Validating a Web Service Security Abstraction by Typing

Andrew D. Gordon
Microsoft Research

Riccardo Pucella
Cornell University

ABSTRACT

An XML web service is, to a first approximation, an RPC service in which requests and responses are encoded in XML as SOAP envelopes, and transported over HTTP. We consider the problem of authenticating requests and responses at the SOAP-level, rather than relying on transport-level security. We propose a security abstraction, inspired by earlier work on secure RPC, in which the methods exported by a web service are annotated with one of three security levels: none, authenticated, or both authenticated and encrypted. We model our abstraction as an object calculus with primitives for defining and calling web services. We describe the semantics of our object calculus by translating to a lower-level language with primitives for message passing and cryptography. To validate our semantics, we embed correspondence assertions that specify the correct authentication of requests and responses. By appeal to the type theory for cryptographic protocols of Gordon and Jeffrey's Cryptyc, we verify the correspondence assertions simply by typing. Finally, we describe an implementation of our semantics via custom SOAP headers.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.6 [**Operating Systems**]: Security and Protection; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Languages, security, theory, verification

Keywords

Web services, remote procedure call, authentication, type systems

1. INTRODUCTION

It is common to provide application-level developers with security abstractions that hide detailed implementations at lower levels of a protocol stack. For example, the identity of the sender of a message may be exposed directly at the application-level, but computed via a hidden, lower level cryptographic protocol. The purpose of this paper is to explore how to build formal models of such security abstractions, and how to validate their correct implementation in terms of cryptographic primitives. Our setting is an experimental implementation of SOAP security headers for XML web services.

1.1 Motivation: Web Services and SOAP

A crisp definition, due to the builders of the TerraService.NET service, is that "a web service is a web site intended for use by computer programs instead of human beings" [8]. Each request to or response from a web service is encoded in XML as a SOAP *envelope* [11]. An envelope consists of a *header*, containing perhaps routing or security information, and a *body*, containing the actual data of the request or response. A promising application for web services is to support direct retrieval of XML documents from remote databases, without resorting to unreliable "screen scraping" of data from HTML pages. Google already offers programmatic access to its database via a web service [20]. Another major application is to support systems interoperability within an enterprise's intranet.

The interface exported by a web service can be captured as an XML-encoded service description, in WSDL format [13], that describes the methods—and the types of their arguments and results—that make up the service. Tools exist for application-level developers to generate a WSDL description from the code of a service, and then to generate proxy code for convenient client access to the web service. Like tools for previous RPC mechanisms, these tools abstract from the details of the underlying messaging infrastructure. They allow us to regard calling a web service, for many if not all purposes, as if it were invoking a method on a local object. Our goal is to augment this abstraction with security guarantees.

There are many signs of fervour over web services: there is widespread tool support from both open source and commercial software suppliers, and frequent news of progress of web service standards at bodies such as OASIS and the W3C. Many previous systems support RPC, but one can argue that what's new about web services is their combination of vendor-neutral interoperability, internet-scale, and toolsets for "mere mortals" [8]. Still, there are some reasons for caution. The XML format was not originally designed for messaging; it allows for interoperability but is inefficient compared to binary encodings. Moreover, it would be useful to use web services for inter-organisational communication, for example, for e-commerce, but there is as yet little experience or agreement on SOAP-level security mechanisms.

In fact, there is already wide support for security at the transport-level, that is, for building secure web services using HTTPS and SSL. Still, SSL encrypts all traffic between the client and the web

ACM Workshop on XML Security 2002 Washington D.C. USA

server, so that it is opaque to intermediaries. Hence, messages cannot be monitored by firewalls and cannot be forwarded by intermediate untrusted SOAP-level routers. There are proposals to avoid some of these difficulties by placing security at the SOAP-level, that is, by partially encrypting SOAP bodies and by including authenticators, such as signatures, in SOAP headers. For example, the WS-Security [6] specification describes an XML syntax for including such information in SOAP envelopes.

Hence, the immediate practical goal of this work is to build and evaluate an exploratory system for SOAP-level security.

1.2 Background: Correspondences and Spi

Cryptographic protocols, for example, protocols for authenticating SOAP messages, are hard to get right. Even if we assume perfect cryptography, exposure to various replay and impersonation attacks may arise because of flaws in message formats. A common and prudent procedure is to invite expert analysis of any protocol, rather than relying on security through obscurity. Moreover, it is a useful discipline to specify and verify protocol goals using formal notations. Here, we specify authenticity goals of our protocol using Woo and Lam's correspondence assertions [37], and verify them, assuming perfect cryptography in the sense of Dolev and Yao [16], using type theories developed as part of the Cryptyc project [21, 22, 23].

Woo and Lam's correspondence assertions [37] are a simple and precise method for specifying authenticity properties. The idea is to specify labelled events that mark progress through the protocol. There are two kinds: begin-events and end-events. The assertion is that every end-event should correspond to a distinct, preceding begin-event with the same label. For example, Alice performs a begin-event with label "Alice sending Bob message M" at the start of a session when she intends to send M to Bob. Upon receiving M and once convinced that it actually comes from Alice, Bob performs an end-event with the same label. If the correspondence assertion can be falsified, Bob can be manipulated into thinking a message comes from Alice when in fact it has been altered, or came from someone else, or is a replay. On the other hand, if the correspondence assertion holds, such attacks are ruled out.

There are several techniques for formally specifying and verifying correspondence assertions. Here, we model SOAP messaging within a process calculus, and model correspondence assertions by begin- and end-statements within the calculus. We use a form of the spi-calculus [21], equipped with a type and effect system able to prove by typechecking that correspondence assertions hold in spite of an arbitrary attacker. Spi [5] is a small concurrent language with primitives for message passing and cryptography, derived from the π-calculus [32].

1.3 Contributions of this Paper

Our approach is as follows:

- Section 2 describes our high-level abstraction for secure messaging.
- Section 3 models the abstraction as an object calculus with primitives for creating and calling web services.
- Section 4 defines the semantics of our abstraction by translating to the spi-calculus. Correspondence assertions specify the authenticity guarantees offered to caller and callee, and are verified by typechecking.
- Section 5 describes a SOAP-based implementation using Visual Studio .NET.

Our main innovation is the idea of formalizing the authentication guarantees offered by a security abstraction by embedding correspondence assertions in its semantics. On the other hand, our high-level abstraction is fairly standard, and is directly inspired by work on secure network objects [35]. Although the rather detailed description of our model and its semantics may seem complex, the actual cryptographic protocol is actually quite simple. Still, we believe our framework and its implementation are a solid foundation for developing more sophisticated protocols and their abstractions.

Some formal details are relegated to the appendices. Appendix A gives sample messages exchanged during web service methods calls using our abstractions, and Appendix B gives a detailed overview of the spi-calculus used in the paper. A technical report [24] includes details omitted from this conference paper, such as the formal details of the translation of our object calculus into the spi-calculus, as well as some extensions, such as a translation relying on certificates, and all proofs.

2. A SECURITY ABSTRACTION

We introduce a security abstraction for web services, where the methods exported by a web service are annotated by one of three security levels:

`None`	unauthenticated call
`Auth`	authenticated call
`AuthEnc`	authenticated and encrypted call

A *call* from a client to a web service is made up of two messages, the *request* from the client to the web service, and the *response* from the web service to the client. The inspiration for the security levels, and the guarantees they provide, comes from SRC Secure Network Objects [35]. An authenticated web method call provides a guarantee of *integrity* (that the request that the service receives is exactly the one sent by the client and that the response that the client receives is exactly the one sent by the service as a response to this request) and *at-most-once semantics* (that the service receives the request most once, and that the client receives the response at most once). An authenticated and encrypted web method call provides all the guarantees of an authenticated call, along with a guarantee of *secrecy* (that an eavesdropper does not obtain any part of the method name, the arguments, or the results of the call).

In C#, where users can specify *attributes* on various entities, our security annotations take the form of an attribute on web methods, that is, the methods exported by a web service. The attribute is written `[SecurityLevel(`*level*`)]`, where *level* is one of `None`, `Auth`, or `AuthEnc`. For example, consider a simple interface to a banking service, where `[WebMethod]` is an attribute used to indicate a method exported by a web service:

```
class BankingServiceClass {

  string callerid;

  [WebMethod] [SecurityLevel(Auth)]
  public int Balance (int account);

  [WebMethod] [SecurityLevel(AuthEnc)]
  public string Statement (int account);

  [WebMethod] [SecurityLevel(Auth)]
  public void Transfer (int source,
                        int dest,
                        int amount);
}
```

The annotations get implemented by code to perform the authentication and encryption, at the level of SOAP envelopes, transparently from the user. The annotations on the web service side will generate a method on the web service that can be used to establish a security context. This method will never be invoked by the user, but automatically by the code implementing the annotations. For the purpose of this paper, we assume a simple setting for authentication and secrecy, namely that the principals involved possess shared keys. Specifically, we assume a distinct key K_{pq} shared between every pair of principals p and q. We use the key K_{pq} when p acts as the client and q as the web service. (Notice that K_{pq} is different from K_{qp}.) It is straightforward to extend our approach to different settings such as public-key infrastructures or certificate-based authentication mechanisms (see our technical report [24]).

