ \mathit{httpChan}(\mathit{envB}) \\
N & = & \textbf{out}\ \mathit{loopChan}(\mathit{sid},\mathit{rid},\mathit{envB}) \\
I[-] & = & \textbf{in}\ \mathit{loopChan}(\mathit{sid},\mathit{rid},\mathit{envB});[-] \\
G^n & = & \textbf{!}A[L^n[M]] \\
G^\omega & = & \textbf{!}A[N] \mid \textbf{!}I[L[N]] \mid \textbf{!}I[M]
\end{array}
$$

Lemma 10 *For some evaluation context C_S, we have $S^n \approx C_S[G^n]$ and $S^\omega = C_S[G^\omega]$.*

Proof: The context C_S represents the rest of the script, which is shared between S^n and S^ω. The proof uses standard pi-calculus observational equivalences to (i) inline the definition of hasSendPolicyA^n; (ii) replace the resulting filter with n filters, each calling hasSendPolicyA; and (iii) reorder the series of restrictions and filters in *GenericSender*. □

Lemma 11 *For any process O, if $S^n|O \rightarrow^* T$, then $S^\omega|O \rightarrow^* T'$ with the same events in T and T'.*

Proof: Let $C_O[\text{-}] = C_S[\text{-}]|O$. By Lemma 10, we have $S^n|O \approx C_O[G^n]$, so $S^n|O \rightarrow^* T$ implies $C_O[G^n] \rightarrow^* T^n$ with the same events in T and T^n.

We let Φ range over multisets of triples (G,i,σ) where G is a linear context that does not bind *loopChan*, $i \in 0 \ldots n-1$, σ is a substitution with domain $\{envId, envB, sid, rid, env', a\}$, and define

$$\begin{array}{rcl} G^n_\Phi & = & \prod_{(G,i,\sigma)\in\Phi} G[L^{n-i}[M]\sigma] \\ G^\omega_\Phi & = & \prod_{(G,i,\sigma)\in\Phi} G[N\sigma] \end{array}$$

We show that, if $C_O[G^n] \to^* T^n$, then $T^n \equiv C\,[G^n|G^n_\Phi]$ and $C_O[G^\omega] \to^* T^\omega \equiv C\,[G^\omega|G^\omega_\Phi]$ for some multiset Φ and evaluation context C. The proof is by induction on the number of reduction steps in $C_O[G^n] \to^* T^n$. In the base case, we let $C = C_O$ and $\Phi = \varnothing$.

For the inductive step, assume that $C_O[G^n] \to^* T^n \to T'$. By induction hypothesis, we have $T^n \equiv C\,[G^n|G^n_\Phi]$ and $C_O[G^\omega] \to^* T^\omega \equiv C\,[G^\omega|G^\omega_\Phi]$. Using the explicit representation T^n, we perform a case analysis on the reduction step $T^n \to T'$, with the following cases:

(1) The step involves G^n, hence it is an input on *initChan*. Assume that this input consumes a message e sent on *initChan* in the context C. The resulting process is of the form

$$T' \equiv C'\left[G^n \mid A'[L^n[M]\sigma] \mid G^n_\Phi\right]$$

where C' is C after the output, and $\sigma = \{env := e\}$ Using the same input with G^ω, we obtain

$$T^\omega \to C'\left[G^\omega \mid A'[N\sigma] \mid G^n_\Phi\right]$$

and the two resulting processes meet the inductive hypothesis for C' and $\Phi \uplus \{(A'[-],0,\sigma)\}$. (An output on *initChan* in some G of Φ is handled similarly, by updating G to G'.)

(2) The step involves one of the linear processes G of Φ, and possibly C. Assume that $\Phi = \Phi' \uplus \{(G,i,\sigma)\}$. We distinguish two cases:

(a) $G[\text{-}\,\sigma]$ yields another (non-empty) guarded context $G'[\text{-}\,\sigma']$. We conclude with C' and $\Phi' \uplus \{(G',i,\sigma')\}$.