An authenticated call by p to a web method ℓ on a web service w owned by q with arguments $u_1, \ldots, u_n$ producing a result r uses the following protocol:

$p \rightarrow q$: request nonce
$q \rightarrow p : n_q$
$p \rightarrow q : p, req(w, \ell(u_1, \ldots, u_n), s, n_q), n_p,$
$\quad Hash(req(w, \ell(u_1, \ldots, u_n), s, n_q), K_{pq})$
$q \rightarrow p : q, res(w, \ell(r), s, n_p), Hash(res(w, \ell(r), s, n_p), K_{pq})$

Here, $Hash$ is a cryptographic hash function (a one-way message digest function such as MD5). We tag the request and the response messages to be able to differentiate them. We also tag the response with the name of the method that was originally called. We include a unique *session tag* s in both the request and response message to allow the caller p to match the response with the actual call that was performed.

An authenticated and encrypted call by p to a web method ℓ on a web service w owned by q with arguments $u_1, \ldots, u_n$ producing a result r uses a similar protocol, with the difference that the third and fourth messages are encrypted using the shared key instead of signed:

$p \rightarrow q$: request nonce
$q \rightarrow p : n_q$
$p \rightarrow q : p, \{req(w, \ell(u_1, \ldots, u_n), s, n_q)\}_{K_{pq}}, n_p$
$q \rightarrow p : q, \{res(w, \ell(r), s, n_p)\}_{K_{pq}}$

To convince ourselves that the above protocols do enforce the guarantees prescribed by the security abstraction, we typically argue as follows. Let's consider the authenticated and encrypted case, the authenticated case being similar. When the web service w run by principal q receives a request $w, \ell(u_1, \ldots, u_n), s, n_q$ encrypted with K_{pq} (q uses the identity p in the request to determine which key to use), it knows that only p could have created the message, assuming that the shared key K_{pq} is kept secret by both p and q. This enforces the integrity of the request. Since the message also contains the nonce n_q that the web service can check has never appeared in a previous message, it knows that the message is not a replayed message, hence enforcing at-most-once semantics. Finally, the secrecy of the shared key K_{pq} implies the secrecy of the request. A similar argument shows that the protocol satisfies integrity, at-most-once-semantics, and secrecy for the response.

What do we have at this point? We have an informal description of a security abstraction, we have an implementation of the abstraction in terms of protocols, and an informal argument that the guarantees prescribed by the abstraction are enforced by the implementation. How do we make our security abstraction precise, and how do we ensure that the protocols do indeed enforce the required guarantees? In the next section, we give a formal model to make the abstraction precise. Then, we formalize the implementation by showing how to translate the abstractions into a lower level calculus that uses the above protocols. We use types to show that guarantees are formally met by the implementation, via correspondence assertions.

3. A FORMAL MODEL

We model the application-level view of authenticated messaging as an object calculus. Object calculi [1, 25, 29] are object-oriented languages in miniature, small enough to make formal proofs feasible, yet large enough to study specific features. As in FJ [29], objects are typed, class-based, immutable, and deterministic. As in some of Abadi and Cardelli's object calculi [1], we omit subtyping and inheritance for the sake of simplicity. In spite of this simplicity, our calculus is Turing complete. We can define classes to implement arithmetics, lists, collections, and so on.

To model web services, we assume there are finite sets $Prin$ and $WebService$ of principal identifiers and web service identifiers, respectively. We think of each $w \in WebService$ as a URL referring to the service; moreover, $class(w)$ is the name of the class that implements the service, and $owner(w) \in Prin$ is the principal running the service.

To illustrate this model, we express the banking service interface introduced in the last section in our calculus. Suppose there are two principals $Alice, Bob \in Prin$, and a web service $w =$ *http://bob.com/BankingService*, where we have $owner(w) = Bob$ and $class(w) = BankingServiceClass$. Suppose we wish to implement the $Balance$ method so that given an account number, it checks that it has been called by the owner of the account, and if so returns the balance. If $Alice$'s account number is 12345, we might achieve this as follows:

$class\ BankingServiceClass$
$\quad Id\ CallerId$
$\quad Num\ Balance(Num\ account)$
$\quad\quad if\ account = 12345\ then$
$\quad\quad\quad if\ this.CallerId = Alice\ then\ 100\ else\ null$
$\quad\quad else\ \ldots$

There are a few points to note about this code. First, as in BIL [25], method bodies conform to a single applicative syntax, rather than there being separate grammars for statements and expressions. Second, while the C# code relies on attributes to specify exported methods and security levels, there are not attributes in our calculus. For simplicity, we assume that all the methods of a class implementing a web service are exported as web methods. Furthermore, we assume that all these exported methods are authenticated and encrypted, as if they had been annotated `AuthEnc`. (It is straightforward to extend our calculus to allow per-method annotations but it complicates the presentation of the translation in the next section.)

Every class implementing a web service has exactly one field, named $CallerId$, which exposes the identity of the caller, and allows application-level authorisation checks.

We write $w{:}Balance(12345)$ for a client-side call to method $Balance$ of the service w. The semantics of such a web service call by $Alice$ to a service owned by Bob is that Bob evaluates the local method call $new\ BankingServiceClass(Alice).Balance(12345)$ as Bob. In other words, Bob creates a new object of the form $new\ BankingServiceClass(Alice)$ (that is, an instance of the class $BankingServiceClass$ with $CallerId$ set to $Alice$) and then calls the $Balance$ method. This would terminate with 100, since the value of $this.CallerId$ is $Alice$. (For simplicity, we assume every class in the object calculus has a single constructor whose argu-

ments are the initial values of the object's fields.) This semantics guarantees to the server Bob that the field $CallerId$ contains the identity of his caller, and guarantees to the client $Alice$ that only the correct owner of the service receives the request and returns the result.

In a typical environment for web services, a client will not invoke web services directly. Rather, a client creates a proxy object corresponding to the web service, which encapsulates the remote invocations. Those proxy objects are generally created automatically by the programming environment. Proxy objects are easily expressible in our calculus, by associating with every web service w a proxy class $proxy(w)$. The class $proxy(w)$ has a method for every method of the web service class, the implementation for which simply calls the corresponding web service method. The proxy class also has a field Id holding the identity of the owner of the web service. Here is the client-side proxy class for our example service:

$$
\begin{array}{l}
class\ BankingServiceProxy \\
\quad Id\ Id() \\
\quad\quad Bob \\
\quad Num\ Balance(Num\ account) \\
\quad\quad w{:}Balance(account)
\end{array}
$$

The remainder of this section details the syntax and informal semantics of our object calculus.

3.1 Syntax

In addition to $Prin$ and $WebService$, we assume finite sets $Class$, $Field$, $Meth$ of class, field, and method names, respectively.

CLASSES, FIELDS, METHODS, PRINCIPALS, WEB SERVICES:

$c \in Class$	class name
$f \in Field$	field name
$\ell \in Meth$	method name
$p \in Prin$	principal name
$w \in WebService$	web service name

There are two kinds of data type: Id is the type of principal identifiers, and $c \in Class$ is the type of instances of class c. A method signature specifies the types of its arguments and result.

TYPES AND METHOD SIGNATURES:

$A, B \in Type ::=$	type
$\quad Id$	principal identifier
$\quad c$	object
$sig \in Sig$	method signature
$\quad B(A_1\ x_1, \ldots, A_n\ x_n)$	(x_i distinct)

An execution environment defines the services and code available in the distributed system. In addition to $owner$ and $class$, described above, the maps $fields$ and $methods$ specify the types of each field and the signature and body of each method, respectively. We write $X \to Y$ and $X \xrightarrow{\text{fin}} Y$ for the sets of total functions and finite maps, respectively, from X to Y.

EXECUTION ENVIRONMENT: $(fields, methods, owner, class)$

$fields \in Class \to (Field \xrightarrow{\text{fin}} Type)$	fields of a class
$methods \in Class \to (Meth \xrightarrow{\text{fin}} Sig \times Body)$	methods of a class
$owner \in WebService \to Prin$	service owner
$class \in WebService \to Class$	service implementation

We complete the syntax by giving the grammars for *method bodies* and for *values*.

VALUES AND METHOD BODIES:

x, y, z	name: variable, argument
$u, v \in Value ::=$	value
$\quad x$	variable
$\quad null$	null
$\quad new\ c(v_1, \ldots, v_n)$	object
$\quad p$	principal identifier
$a, b \in Body ::=$	method body
$\quad v$	value
$\quad let\ x{=}a\ in\ b$	let-expression
$\quad if\ u = v\ then\ a\ else\ b$	conditional
$\quad v.f$	field lookup
$\quad v.\ell(u_1, \ldots, u_n)$	method call
$\quad w{:}\ell(u_1, \ldots, u_n)$	service call

The free variables $fv(a)$ of a method body are defined in the usual way, where the only binder is x being bound in b in the expression $let\ x{=}a\ in\ b$. We write $a\{x{\leftarrow}b\}$ for the outcome of a capture-avoiding substitution of b for each free occurence of the variable x in method body a. We view method bodies as being equal up to renaming of bound variables. Specifically, we take $let\ x{=}a\ in\ b$ to be equal to $let\ x'{=}a\ in\ b\{x{\leftarrow}x'\}$, if $x' \notin fv(b)$.

Our syntax for bodies is in a reduced form that simplifies its semantics; in examples, it is convenient to allow a more liberal syntax. For instance, we define $if\ a_1 = a_2\ then\ b_1\ else\ b_2$ as a shorthand for $let\ x_1{=}a_1\ in\ let\ x_2{=}a_2\ in\ if\ x_1 = x_2\ then\ b_1\ else\ b_2$. We already used this shorthand when writing $if\ this.CallerId = Alice\ then\ 100\ else\ null$ in our example. Similarly, we assume a class Num for numbers, and write integer literals such as 100 as shorthand for objects of that class.

Although objects are values, in this calculus, web services are not. This reflects the fact that current WSDL does not allow for web services to be passed as requests or results.

We assume all method bodies in our execution environment are well-typed. If $methods(c)(\ell) = (sig, b)$ and the signature $sig = B(A_1\ x_1, \ldots, A_n\ x_n)$ we assume that the body b has type B given a typing environment $this{:}c, x_1{:}A_1, \ldots, x_n{:}A_n$. The variable $this$ refers to the object on which the ℓ method was invoked. The typing rules, which are standard, appear in the technical report [24]. We also assume the class $class(w)$ corresponding to each web service w has a single field $callerid$.

3.2 Informal Semantics of our Model

We explain informally the outcome of evaluating a method body b as principal p, that is, on a client or server machine controlled by p. (Only the semantics of web service calls depend on p.)

To evaluate a value v as p, we terminate at once with v itself.

To evaluate a let-expression $let\ x{=}a\ in\ b$ as p, we first evaluate a as p. If a terminates with a value v, we proceed to evaluate $b\{x{\leftarrow}v\}$, that is, b with each occurrence of the variable x replaced with v. The outcome of evaluating $b\{x{\leftarrow}v\}$ as p is the outcome of evaluating the whole expression.

To evaluate a conditional $if\ u = v\ then\ a\ else\ b$ as p, we evaluate a as p if u and v are the same; else we evaluate b as p.

To evaluate a field lookup $v.f$ as p, when v is an object value $new\ c(v_1, \ldots, v_n)$, we check f is the jth field of class c for some $j \in 1..n$ (that is, that $fields(c) = f_i \mapsto A_i\ ^{i \in 1..n}$ and that $f = f_j$), and then return v_j. If v is null or if the check fails, evaluation has gone wrong.

To evaluate a method call $v.\ell(u_1, \ldots, u_n)$ as p, when v is an object $new\ c(v_1, \ldots, v_n)$, we check ℓ is a method of class c (that is, that $methods(c) = \ell_i \mapsto (sig_i, b_i)\ ^{i \in 1..m}$ and that $\ell = \ell_j$ for some $j \in 1..m$) and we check the arity of its signature is n

(that is, that $sig_j = B(A_1\ x_1, \ldots, A_n\ x_n)$) and then we evaluate the method body as p, but with the object v itself in place of the variable $this$, and actual parameters u_1, ..., u_n in place of the formal parameters x_1, ..., x_n (that is, we evaluate the expression $b_i\{this \leftarrow v, x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\}$). If v is null or if either check fails, evaluation has gone wrong.

To evaluate a service call $w{:}\ell(u_1, \ldots, u_n)$ as p, we evaluate the method call $new\ c(p).\ell(u_1, \ldots, u_n)$ as q, where $c = class(w)$ is the class implementing the service, and $q = owner(w)$ is the principal owning the service. (By assumption, c's only field is $CallerId$ of type Id.) This corresponds directly to creating a new object on q's web server to process the incoming request.

4. A SPI-CALCULUS SEMANTICS

We confer a formal semantics on our object calculus by translation to the spi-calculus [5, 21], a lower-level language with primitives for message-passing (to model SOAP requests and responses) and cryptography (to model encryption and decryption of SOAP headers and bodies).

4.1 A Typed Spi-Calculus (Informal Review)

To introduce the spi-calculus, we formalize the situation where Alice sends a message to Bob using a shared key, together with a correspondence assertion concerning authenticity of the message, as outlined in Section 1. A *name* is an identifier that is atomic as far as our analysis is concerned. In this example, the names $Alice$ and Bob identify the two principals, the name K represents a symmetric key known only to $Alice$ and Bob, and the name n represents a public communication channel. A *message*, M or N, is a data structure such as a name, a tuple $(M_1, \ldots, M_n)$, a tagged message $t(M)$, or a ciphertext $\{M\}_N$ (that is, a message M encrypted with a key N, which is typically a name). A *process*, P or Q, is a program that may perform local computations such as encryptions and decryptions, and may communicate with other processes by message-passing on named channels. For example, the process $P_{Alice} = \mathsf{begin}\ sending(Alice, Bob, M); \mathsf{out}\ n\ \{M\}_K$ defines Alice's behaviour. First, she performs a begin-event labelled by the tagged tuple $sending(Alice, Bob, M)$, and then she sends the ciphertext $\{M\}_K$ on the channel n. The process $P_{Bob} = \mathsf{inp}\ n\ (x); \mathsf{decrypt}\ x\ \mathsf{is}\ \{y\}_K; \mathsf{end}\ sending(Alice, Bob, y);$ defines Bob's behaviour. He blocks till a message x arrives on the channel n. Then he attempts to decrypt the message with the key K. We assume there is sufficient redundancy, such as a checksum, in the ciphertext that we can tell whether it was encrypted with K. If so, the plaintext message is bound to y, and he performs an end-event labelled $sending(Alice, Bob, y)$. The process $\mathsf{new}\ (K); (P_{Alice} \mid P_{Bob})$ defines the complete system. The composition $P_{Alice} \mid P_{Bob}$ represents Alice and Bob running in parallel, and able to communicate on shared channels such as n. The binder $\mathsf{new}(K)$ restricts the scope of the key K to the process $P_{Alice} \mid P_{Bob}$ so that no external process may use it. Appendix B contains the grammar of spi messages and processes. The grammar includes the type annotations that are required to appear in spi terms. In this section, we omit the type annotations in spi terms for the purpose of illustrating our approach.

We include begin- and end-events in processes simply to specify correspondence assertions. We say a process is *safe* to mean that in every run, and for every L, there is a distinct, preceding $\mathsf{begin}\ L$ event for every $\mathsf{end}\ L$ event. Our example is safe, because Bob's end-event can only happen after Alice's begin-event.

For correspondence assertions to be interesting, we need to model the possibility of malicious attacks. Let an *opponent* be a spi-calculus process O, arbitrary except that O itself cannot perform begin- or end-events. We say a process P is *robustly safe* if and only if $P \mid O$ is safe for every opponent O. Our example system $\mathsf{new}\ (K); (P_{Alice} \mid P_{Bob})$ is not robustly safe. The opponent cannot acquire the key K since its scope is restricted, but it can intercept messages on the public channel n and mount a replay attack. The opponent $\mathsf{inp}\ n\ (x); \mathsf{out}\ n\ x; \mathsf{out}\ n\ x$ duplicates the encrypted message so that Bob may mistakenly accept M and perform the end-event $sending(Alice, Bob, M)$ twice. To protect against replays, and to achieve robust safety, we can add a nonce handshake to the protocol.

In summary, spi lets us precisely represent the behaviour of protocol participants, and specify authenticity guarantees by process annotations. Robust safety is the property that no opponent at the level of the spi-calculus may violate these guarantees. We omit the details here, but a particular type and effect system verifies robust safety: if a process can be assigned the empty effect, then it is robustly safe. The example above is simple, but the general method works for a wide range of protocol examples [21, 23].