(b) $G[\text{-}\,\sigma]$ terminates, yielding a context $H[\text{-}\,\sigma']$ that binds names distinct from *loopChan*, and triggers its guarded process. We have

$$T' \equiv C'\left[G^n \mid H[L^{n-i}[M]\sigma'] \mid G^n_{\Phi'}\right]$$

where C' is C after the output. Using the same step, we have

$$T^\omega \to C'\left[G^\omega \mid H[N\sigma'] \mid G^n_{\Phi'}\right]$$

In case $i < n-1$, we have $T' \equiv C'\left[G^n \mid H[L[L^{n-i-1}[M]]\sigma'] \mid G^n_{\Phi'}\right]$ and, using the first replicated input on *loopChan*,

$$T^\omega \to\to C'\left[G^\omega \mid H[L[N]\sigma'] \mid G^n_{\Phi'}\right]$$

The new substitution after the input on *loopChan* is actually $\sigma'\{envB := \sigma'(envB)\}$ which is the same as σ'. We are left with processes that meet the inductive hypothesis for C' and $\Phi' \uplus \{(H[L[\text{-}]], i+1, \sigma')\}$.

In case $i = n-1$, we have $T' \equiv C'\left[G^n \mid H[M\sigma'] \mid G^n_{\Phi'}\right]$ and, using the second replicated input on *loopChan*,

$$T^\omega \rightarrow\rightarrow C'\left[G^\omega \mid H[M\sigma'] \mid G^n_{\Phi'}\right]$$

We are left with processes that meet the inductive hypothesis for $C''[\text{-}] = C'[\text{-} \mid H[M\sigma']]$ and Φ'.

(3) The step involves only the context C, leading to processes that meet the inductive hypothesis for some C' instead of C.

Finally, events in evaluation contexts in T^n or T^ω may occur only in C, hence these events are the same. □

Lemma 12 *If S^ω is robustly link-safe, so is S^n.*

Proof: Suppose $S^n|O \rightarrow^* T^n \Downarrow$ **end** $L([tag$ `"Clear"`s @ $a])$. By Lemma 11, there is a T^ω such that $S^\omega|O \rightarrow^* T^\omega$ and $T^\omega \Downarrow$ **end** $L([tag$ `"Clear"`s @ $a])$. Since S^ω is robustly link-safe, we have either $T^\omega \Downarrow$ **begin** $L([tag$ `"Clear"`s @ $a])$, or $T^\omega \Downarrow$ **begin** $L([tag$ `"Encrypted"`$s\ r$ @ $a])$, or $T^\omega \Downarrow$ **begin** $Leak(s)$. By Lemma 11 again, either $T^n \Downarrow$ **begin** $L([tag$ `"Clear"`s @ $a])$, or $T^n \Downarrow$ **begin** $L([tag$ `"Encrypted"`$s\ r$ @ $a])$, or $T^n \Downarrow$ **begin** $Leak(s)$.

Similarly, if $S^n|O \rightarrow^* T^n \Downarrow$ **end** $L([tag$ `"Encrypted"`$s\ r$ @ $a])$, then either $T^n \Downarrow$ **begin** $L([tag$ `"Encrypted"`$s\ r$ @ $a])$, or $T^n \Downarrow$ **begin** $L([tag$ `"Clear"`s @ $a])$, or $T^n \Downarrow$ **begin** $Leak(s)$. Hence, S^n is robustly safe. □

Proof of Theorem 4: By Lemma 6, Lemma 12, Lemma 1 applied to Lemma 9, and Lemma 5, $\mathscr{S}(\mathscr{C}_R(L), L)$ is robustly safe. □

6.6 Proof of Theorem 5

The proof of Theorem 5 has the same structure as the proof of Theorem 3, but is technically simpler. Let $\mathscr{C}_S(L)$ range over configurations with the same Send policy maps as in L. We exhibit a single script S_s that is logically refined by $\mathscr{S}(\mathscr{C}_S(L), L)$, for any L and $\mathscr{C}_S(L)$. The complete script S_s appears in `partialrecv.tf` [9]. It models a configuration where senders implement link-generated policies, much as in the script S used in Theorem 3, and receivers implement an arbitrary policy. Hence, S_s differs from S mainly in the definition of the process *GenericReceiver*:

process *GenericReceiver*() =
 !in *httpChan*(*env*); **in** *dbChan*(*sid*); **in** *dbChan*(*rid*).

This generic receiver process, called R_s in the proofs, inputs messages then does nothing. (In contrast, the original receiver process calls isConformant and linkAssert then emits an **end** *Log* event, which is irrelevant for secrecy.)

First, we verify that S_s meets the security goal of Theorem 5:

Lemma 13 *S_s preserves the secrecy of B.*

Proof: By verification of script `partialrecv.tf`. □

Then we show that for a given attacker process, every run of $\mathscr{S}(C,L)$ contains the same secrecy-related events as some run of S_s:

Lemma 14 *For any context C, if $\mathscr{S}(\mathscr{C}_S(L),L)|O \rightarrow^* T$ then $S_s|O \rightarrow^* T_s$ with the same messages and* **begin** *events in T and T_s.*

Proof: From the proof of Lemma 4, every sequence of steps in $\mathscr{S}(\mathscr{C}_S(L),L)|O$ that does not involve the generic receiver can be performed by $S|O$ and hence by $S_s|O$ (since the corresponding parts of the script are unchanged.) If a step involves *GenericReceiver*, it must be a communication on *httpChan*, or an input on *dbChan*, or a predicate invocation on isConformant or linkAssert, or an **end** *Log* event. The first two can be performed by the corresponding process in $S_s|O$ and the last three do not generate any messages or **begin** events. □

To complete the proof, we carry over our notion of secrecy from S_s to $\mathscr{S}(\mathscr{C}_S(L),L)$:

Proof of Theorem 5: By Lemma 14, if we have $\mathscr{S}(\mathscr{C}_S(L),L)|O \rightarrow^* T \Downarrow$ **out** $c(B)$ for some attacker process O and public channel c, then we also have $S_s|O \rightarrow^* T_s \Downarrow$ **out** $c(B)$.

By Lemma 13, either $T_s \Downarrow$ **begin** *Log*([*tag* `"Clear"`*sender receiver* @ *a*]), or $T_s \Downarrow$ **begin** *Log*([*tag* `"Encrypted"`*sender receiver* @ *a*]) and $T_s \Downarrow$ **begin** *Leak*(*receiver*).

By Lemma 14 again, either $T \Downarrow$ **begin** *Log*([*tag* `"Clear"`*sender receiver* @ *a*]), or $T \Downarrow$ **begin** *Log*([*tag* `"Encrypted"`*sender receiver* @ *a*]) and $T \Downarrow$ **begin** *Leak*(*receiver*). Hence $\mathscr{S}(\mathscr{C}_S(L),L)$ preserves the secrecy of B. □

7 Correlation

In this section, we extend and adapt our model, developed in Sections 4, 5, and 6, in order to verify more general authentication properties relating sequences of messages. We refer to those properties as correlation properties.

We focus on a simple correlation property between requests and responses. Whenever a client accepts a response message from the web service, we want to guarantee that the server generated this message in response to a particular earlier request from this client. (In some related work, we further develop TulaFale models that can deal with secure sessions of arbitrary length [3], enabled by the specifications WS-Trust and WS-SecureConversation, but these models are not policy-based.)

We adapt our TulaFale scripts to formalize this property. Instead of the generic senders and receivers of $\mathscr{S}(C,L)$, the adapted script $\mathscr{T}(C,L)$ models generic clients and services. The generic client repeatedly sends a request and then waits for a response that matches the request. Symmetrically, the generic server repeatedly receives requests and sends back responses. We present the generic client for $\mathscr{T}(C_0,L_0)$:

process *GenericClient*() =
 !in *initChan* (*req*);
  in *dbChan* (*sid*); **in** *dbChan* (*rid*);
  new *freshid*:string; **new** *n*:string; **new** *t*:string;
  filter freshId (*req*,*freshid*,*envId*) $\rightarrow$ *envId*;
  filter secretEnvelope (*envId*,*B*,*envB*) $\rightarrow$ *envB*;

```
filter linkAssert(sid,rid,envB,aReq) → aReq in
begin Log(aReq);
filter mkConformant(env,[sid],[rid],[n t],env′) → env′ in
out httpChan(env′);
in httpChan(envResp);
filter isConformant(envResp,[rid],[sid],resp) → resp in
filter linkAssert(rid,sid,resp,aResp) → aResp in
filter hasCorrelator(resp,env′,sid) → in
end Log(aResp);
end LogCorr(aReq,aResp)
```

To a first approximation, *GenericClient* is the sequential composition of one copy of *GenericSender* followed by one copy of *GenericReceiver*. As in *GenericSender*, the client accepts an envelope from the environment and enforces the send policy on it by invoking predicate mkConformant. It also issues **begin** *Log*(*aReq*), where *aReq* summarizes the request message, before sending the request message out on *httpChan*.

Then, as in *GenericReceiver*, the client waits for a response, checks that the received envelope conforms to the receive policy, then issues **end** *Log*(*aResp*) where *aResp* summarizes the response message. In addition, it checks that the response is correlated to the request, using predicate hasCorrelator, then issues another, more comprehensive event **end** *LogCorr*(*aReq*,*aResp*) that expresses its intent to process the received message as the genuine response to its request.

Symmetrically, $\mathscr{T}(C_0, L_0)$ defines a process *GenericService* that receives a request, and issues an additional event **begin** *LogCorr*(*aReq*,*aResp*) before sending a response.

In practice, correlation between SOAP requests and responses is often implemented by including in responses a (signed) *RelatesTo* header that echoes the (signed) *MessageId* of requests. However, this mechanism does not suffice to prevent some insider correlation attacks, as for instance nothing prevents a compromised client from issuing a different request with the same *MessageId* as in an intercepted message. Since our model enables these attacks, we extend the *RelatesTo* header so that it also includes in every response both the *MessageId* of the request and the name of the principal that signed that request. Our extended script defines hasCorrelator as follows:

```
predicate hasCorrelator(resp,req,sid:item) :−
  hasHeaderMessageId(req,_,id),
  hasHeaderRelatesTo(resp,relatesTo,id),
  hasName(sid,su),
  relatesTo = <RelatesTo client=su>_</>.
```

(This extension is technically convenient, as the extended *RelatesTo* header is signed as before, but other approaches may be preferred in order to comply with WS-Addressing.)

We now formally state our intended correlation property:

Correlation:

A script *S correlates responses to requests* when, for any run with any attacker process, if **end** *LogCorr*(*a1*,*a2*) occurs, then either **begin** *LogCorr*(*a1*,*a2*) or **begin** *Leak*(*sender*) previously occurred, where *a1* = ["Request"*secr sender* @ *contents*] and *a2* = ["Response"*secr sender* @ *contents*].

We easily extend our verification results to the adapted scripts, first for our example links L_0 and L_1 and configurations C_0 and C_1 (used in Theorems 1 and 2).

Theorem 6 *Let $T_i = \mathscr{T}(C_i, L_i)$ for $i = 0, 1$. We have (1) T_i is robustly safe; (2) T_i preserves secrecy; (3) T_i is adequate for values of the forms $aReq_i$ and $aResp_i$; (4) T_i correlates responses to requests.*

Proof: By verification of scripts `s0-corr.tf` and `s1-corr.tf` [9]. □

On the other hand, property (4) fails to verify if we omit to correlate the usernames as well as the message identifier in hasCorrelator, as discussed above.