4.2 A Semantics for Local Computation

We translate the values and method bodies of our object calculus to messages and processes, respectively, of the spi calculus. To begin with, we omit web services. Many computational models can be studied by translation to the π-calculus; our translation of local computation follows a fairly standard pattern.

We assume that $Prin$ are π-calculus names, and that $Field \cup Meth \cup Class \cup \{null\}$ are message tags. Values translate easily; in particular, an object translates to a tagged tuple containing the values of its fields.

TRANSLATION OF A VALUE v TO A MESSAGE $[\![v]\!]$:

$$[\![x]\!] \triangleq x$$
$$[\![null]\!] \triangleq null()$$
$$[\![new\ c(v_1, \ldots, v_n)]\!] \triangleq c([\![v_1]\!], \ldots, [\![v_n]\!])$$
$$[\![p]\!] \triangleq p$$

We translate a body b to a process $[\![b]\!]^p_k$ that represents the evaluation of b as principal p. The name k is a continuation, a communications channel on which we send $[\![v]\!]$ to represent termination with value v. Since our focus is representing safety rather than liveness properties, we represent an evaluation that goes wrong simply by the inactive process stop; it would be easy—but a complication—to add an exception mechanism. We use standard split and case statements to analyse tuples and tagged messages, respectively. To call a method ℓ of an object v of class c, with arguments $u_1, \ldots, u_n$ we send the tuple $(p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k)$ on the channel c_ℓ. The name p is the caller, and channel k is the continuation for the call. We translate method ℓ of class c to a process that repeatedly awaits such messages, and triggers evaluations of its body. Our translation depends in part on type information; we write v_c in the translation of field lookups and method calls to indicate that c is the type of v.

TRANSLATION OF A METHOD BODY b TO A PROCESS $[\![b]\!]^p_k$:

$$[\![v]\!]^p_k \triangleq \mathsf{out}\ k\ [\![v]\!]$$
$$[\![let\ x{=}a\ in\ b]\!]^p_k \triangleq \mathsf{new}\ (k'); ([\![a]\!]^p_{k'} \mid \mathsf{inp}\ k'\ (x); [\![b]\!]^p_k)$$
$$[\![if\ u = v\ then\ a\ else\ b]\!]^p_k \triangleq \mathsf{if}\ [\![u]\!] = [\![v]\!]\ \mathsf{then}\ [\![a]\!]^p_k\ \mathsf{else}\ [\![b]\!]^p_k$$
$$[\![v_c.f_j]\!]^p_k \triangleq$$
$\quad \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ null(y); \mathsf{stop}$
$\qquad \mathsf{is}\ c(y); \mathsf{split}\ y\ \mathsf{is}\ (x_1, \ldots, x_n); \mathsf{out}\ k\ x_j$
$\qquad\quad$ where $fields(c) = f_i \mapsto A_i\ ^{i \in 1..n}$, and $j \in 1..n$
$$[\![v_c.\ell(u_1, \ldots, u_n)]\!]^p_k \triangleq$$
$\quad \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ null(y); \mathsf{stop}$
$\qquad \mathsf{is}\ c(y); \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k)$

TRANSLATION OF METHOD ℓ OF CLASS c:

$I_{class}(c, \ell) \triangleq$
  repeat inp $c\_\ell\ (z)$; split z is $(p, this, x_1, \ldots, x_n, k)$; $[\![b]\!]_k^p$
    where $methods(c)(\ell) = (B(A_1\ x_1, \ldots, A_n\ x_n), b)$

4.3 A Semantics for Web Services

We complete the semantics for our object calculus by translating our cryptographic protocol for calling a web service to the spi-calculus. A new idea is that we embed begin- and end-events in the translation to represent the abstract authenticity guarantees offered by the object calculus.

We assume access to all web methods is at the highest security level AuthEnc from Section 2, providing both authentication and secrecy. Here is the protocol, for p making a web service call $w{:}\ell(u_1, \ldots, u_n)$ to service w owned by q, including the names of continuation channels used at the spi level. Recall that the protocol assumes that the client has a way to query the web service for a nonce. Therefore, we assume that in addition to the methods of $class(w)$, each web service also supports a method $getnonce$, which we implement specially.

$$
\begin{array}{l}
p \rightarrow q \text{ on } w : req(getnonce()), k_1 \\
q \rightarrow p \text{ on } k_1 : res(getnonce(n_q)) \\
p \rightarrow q \text{ on } w : p, \{req(w, \ell(u_1, \ldots, u_n), t, n_q)\}_{K_{pq}}, n_p, k_2 \\
q \rightarrow p \text{ on } k_2 : q, \{res(w, \ell(r), t, n_p)\}_{K_{pq}}
\end{array}
$$

We are assuming there is a shared key K_{pq} for each pair of principals $p, q \in Prin$. For the sake of brevity, we omit the formal description of the type and effect system [23] we rely on, but see Appendix B for a detailed overview. Still, to give a flavour, we can define the type of a shared key K_{pq} as follows:

TYPE OF KEY SHARED BETWEEN CLIENT p AND SERVER q:

CSKey$(p, q) \triangleq$
  SharedKey(Union(
    $req(w$:Un, a:Un, t:Un,
      n_q:Public Response [end $req(p, q, w, a, t)$]),
    $res(w$:Un, r:Un, t:Un,
      n_p:Public Response [end $res(p, q, w, r, t)$])))

The type says we can use the key in two modes. First, we may encrypt a plaintext tagged req containing four components: a public name w of a service, an argument a suitable for the service, a session tag t, and a nonce n_q proving that a begin-event labelled $req(p, q, w, a, t)$ has occurred, and therefore that an end-event with that label would be safe. Second, we may encrypt a plaintext tagged res containing four components: a service w, a result r from that service, the session tag t, and a nonce n_p proving that a begin-event labelled $res(p, q, w, r, t)$ has occurred.

We translate a service call to the client-side of our cryptographic protocol as follows. We start by embedding a begin-event labelled $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$ to record the details of client p's call to server $q = owner(w)$. We request a nonce n_q, and use it to freshen the encrypted request, which we send with our own nonce n_p, which the server uses to freshen its response. If the response indeed contains our nonce, we embed an end-event to record successful authentication. For the sake of brevity, we rely on some standard shorthands for pattern-matching.

SEMANTICS OF WEB METHOD CALL:

$[\![w{:}\ell(u_1, \ldots, u_n)]\!]_k^p \triangleq$
  new (k_1, k_2, t, n_p);
  begin $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$;
  out w $(req(getnonce()), k_1)$;
  inp k_1 $(res(getnonce(n_q)))$;
  out w $(p, \{req(w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t, n'_q)\}_{K_{pq}}, n_p, k_2)$;
  inp k_2 (q, bdy);
  decrypt bdy is $\{res(plain)\}_{K_{pq}}$;
  match $plain$ is $(w, rest)$;
  split $rest$ is $(r, rest')$;
  match $rest'$ is (t, n'_p);
  check n_p is n'_p;
  end $res(p, q, w, r, t)$;
  case r is $\ell(x)$; out $k\ x$
where $q = owner(w)$

Our server semantics relies on a shorthand notation defined below; let x=call$_w(p, \ell(u_1, \ldots, u_n)); P$ runs the method ℓ of the class $class(w)$ implementing the service w, with arguments u_1, …, u_n, and with its $CallerId$ field set to p, binds the result to x and runs P.

SERVER-SIDE INVOCATION OF A WEB METHOD:

let x=call$_w(p, args); P \triangleq$
  new (k);
    case $args$
    (is $\ell_i(xs_i)$;
      new (k'); (out c_ℓ_i $(q, c(p), xs_i, k')$ |
                   inp k' (r); out k $\ell_i(r)$)
)$^{i \in 1..n}$
    | inp k (x); P
  where $c = class(w)$, $q = owner(w)$,
   and $methods(c) = \ell_i \mapsto (B_i(As_i, xs_i), b_i)^{\ i \in 1..n}$

Finally, we implement each service w by a process $I_{ws}(w)$. We repeatedly listen for nonce requests, reply with one, and then await a web service call freshened by the nonce. If we find the nonce, it is safe to perform an end-event labelled $req(p, q, w, a, t)$, where p is the caller, $q = owner(w)$ is the service owner, a is the received method request, and t is the session tag. We use the shorthand above to invoke a. If r is the result, we perform a begin-event labelled $res(p, q, w, r, t)$ to record we are returning a result, and then send a response, freshened with the nonce we received from the client. In general, the notation $\prod_{i \in 1..n} P_i$ means $P_1 \mid \cdots \mid P_n$.