As a more complex example, we consider a variant of the configuration C_2. Instead of two services, C_2' has one service with two actions with different policies:

Target′ = ToAction(`"http://BobsPetShop.com/service.asmx"`,
`"http://petshop/regular"`)
C_2' = C_0 @ [Receive(*Target*′, Integrity(Username, [Body]))]

The configuration C_2' has a receive policy but no send policy for *Target*′. The reason for this is that, since the policy for this new action is clearly too weak, we choose not to consider the security of clients sending messages to it. Still, we would expect the correlation property to hold for clients talking to the premium *Target* (in L_0 and C_0). As we try to automatically verify $\mathscr{T}(C_2, L_0)$, however, the proof fails and ProVerif generates a counter-example: even though no client tries to send messages to *Target*', an attacker can take a message sent to *Target*, modify its headers, and send it to *Target*', then take the response and forward it as a response from *Target*. Since both actions use the same server principal identity, the two kinds of response messages are indistinguishable, so the attack succeeds. This correlation attack illustrates the risk of running logically separate services on behalf of the same security principal.

Finally, we can verify that our correlation property holds for all link-generated configurations. We extend Theorem 3 accordingly:

Theorem 7 *For any link specification L, let $T = \mathscr{T}(\mathscr{C}(L), L)$. We have (1) T is robustly safe; (2) T preserves secrecy; (3) T correlates responses to requests.*

Proof: We reuse the structure of the proof of Theorem 3—the manual part of the proof is almost unchanged.

We first define a variant $\mathscr{T}'(\mathscr{C}(L), L)$ of the script $\mathscr{T}(\mathscr{C}(L)), L)$ by rearranging filters in *GenericClient* and *GenericServer*. We adapt Lemma 2 and similarly show, if $\mathscr{T}'(\mathscr{C}(L), L)$ is robustly safe, preserves secrecy, and correlates responses to requests, so does $\mathscr{T}(\mathscr{C}(L)), L)$. We automatically verify a variant of Lemma 3 using script `linkgen-corr.tf` [9]. We establish a variant of Lemma 4 with $\mathscr{T}$ instead of $\mathscr{S}$, with the same proof as for Lemma 4 (since $\mathscr{T}$ and $\mathscr{S}$ have the same definitions for the predicates refined in the lemma). We conclude by applying Lemma 1. □

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 that they correctly implement the intended security properties.

Our link language is more abstract than WS-SecurityPolicy, in that it describes links between endpoints, rather than how a single endpoint sends or receives messages. Although, for the sake of exposition the link language of this paper is rather simple, it conveniently expresses strong authentication, secrecy, and correlation properties that are not so easy to achieve at the message level using WS-Security and WS-SecurityPolicy.

Our tools automatically build a formal model of a given SOAP configuration, with an unbounded number of senders and receivers, then verify that there are no vulnerabilities to XML rewriting attacks by an unbounded opponent. An improvement in the threat model compared to our previous work on TulaFale [4, 9] is that the opponent can learn the credentials of some users, and hence can attempt insider attacks. Inasmuch as WS-SecurityPolicy files entirely control security processing for SOAP messaging (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.

Formal security guarantees apply to our model, rather than actual WSE systems, so it is important to understand their relationship. Our model, seen as a “reference implementation” (on the right in Figure 1), carefully follows the modular structure of our target system (on the left in Figure 1). Our modelling of WS-Security, for example, was developed and thoroughly tested independently of higher-level specifications, by checking that TulaFale envelopes correspond to those experimentally observed with WSE for a series of examples, and by comparing the dynamic checks on security tokens performed in TulaFale and in WSE. We experimentally validated our compilation of policies, and checked that our generated policies are accepted by WSE and yield the expected SOAP envelopes. Moreover, our tools check (rather than assume) the correctness of policy generation. Some of our theorems are specific results concerning given configurations, while others are general results concerning all configurations either wholly or partially conforming to link specifications.

Policies capture security aspects of web services, but of course the security of the system also depends on the application. For instance, an application, or upper layers in the protocol, may still rely on additional inputs extracted from custom headers not within the scope of SOAP policies, at its own risk.