WEB SERVICE TRANSLATION:

$I_{ws}(w) \triangleq$ repeat inp w (bdy, k_1);
    case bdy is $req(getnonce())$;
    new (n_q);
    out k_1 $(res(getnonce(n_q)))$;
    inp w $(p', cipher, n_p, k_2)$;
    $\prod_{p \in Prin}$ if $p = p'$ then
    decrypt $cipher$ is $\{req(plain)\}_{K_{pq}}$;
    match $plain$ is $(w, rest)$;
    split $rest$ is (a, t, n'_q);
    check n_q is n'_q;
    end $req(p, q, w, a, t)$;
    let r=call$_w(p, a)$;
    begin $res(p, q, w, r, t)$;
    out k_2 $(q, \{res(w, r, t, n'_p)\}_{K_{pq}})$
  where $q = owner(w)$

This semantics is subject to more deadlocks than a realistic implementation, since we do not have a single database of outstanding nonces. Still, since we are concerned only with safety properties, not liveness, it is not a problem that our semantics is rather more nondeterministic than an actual implementation.

4.4 Security Properties of a Complete System

We define the following process $Sys(b, p)$ to model a piece of code b being run by principal p in the context of implementations of all the classes and web services in $Class$ and $WebService$.

$$
\begin{array}{l}
Sys(b,p) \\
\quad \triangleq \quad \mathsf{new}\ (c\_\ell^{\ c\in Class, \ell\in dom(methods(c))}); \mathsf{new}\ (K_{pq}^{\ p,q\in Prin}); \\
\qquad (\prod_{c\in Class,\ell\in dom(methods(c))} I_{class}(c,\ell) \mid \\
\qquad \prod_{w\in WebService} I_{ws}(w) \mid \\
\qquad \mathsf{new}\ (k); [\![b]\!]_k^p)
\end{array}
$$

We claim that the ways an opponent O can interfere with the behaviour of $Sys(b, p)$ correspond to the ways in which an actual opponent lurking on a network could interfere with SOAP-level messages being routed between web servers. The names c_ℓ of methods are hidden, so O cannot interfere with calls to local methods. The keys K_{pq} are also hidden, so O cannot decrypt or fake SOAP-level encryption. On the other hand, the names w on which $Sys(b, p)$ sends and receives our model of SOAP envelopes are public, and so O is free to intercept, replay, or modify such envelopes.

Our main result is that an opponent cannot disrupt the authenticity properties embedded in our translation. The proof is by showing the translation preserves types.

THEOREM 1. *If body b is well-typed and $p \in Prin$ then $Sys(b, p)$ is robustly safe.*

5. A SOAP-LEVEL IMPLEMENTATION

We have implemented the security abstraction introduced in Section 2 and formalized in Sections 3 and 4 on top of the Visual Studio .NET implementation of web services, as a library that web service developers and clients can use. A web service developer adds security attributes to the web methods of the service. The developer also needs to provide a web method to supply a nonce to the client. On the client side, the client writer is provided with a modified proxy class that encapsulates the implementation of the security abstraction and takes into account the security level of the corresponding web service methods. Hence, from a client's point of view, there is no fundamental difference between accessing a web service with security annotations and one without.

Consider an implementation of our running example of a banking service. Here is what (an extract of) the class implementing the web service looks like:

```
class BankingServiceClass :
        System.Web.Services.WebService
{
  ...
  [WebMethod]
  public int RequestNonce () { ... }

  public DSHeader header;

  [WebMethod]
  [SecurityLevel(Level=SecLevel.Auth)]
  [SoapHeader("header",
              Direction=Direction.InOut,
              Required=true)]
  public int Balance (int account) { ... }
}
```

This is the code we currently have, and it is close to the idealized interface we gave in Section 2. The differences are due to implementation restrictions imposed by the development environment. The extract shows that the web service implements the `RequestNonce` method required by the authentication protocol. The `Balance` method is annotated as an authenticated method, and is also annotated to indicate that the headers of the SOAP messages used during a call will be available through the `header` field of the interface. (The class `DSHeader` has fields corresponding to the headers of the SOAP message.) As we shall see shortly, SOAP headers are used to carry the authentication information. Specifically, the authenticated identity of the caller is available in a web method through `header.callerid`.

To implement the security abstraction on the web service side, we use a feature of Visual Studio .NET called SOAP Extensions. Roughly speaking, a SOAP Extension acts like a programmable "filter". It can be installed on either (or both) of a client or a web service. It gets invoked on every incoming and outgoing SOAP message, and can be used to examine and modify the content of the message before forwarding it to its destination. In our case, the extension will behave differently according to whether the message is incoming or outgoing, and depending on the security level specified. For an outgoing message, if the security level is `None`, the SOAP message is unchanged. If the security level is `Auth`, messages are signed as specified by the protocol: a cryptographic hash of the SOAP body and the appropriate nonce is stored in a custom header of the messages. If the security level is `AuthEnc`, messages are encrypted as specified by the protocol, before being forwarded. For incoming messages, the messages are checked and decrypted, if required. If the security level is `Auth`, the signature of the message checked. If the security level is `AuthEnc`, the message is decrypted before being forwarded. Our implementation uses the SHA1 hash function for signatures, and the RC2 algorithm for symmetric encryption.

To implement the security abstraction on the client side, we provide the client with a new proxy class. The new proxy class provides methods `None`, `Auth`, and `AuthEnc`, that are called by the proxy methods to initiate the appropriate protocol. The method `None` simply sets up the headers of the SOAP message to include the identity of the caller and the callee. `Auth` and `AuthEnc` do the same, but also make a call to the web service to get a nonce and add it (along with a newly created nonce) to the headers. The actual signature and encryption of the SOAP message is again performed using SOAP Extensions, just as on the web service side.

Our implementation uses a custom SOAP header `DSHeader` to carry information such as nonces, identities, and signatures. It provides the following elements:

`callerid`	identity of the client
`calleeid`	identity of the web service provider
`np`	client nonce
`nq`	web service nonce
`signature`	cryptographic signature of the message

Not all of those elements are meaningful for all messages. In addition to these headers, in the cases where the message is encrypted, the SOAP body is replaced by the encrypted body. Appendix A gives actual SOAP messages exchanged between the client and web service during an authenticated call to `Balance`, and an authenticated and encrypted call to `Statement`.

Our implementation is meant as a preliminary design of a C# abstraction for secure RPC, a starting point to explore abstractions

for more general security policies. There are still issues that need to be addressed, even in a setting as simple as the one presented in this paper. First, we plan to adopt recognized formats for encryption and signature of XML data, such as XML-Encryption and XML-Signature (though our validation does not depend on the exact XML syntax for cryptography). Second, it would be valuable to generate the new proxy class automatically.

6. RELATED WORK

There has been work for almost twenty years on secure RPC mechanisms, going back to Birrell [9]. More recently, secure RPC has been studied in the context of distributed object systems. As we mentioned, our work was inspired by the work of van Doorn *et al.* [35], itself inspired by [30, 36]. These techniques (or similar ones) have been applied to CORBA [31], DCOM [10], and Java [7, 18].

In contrast, little work seems to have been done on formalizing secure RPC. Of note is the work of Abadi, Fournet, and Gonthier [2, 3], who show how to compile the standard join-calculus into the sjoin-calculus, and show that the compilation is fully abstract. In a subsequent paper [4], they treat similarly and more simply a join-calculus with authentication primitives: each message contains its source address, there is a way to extract the principal owning a channel from the channel, and any piece of code runs as a particular principal. Their fully abstract translation gives very strong guarantees: it shows that for all intents and purposes, we can reason at the highest level (at the level of the authentication calculus). Although our guarantees are weaker, they are easier to establish.

Duggan [17] formalizes an application-level security abstraction by introducing types for signed and encrypted messages; he presents a fully abstract semantics for the abstraction by translation to a spi-calculus.

Much of the literature on security in distributed systems studies the question of *access control*. Intuitively, access control is the process of determining if the principal calling a particular method has permission to access the objects that the method refers to, according to a particular access control policy. There is a distinction to be made between authentication and access control. Authentication determines whether the principal calling a method is indeed the principal claiming to be calling the method, while access control can use this authenticated identity to determine whether that principal is allowed access. This distinction is made clear in the work of Balfanz *et al.* [7], where they provide authenticated and encrypted communication over Java RMI (using SSL) and use that infrastructure as a basis for a logic-based access control mechanism. The access control decisions are based on the authenticated caller identity obtained from the layer in charge of authentication. This approach is also possible in our framework, which provides access to an authenticated identity as well. We plan to study access control abstractions in our framework. Note that various forms of access control mechanisms have been formalized via π-calculi, [26, 33, 27], and other process calculi [12, 15]. An access control language based on temporal logic has been defined by Sirer and Wang [34] specifically for web services. Damiani *et al.* [14] describe an implementation of an access control model for SOAP; unlike our work, and the WS-Security proposal [6], it relies on an underlying secure channel, such as an SSL connection.

The GRID is a proposed distributed infrastructure with scientific computing as an important application; consequently, the need arises for a distributed security architecture [19] including authentication and access control.

An intense area of activity in the world of web services is the definition of standards for web service security. WS-Security is a standard that describes how to attach signature and encryption headers to SOAP envelopes. Envisioned standards, described in [28], will build on the specifications of WS-Security, for example, to manage and authenticate message exchanges between participants. Our work has an immediate application in this context. It is straightforward, for example, to adapt our implementation to produce WS-Security compliant SOAP envelopes. More importantly, we can use the techniques in this paper to model security abstractions provided by emerging standards and study them formally.

Despite its enjoyable properties, the formal model we use to study the implementation of our security abstraction suffers from some limitations. For instance, it makes the usual Dolev-Yao assumptions that the adversary can compose messages, replay them, or decipher them if it knows the right key, but cannot otherwise "crack" encrypted messages. A more severe restriction is that we cannot yet model insider attacks: principals with shared keys are assumed well-behaved. Work is in progress to extend the Cryptyc type theory to account for malicious insiders. We have not verified the hash-based protocol of Section 2.

7. CONCLUSIONS

Authenticated method calls offer a convenient abstraction for developers of both client and server code. Various authorisation mechanisms may be layered on top of this abstraction. This paper proposes such an abstraction for web services, presents a theoretical model, and describes an implementation using SOAP-level security. By typing our formal semantics, we show no vulnerability exists to attacks representable within the spi-calculus, given certain assumptions. Vulnerabilities may exist outside our model—there are no methods, formal or otherwise, to guarantee security absolutely.

Our work shows that by exploiting recent advances in authenticity types, we can develop a theoretical model of a security abstraction, and then almost immediately obtain precise guarantees. We intend to exploit these ideas further by exploring enriched programming models for authentication and authorisation, while simultaneously building theoretical models and SOAP-level implementations.

This study furthermore validates the adequacy of the spi-calculus, and Cryptyc in particular, to formally reason about security properties in a distributed communication setting.

Acknowledgments

Cryptyc is an ongoing collaboration between Alan Jeffrey and the first author. Ernie Cohen, Cédric Fournet, and Alan Jeffrey made useful suggestions during the writing of this paper.

APPENDIX

A. SAMPLE SOAP MESSAGES

We give some sample SOAP messages exchanged during web service method calls of the web service described in Section 5. One thing that is immediately clear is that we are not using standard XML formats for signing and encrypting messages, such as XML-Encryption and XML-Signature. There is no intrinsic difficulty in adapting our infrastructure to use standard formats. The point is that the validation of the security abstraction does not rely on the exact syntax of the SOAP envelopes.

A.1 An Authenticated Call

We describe an authenticated call to the `Balance` method. The messages exchanged to obtained the nonce are standard SOAP messages. To simplify the presentation of these messages, we have re-

moved some of the namespace information. More specifically, the <soap:Envelope> element carries the following namespaces:

```
xmlns:soap="http://schemas.xmlsoap.org/soap/
envelope"
xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

The following message is the request from Alice to the web service to execute the Balance method on argument 12345. Notice the DSHeader element holding the identity of the principals involved, as well as the nonces and the cryptographic signature.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
  <soap:Header>
    <DSHeader xmlns="http://tempuri.org/">
      <callerid>Alice</callerid>
      <calleeid>Bob</calleeid>
      <np>13</np>
      <nq>42</nq>
      <signature>
        3E:67:75:28:3B:AD:DF:32:E7:6C:D3:66:2A:CF:
        E7:8A:3F:0A:A6:0D
      </signature>
    </DSHeader>
  </soap:Header>
  <soap:Body>
    <Balance xmlns="http://tempuri.org/">
      <account>12345</account>
    </Balance>
  </soap:Body>
</soap:Envelope>
```

The response from the web service has a similar form:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
  <soap:Header>
    <DSHeader xmlns="http://tempuri.org/">
      <callerid>Alice</callerid>
      <calleeid>Bob</calleeid>
      <np>13</np>
      <nq>42</nq>
      <signature>
        8D:31:52:6E:08:F0:89:7B:1E:12:3F:5E:63:EE:
        B0:D2:63:89:CA:73
      </signature>
    </DSHeader>
  </soap:Header>
  <soap:Body>
    <BalanceResponse xmlns="http://tempuri.org/">
      <BalanceResult>100</BalanceResult>
    </BalanceResponse>
  </soap:Body>
</soap:Envelope>
```

A.2 Authenticated and Encrypted Call

We describe an authenticated and encrypted call, this time to the Statement method. Again, the messages exchanged to obtained the nonce are standard SOAP messages. The following message is the request from Alice to the web service to execute the Statement method on argument 12345. As in the authenticated call above, the DSHeader element holds identity information. The body of the message itself is encrypted. Note that the nonce nq must be encrypted according to the protocol, so its encrypted value is included in the encrypted data, and its element is reset to a dummy value (here, -1). Similarly, the signature is unused and set to a dummy value.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
  <soap:Header>
    <DSHeader xmlns="http://tempuri.org/">
      <callerid>Alice</callerid>
      <calleeid>Bob</calleeid>
      <np>13</np>
      <nq>-1</nq>
      <signature>4E:00:6F:00</signature>
    </DSHeader>
  </soap:Header>
  <soap:Body>
    9D:8F:95:2B:BC:60:B1:73:A7:C4:82:F5:39:20:97:
    F7:69:71:66:D3:A3:A0:90:B9:9B:FE:71:0A:65:C1:
    EF:EE:99:CB:4D:8A:40:37:CA:1E:D0:03:50:34:76:
    8C:E3:F3:30:DD:C9:34:19:D4:04:CB:39:7D:1A:84:
    2F:CA:30:DA:68:7E:E1:CB:07:9C:EB:79:F9:E9:4B:
    47:5B:94:56:D7:22:0E:02:CD:AA:F5:D3:40:C1:EC:
    13:FB:B9:E6:4F:13:CD:70:FD:BA:18:80:FC:50:F3:
    75:F2:2F:95:50:5D:41:7E:C8:8B:BB:AB:76:C9:59:
    BA:E2:3B:E5:4D:79:71:E4:AD:18:5A:4B:EA:29:17:
    30:90:66:08:27:ED:B4:BD:2E:89:06:6D:0B:56:40:
    43:35:A1:77:AE:12:7E:4B:19:26:B5:24:1A:D9:67:
    3D:A0:91
  </soap:Body>
</soap:Envelope>
```