Although there are several tools to check security policies at lower layers, such as IPsec [24], 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 [28] itself includes a security settings wizard that interactively gathers data analogous to our link specifications 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 [34] 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 [32, 29, 27, 33] generate implementation code from abstract protocol descriptions in Dolev-Yao [19] formalisms such as strand spaces, CAPSL, and the spi-calculus. The AGVI project [32] using the Athena protocol verifier, itself based on strand spaces, appears to be the first attempt to generate implementation code from protocol description; AGVI generates Java code that produces and consumes byte arrays, but it was not connected to actual networking code. Millen and Muller [29] generate Java implementations from CAPSL descriptions; clients and servers communicate over sockets. Lukell, Veldman, and Hutchison [27] generate Java code that supports ASN.1 message formats. Spi2Java [33] generates Java implementations of protocols specified in 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 users get the benefits of formal verification without needing to rewrite their protocols in a new language. (With our tools, users only need to express security goals, either directly in TulaFale, or via our link language.)

Zheng, Chong, Myers, and Zdancewic [38] 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.

The tools of this paper catch the sort of errors we found by careful manual inspection of actual policies written for use with WSE. Still, in the case that intended security guarantees cannot be proved, the error messages are at a low-level (TulaFale and ProVerif) and are likely to be cryptic to many users of WSE, if not to experts in security protocols. In an effort to make some of the benefits of this research more widely available, we subsequently designed a simple query-based analyzer that checks for syntactic error patterns in WSE configuration and policy files. This tool, the WSE Policy Advisor [8], has queries to detect specific vulnerabilities to impersonation, replay, redirection, and other attacks. The documentation lists three dozen different conditions, including some conditions not detected by TulaFale, such as vulnerabilities to dictionary attacks on weak secrets. Since each query detects a particular syntactic condition, the tool can generate fairly high-level advice concerning the risk, and suggest specific remedial action. Overall, although the advisor has many limitations—it offers no formal guarantees and generates false alarms—it does detect common failure modes that typical users would otherwise not notice. A productized version of the advisor [6] ships in version 3.0 of WSE.

Since the research described here was completed, we have developed tools for verifying implementation code of cryptographic protocols [11], with the aim of verifying reference implementations of web services security protocols [7]. A forthcoming pa-

per [10] describes a tool for generating and verifying reference implementations of security policies expressed in a recent version of WS-SecurityPolicy [25].

Web services security will certainly evolve; the specifications we have discussed 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 to enable this work. Riccardo Pucella prototyped a first compiler from policies to TulaFale during an internship. Tuomas Aura, Daniel Stieger, and anonymous reviewers made useful comments on earlier versions of the paper.

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] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005.

[3] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. *ACM Transactions on Information and System Security (TISSEC)*, 10(8), May 2006.

[4] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Comput. Sci.*, 340(1):102–153, June 2005. See also Microsoft Research Technical Report MSR-TR-2003-83.

[5] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. Technical Report MSR–TR–2004–84, Microsoft Research, Nov. 2005.

[6] K. Bhargavan, C. Fournet, and A. D. Gordon. Policy advisor for WSE 3.0. In *Web Service Security: Scenarios, patterns, and implementation guidance for Web Services Enhancements (WSE) 3.0*, pages 324–330. Microsoft Press, 2006.

[7] K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-Security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *LNCS*, pages 88–106. Springer, 2006.

[8] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O'Shea. An advisor for web services security policies. In *2005 ACM Workshop on Secure Web Services (SWS)*, pages 1–9, 2005. Tool download available from `http://Securing.WS`.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004. Tool download available from http://Securing.WS.

[10] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *ASIACCS'08*, 2008. To appear.

[11] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

[12] 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.

[13] 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.

[14] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, USA, June 2005. IEEE Computer Society.

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

[16] 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.

[17] 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.

[18] 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. Version 1.0.

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

[20] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon. *XML Encryption Syntax and Processing*, 2002. W3C Recommendation, at http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

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

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

[23] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. *Formal Aspects of Computing*, 17:277–318, 2005.

[24] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. *International Journal of Information Security*, 4(1–2):29–48, 2005.

[25] C. Kaler, A. Nadalin, et al. Web services security policy language (WS-SecurityPolicy), version 1.1, July 2005.

[26] G. Lowe. Analyzing protocols subject to guessing attacks. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.

[27] 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.

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

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

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

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

[38] 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.