The response is similarly encoded. Notice that this time the nonce np must be encrypted, so its value is again included in the encrypted data, and its element is reset to a dummy value.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
  <soap:Header>
    <DSHeader xmlns="http://tempuri.org/">
      <callerid>Alice</callerid>
      <calleeid>Bob</calleeid>
      <np>-1</np>
      <nq>-1</nq>
      <signature>4E:00:6F:00</signature>
    </DSHeader>
  </soap:Header>
  <soap:Body>
    98:FD:6A:5B:38:0A:82:95:3F:01:EC:D3:55:F9:AA:
    35:4D:18:DB:1B:7D:9D:FE:3F:78:52:29:99:C9:41:
    84:EE:B1:42:12:B2:02:AC:63:F5:0C:92:9B:DB:75:
    FB:6C:8B:65:EB:3C:42:6B:79:70:AF:61:2A:C2:7B:
    ED:96:E1:D6:7A:F6:D2:0C:DF:BC:2A:4C:93:B3:D0:
    7B:7D:2D:83:18:60:D2:D8:05:EB:73:74:2D:75:A2:
    B2:57:C9:04:B4:C1:E6:66:54:BA:42:86:AF:22:72:
    3D:B7:90:CF:03:22:E5:C4:47:03:F0:77:A0:30:01:
    C9:FE:78:A1:AB:FA:B1:CB:EE:E2:0B:F2:79:17:1B:
    8E:82:E2:13:F4:66:52:76:6D:BA:1B:E9:8E:75:15:
    90:37:0A:64:ED:F3:9C:18:94:EC:4F:CF:61:92:38:
    EF:A9:46:E8:4E:E9:4A:E6:8A:C9:5E:ED:A7:34:72:
    3E:72:A2:BE:0D:DC:07:22:45:B0:E6:79:33:8F:CD:
    90:B8:97:DB:BA:3B:B2:8B:38:38:B6:5B:F1:11:FB:
    DD:88:CE:9A:3E:B4:E6:31:13:CB:1C:F3:B5:17:D8:
    9B:CF:2E:65:23:4D:BA:ED:72:6D:F4:53:97:B8:7A:
    D2:9C:2C:10:58:A3:0E:FE:48:A2:2A:2A:57:AE:6D:
    69:4D:97:90:EF:9F:C6:7E:9B
  </soap:Body>
</soap:Envelope>
```

B. THE SPI-CALCULUS IN MORE DETAIL

We give an overview of the language and type system on which our analysis of web services depends. We give the syntax in detail, but for the sake of brevity give only an informal account of the operational semantics and type system. Full details are in a technical report [23], from which some of the following explanations are drawn.

NAMES, MESSAGES:

$k ::=$ Encrypt | Decrypt	key attribute
m, n, x, y, z	name: nonce, key, key-pair
$L, M, N ::=$	message
x	name
$(M_1, \ldots, M_n)$	record, $n \geq 0$
$t_i(M)$	tagged union
$\{M\}_N$	symmetric encryption
$\{\!\lvert M \rvert\!\}_N$	asymmetric encryption
$k\,(M)$	key-pair component

The message x is a name, representing a channel, nonce, symmetric key, or asymmetric key-pair. We do not differentiate in the syntax or operational semantics between key-pairs used for public key cryptography and those used for digital signatures.

The message $(M_1, \ldots, M_n)$ is a record with n fields, M_1, ..., M_n.

The message $t_i(M)$ is message M tagged with tag t_i. The message $\{M\}_N$ is the ciphertext obtained by encrypting the plaintext M with the symmetric key N.

The message $\{\!|M|\!\}_N$ is the ciphertext obtained by encrypting the plaintext M with the asymmetric encryption key N.

The message $\mathsf{Decrypt}\ (M)$ is the decryption key (or signing key) component of the key-pair M, and $\mathsf{Encrypt}\ (M)$ is the encryption key (or verification key) component of the key-pair M.

TYPES AND EFFECTS:

$\ell ::= \mathsf{Public} \mid \mathsf{Private}$	nonce attribute
$S, T, U ::=$	type
$\quad \mathsf{Un}$	data known to the opponent
$\quad (x_1{:}T_1, \ldots, x_n{:}T_n)$	dependent record, $n \geq 0$
$\quad \mathsf{Union}(t_1(T_1), \ldots, t_n(T_n))$	tagged union
$\quad \mathsf{Top}$	top
$\quad \mathsf{SharedKey}(T)$	shared-key type
$\quad \mathsf{KeyPair}(T)$	asymmetric key-pair
$\quad k\ \mathsf{Key}(T)$	encryption or decryption part
$\quad \ell\ \mathsf{Challenge}\ \mathit{es}$	challenge type
$\quad \ell\ \mathsf{Response}\ \mathit{fs}$	response type
$e, f ::=$	atomic effect
$\quad \mathsf{end}\ L$	end-event labelled L
$\quad \mathsf{check}\ \ell\ N$	name-check for a nonce N
$\quad \mathsf{trust}\ M{:}T$	trust that $M{:}T$
$\mathit{es}, \mathit{fs} ::=$	effect
$\quad [e_1, \ldots, e_n]$	multiset of atomic effects

The type Un describes messages that may flow to or from the opponent, which we model as an arbitrary process of the calculus. We say that a type is *public* if messages of the type may flow to the opponent. Dually, we say a type is *tainted* if messages from the opponent may flow into the type. The type Un is both public and tainted.

The type $(x_1{:}T_1, \ldots, x_n{:}T_n)$ describes a record $(M_1, \ldots, M_n)$ where each $M_i : T_i$. The scope of each variable x_i consists of the types $T_{i+1}, \ldots, T_n$. Type $(x_1{:}T_1, \ldots, x_n{:}T_n)$ is public just if all of the types T_i are public, and tainted just if all of the types T_i are tainted.

The type $\mathsf{Union}(t_1(T_1), \ldots, t_n(T_n))$ describes a tagged message $t_i(M)$ where $i \in 1..n$ and $M : T_i$. Type $\mathsf{Union}(t_1(T_1), \ldots, t_n(T_n))$ is public just if all of the types T_i are public, and tainted just if all of the types T_i are tainted.

The type Top describes all well-typed messages; it is tainted but not public.

The type $\mathsf{SharedKey}(T)$ describes symmetric keys for encrypting messages of type T; it is public or tainted just if T is both public and tainted.

The type $\mathsf{KeyPair}(T)$ describes asymmetric key-pairs for encrypting or signing messages of type T; it is public or tainted just if T is both public and tainted. The key-pair can be used for public-key cryptography just if T is tainted, and for digital signatures just if T is public.

The type $\mathsf{Encrypt\ Key}(T)$ describes an encryption or signing key for messages of type T; it is public just if T is tainted, and it is tainted just if T is public.

The type $\mathsf{Decrypt\ Key}(T)$ describes a decryption or verification key for messages of type T; it is public just if T is public, and it is tainted just if T it tainted.

The types $\ell\ \mathsf{Challenge}\ \mathit{es}$ and $\ell\ \mathsf{Response}\ \mathit{fs}$ describe nonce challenges and responses, respectively. The effects es and fs embedded in these types represent certain events. An outgoing challenge of some type $\ell\ \mathsf{Challenge}\ \mathit{es}$ can be cast into a response of type $\ell\ \mathsf{Response}\ \mathit{fs}$ and then returned, provided the events in the effect $\mathit{es} + \mathit{fs}$ have been justified, as explained below. Therefore, if we have created a fresh challenge at type $\ell\ \mathsf{Challenge}\ \mathit{es}$, and check that it equals an incoming response of type $\ell\ \mathsf{Response}\ \mathit{fs}$, we can conclude that the events in $\mathit{es} + \mathit{fs}$ may safely be performed. The attribute ℓ is either Public or $\mathsf{Private}$; the former means the nonce may eventually be public, while the latter means the nonce is never made public. Type $\mathsf{Public\ Challenge}\ \mathit{es}$ is public, or tainted, just if $\mathit{es} = [\,]$. Type $\mathsf{Public\ Response}\ \mathit{fs}$ is always public, but tainted just if $\mathit{es} = [\,]$. Neither $\mathsf{Private\ Challenge}\ \mathit{es}$ nor $\mathsf{Private\ Response}\ \mathit{fs}$ is public; both are tainted.

An effect es is a multiset, that is, an unordered list of atomic effects, e or f. Effects embedded in challenge or response types signify that certain actions are justified, that is, may safely be performed. An atomic effect $\mathsf{end}\ L$ justifies a single subsequent end-event labelled L, and is justified by a distinct, preceding begin-event labelled L. An atomic effect $\mathsf{check}\ \ell\ N$ justifies a single subsequent check that an ℓ response equals an ℓ challenge named N, where ℓ is Public or $\mathsf{Private}$, and is justified by freshly creating the challenge N. An atomic effect $\mathsf{trust}\ M{:}T$ justifies casting message M to type T, and is justified by showing that M indeed has type T.

PROCESSES:

$O, P, Q, R ::=$	process
$\quad \mathsf{out}\ M\ N$	output
$\quad \mathsf{inp}\ M\ (x{:}T); P$	input
$\quad \mathsf{repeat\ inp}\ M\ (x{:}T); P$	replicated input
$\quad \mathsf{split}\ M\ \mathsf{is}\ (x_1{:}T_1, \ldots, x_n{:}T_n); P$	record splitting
$\quad \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}T); P$	pair matching
$\quad \mathsf{case}\ M\ \mathsf{is}\ t_i(x_i{:}T_i); P_i\ ^{i \in 1..n}$	tagged union case
$\quad \mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional (new)
$\quad \mathsf{new}\ (x{:}T); P$	name generation
$\quad P \mid Q$	composition
$\quad \mathsf{stop}$	inactivity
$\quad \mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P$	symmetric decrypt
$\quad \mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}T|\!\}_{N^{-1}}; P$	asymmetric decrypt
$\quad \mathsf{check}\ M\ \mathsf{is}\ N; P$	nonce-checking
$\quad \mathsf{begin}\ L; P$	begin-assertion
$\quad \mathsf{end}\ L; P$	end-assertion
$\quad \mathsf{cast}\ M\ \mathsf{is}\ (x{:}T); P$	cast to nonce type
$\quad \mathsf{witness}\ M{:}T; P$	witness testimony
$\quad \mathsf{trust}\ M\ \mathsf{is}\ (x{:}T); P$	trusted cast

The processes $\mathsf{out}\ M\ N$ and $\mathsf{inp}\ M\ (x{:}T); P$ are output and input, respectively, along an asynchronous, unordered channel M. If an output $\mathsf{out}\ x\ N$ runs in parallel with an input $\mathsf{inp}\ x\ (y); P$, the two can interact to leave the residual process $P\{y{\leftarrow}N\}$, the outcome of substituting N for each free occurrence of y in P. We write $\mathsf{out}\ x\ (M); P$ as a simple shorthand for $\mathsf{out}\ x\ M \mid P$.

The process $\mathsf{repeat\ inp}\ M\ (x{:}T); P$ is replicated input, which behaves like input, except that each time an input of N is performed, the residual process $P\{y{\leftarrow}N\}$ is spawned off to run concurrently with the original process $\mathsf{repeat\ inp}\ M\ (x{:}T); P$.

The process $\mathsf{split}\ M\ \mathsf{is}\ (x_1{:}T_1, \ldots, x_n{:}T_n); P$ splits the record M into its n components. If M is $(M_1, \ldots, M_n)$, the process behaves as $P\{x_1{\leftarrow}M_1\} \cdots \{x_n{\leftarrow}M_n\}$. Otherwise, it deadlocks, that is, does nothing.

The process match M is $(N, y{:}U); P$ splits the pair (binary record) M into its two components, and checks that the first one is N. If M is (N, L), the process behaves as $P\{y{\leftarrow}L\}$. Otherwise, it deadlocks.

The process case M is $t_i(x_i{:}T_i); P_i\ ^{i\in 1..n}$ checks the tagged union M. If M is $t_j(L)$ for some $j \in 1..n$, the process behaves as $P\{x_i{\leftarrow}L\}$. Otherwise, it deadlocks.

The process if $M = N$ then P else Q behaves as P if M and N are the same message, and otherwise as Q. (This process is not present in the original calculus [23] but is a trivial and useful addition.)

The process new $(x{:}T); P$ generates a new name x, whose scope is P, and then runs P. This abstractly represents nonce or key generation.

The process $P \mid Q$ runs processes P and Q in parallel.

The process stop is deadlocked.

The process decrypt M is $\{x{:}T\}_N; P$ decrypts M using symmetric key N. If M is $\{L\}_N$, the process behaves as $P\{x{\leftarrow}L\}$. Otherwise, it deadlocks. We assume there is enough redundancy in the representation of ciphertexts to detect decryption failures.

The process decrypt M is $\{|x{:}T|\}_{N^{-1}}; P$ decrypts M using asymmetric key N. If M is $\{|L|\}_{\mathsf{Encrypt}\ (K)}$ and N is Decrypt (K), then the process behaves as $P\{x{\leftarrow}L\}$. Otherwise, it deadlocks.

The process check M is $N; P$ checks the messages M and N are the same name before executing P. If the equality test fails, the process deadlocks.

The process begin $L; P$ autonomously performs a begin-event labelled L, and then behaves as P.

The process end $L; P$ autonomously performs an end-event labelled L, and then behaves as P.

The process cast M is $(x{:}T); P$ binds the message M to the variable x of type T, and then runs P. In well-typed programs, M is a challenge of type ℓ Challenge es, and T is a response type ℓ Challenge fs. This is the only way to populate a response type.

The process witness $M{:}T; P$ simply runs P, but is well-typed only if M has the type T. This is the only way to justify a trust $M{:}T$ effect.

The process trust M is $(x{:}T); P$ binds the message M to the variable x of type T, and then runs P. In well-typed programs, this cast is justified by a previous run of a witness $M{:}T; Q$ process.

Next, we recall the notions of process safety, opponents, and robust safety introduced in Section 4. The notion of a run of a process can be formalized by an operational semantics.

SAFETY:

A process P is *safe* if and only if
  for every run of the process and for every L,
    there is a distinct begin L event for every end L event.

OPPONENTS AND ROBUST SAFETY:

A process P is *assertion-free* if and only if
  it contains no begin- or end-assertions.
A process P is *untyped* if and only if
  the only type occurring in P is Un.
An *opponent* O is an assertion-free untyped process.
A process P is *robustly safe* if and only if
  $P \mid O$ is safe for every opponent O.

Our problem, then, is to show that processes representing protocols are robustly safe. We appeal to a type and effect system to establish robust safety (but not to define it). The system involves the following type judgments.

JUDGMENTS $E \vdash \mathcal{J}$:

$E \vdash \diamond$	good environment
$E \vdash \mathit{es}$	good effect es
$E \vdash T$	good type T
$E \vdash M : T$	good message M of type T
$E \vdash P : \mathit{es}$	good process P with effect es

We omit the rules defining these judgments, which can be found in [23]; our previous informal explanation of types should give some intuitions.

We made two additions to the language as defined in [23], namely the empty record type $()$ (and corresponding empty record message $()$), and the conditional form if $M = N$ then P else Q. The empty record type can be handled by simply extending the typing rules for records to the case where there are no elements. The main consequence of this is that the type $()$ will be isomorphic to the type Un, by the extended subtyping rules. The extension of spi to handle the conditional is similarly straightforward, except that we need to actually add a transition rule to the operational semantics, and a new typing rule to propagate the effects. For completeness, we describe the additions here, with the understanding that they rely on terminology defined and explained in [23]:

EXTENSIONS TO SPI FOR THE CONDITIONAL:

$[\text{if } M = N \text{ then } P_{true} \text{ else } P_{false}] + \mathit{As} \rightarrow [P_{M=N}] + \mathit{As}$ — transition rule

(Proc If)

$$\frac{E \vdash M : \mathsf{Top} \quad E \vdash N : \mathsf{Top} \quad E \vdash P : \mathit{es} \quad E \vdash Q : \mathit{fs}}{E \vdash \text{if } M = N \text{ then } P \text{ else } Q : \mathit{es} \vee \mathit{fs}}$$ typing rule

The type and effect system can guarantee the robust safety of a process, according to the following theorem [23]:

THEOREM 2 (ROBUST SAFETY). *If* $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : [\,]$ *then* P *is robustly safe.*

C. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure communications implementation of channel abstractions. In *13th IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 105–116, 1998.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 74–88, 1999.

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

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

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

[7] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 15–26. IEEE Computer Society Press, 2000.

[8] T. Barclay, J. Gray, E. Strand, S. Ekblad, and J. Richter. TerraService.NET: An introduction to web services. Technical Report MS–TR–2002–53, Microsoft Research, June 2002.

[9] A. D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, 1985.

[10] D. Box. *Essential COM*. Addison Wesley Professional, 1997.

[11] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. Available from `http://www.w3.org/TR/SOAP`, 2000.

[12] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.

[13] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.2. Available from `http://www.w3.org/TR/2002/WD-wsdl12-20020709`, 2002.

[14] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing soap e-services. *International Journal of Information Security (IJIS)*, 1(2):100–115, 2002.

[15] R. De Nicola, G. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming 1999*, volume 1603 of *Lecture Notes in Computer Science*, pages 117–146. Springer, 1999.

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

[17] D. Duggan. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop*, pages 238–252. IEEE Computer Society Press, 2002.

[18] P. Eronen and P. Nikander. Decentralized Jini security. In *Proceedings of Network and Distributed System Security 2001 (NDSS2001)*, pages 161–172, 2001.

[19] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *5th ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[20] Google. Google Web APIs (beta). `http://www.google.com/apis`, July 2002.

[21] A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 145–159. IEEE Computer Society Press, 2001. Extended version to appear in *Journal of Computer Security*.

[22] A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics 17*, volume 45 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001. Extended version to appear in *Theoretical Computer Science*.

[23] A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop*, pages 77–91. IEEE Computer Society Press, 2002. An extended version appears as Technical Report MSR–TR–2002–31, Microsoft Research, August 2002.

[24] A.D. Gordon and R. Pucella. Validating a web service security abstraction by typing. Technical Report MS–TR–2002–108, Microsoft Research, October 2002.

[25] A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 248–260, 2001.

[26] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings HLCL'98*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[27] D. Hoshina, E. Sumii, and A. Yonezawa. A typed process calculus for fine-grained resource access control in distributed computation. In *Fourth International Symposium on Theoretical Aspects of Computer Software (TACS2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2001.

[28] IBM Corporation and Microsoft Corporation. Security in a web services world: A proposed architecture and roadmap. White paper available from `http://msdn.microsoft.com/library/en-us/dnwssecur/html/securitywhitepaper.asp`, April 2002.

[29] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA '99)*, pages 132–146. ACM Press, 1999.

[30] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[31] U. Lang and R. Schreiner. *Developing Secure Distributed Systems with CORBA*. Artech House, 2002.

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

[33] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 1998.

[34] E. G. Sirer and K. Wang. An access control language for web services. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 23–30. ACM Press, 2002.

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

[36] T. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

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