Under consideration for publication in Formal Aspects of Computing

Validating a Web Service Security Abstraction by Typing

Andrew D. Gordon[1] and Riccardo Pucella[2]

[1]Microsoft Research
[2]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.

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.

Correspondence and offprint requests to: Andrew D. Gordon, Microsoft Research, Roger Needham Building, 7 J J Thomson Ave, Cambridge CB3 0FB, UK. E-mail: adg@microsoft.com

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* [12]. 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. For example, 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 [14], 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 SOAP itself does not define any 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 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. In particular, 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 [17], using type theories developed as part of the Cryptyc project [22, 23, 21].

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 [22], 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.
- Section 6 shows how we can accommodate public-key infrastructures to implement the abstraction of Section 2.

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.

Many formal details, as well as the proofs of our formal results, have been relegated to the appendices. Specifically, Appendix A gives sample messages exchanged during web service method calls using our abstractions, Appendix B gives a formal description of our object calculus, Appendix C gives a formal definition of the spi-calculus used in the paper, Appendix D gives the proofs of our formal results, and Appendix E describes an extension of our object calculus to capture a form of first-class web services.

A part of this article, in preliminary form, appears as a conference paper [24].

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).

We use the language C$^{\#}$ to present our security abstraction. (There is nothing specific to C$^{\#}$ in our approach, although the implementation we describe in this section and in Section 5 takes advantage of some features of the language.) 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 Section 6).

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:

$$\begin{array}{l}
p \rightarrow q : \text{request nonce} \\
q \rightarrow p : n_q \\
p \rightarrow q : p, \mathit{req}(w, \ell(u_1, \ldots, u_n), s, n_q), n_p, \mathit{Hash}(\mathit{req}(w, \ell(u_1, \ldots, u_n), s, n_q), K_{pq}) \\
q \rightarrow p : q, \mathit{res}(w, \ell(r), s, n_p), \mathit{Hash}(\mathit{res}(w, \ell(r), s, n_p), K_{pq})
\end{array}$$

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:

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

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 arithmetic, 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 $w \in \mathit{WebService}$ as a URL referring to the service; moreover, $\mathit{class}(w)$ is the name of the class that implements the service, and $\mathit{owner}(w) \in \mathit{Prin}$ 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 $\mathit{Alice}, \mathit{Bob} \in \mathit{Prin}$, and a web service $w = \mathit{http://bob.com/BankingService}$, where we have $\mathit{owner}(w) = \mathit{Bob}$ and $\mathit{class}(w) = \mathit{BankingServiceClass}$. 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
  Id CallerId
  Num Balance(Num account)
    if account = 12345 then
      if this.CallerId = Alice then 100 else null
    else ...
```

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 $w{:}\mathit{Balance}(12345)$ 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 call $\mathit{new\ BankingServiceClass}(\mathit{Alice}).\mathit{Balance}(12345)$ as *Bob*. In other words, *Bob* creates a new object of the form$\mathit{new\ BankingServiceClass}(\mathit{Alice})$ (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 arguments 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 $\mathit{proxy}(w)$. The class $\mathit{proxy}(w)$ 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:

```
class BankingServiceProxy
  Id Id()
    Bob
  Num Balance(Num account)
    w:Balance(account)
```

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:

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

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

Types and Method Signatures:

$A, B \in \mathit{Type} ::=$	type
$\quad \mathit{Id}$	principal identifier
$\quad c$	object
$\mathit{sig} \in \mathit{Sig} ::= B(A_1\ x_1, \ldots, A_n\ x_n)$	method signature (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 $X \overset{\text{fin}}{\to} Y$ for the sets of total functions and finite maps, respectively, from X to Y.

Execution Environment: (*fields*, *methods*, *owner*, *class*)

$\mathit{fields} \in \mathit{Class} \to (\mathit{Field} \overset{\text{fin}}{\to} \mathit{Type})$	fields of a class
$\mathit{methods} \in \mathit{Class} \to (\mathit{Meth} \overset{\text{fin}}{\to} \mathit{Sig} \times \mathit{Body})$	methods of a class
$\mathit{owner} \in \mathit{WebService} \to \mathit{Prin}$	service owner
$\mathit{class} \in \mathit{WebService} \to \mathit{Class}$	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
$u, v \in \mathit{Value} ::=$	value
$\quad x$	variable
$\quad \mathit{null}$	null
$\quad \mathit{new}\ c(v_1, \ldots, v_n)$	object
$\quad p$	principal identifier
$a, b \in \mathit{Body} ::=$	method body
$\quad v$	value
$\quad \mathit{let}\ x{=}a\ \mathit{in}\ b$	let-expression
$\quad \mathit{if}\ u = v\ \mathit{then}\ a\ \mathit{else}\ b$	conditional

$v.f$	field lookup
$v.\ell(u_1, \dots, u_n)$	method call
$w{:}\ell(u_1, \dots, u_n)$	service call

The free variables $\mathit{fv}(a)$ of a method body are defined in the usual way, where the only binder is x being bound in b in the expression $\mathit{let}\ x{=}a\ \mathit{in}\ b$. 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 $\mathit{let}\ x{=}a\ \mathit{in}\ b$ to be equal to $\mathit{let}\ x'{=}a\ \mathit{in}\ b\{x{\leftarrow}x'\}$, if $x' \notin \mathit{fv}(b)$.

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, let the term $\mathit{if}\ a_1 = a_2\ \mathit{then}\ b_1\ \mathit{else}\ b_2$ be short for $\mathit{let}\ x_1{=}a_1\ \mathit{in}\ \mathit{let}\ x_2{=}a_2\ \mathit{in}\ \mathit{if}\ x_1 = x_2\ \mathit{then}\ b_1\ \mathit{else}\ b_2$. We already used this when writing $\mathit{if}\ \mathit{this.CallerId} = \mathit{Alice}\ \mathit{then}\ 100\ \mathit{else}\ \mathit{null}$ 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 explore an extension of our model to account for web services as “first-class values” in Appendix E.

We assume all method bodies in our execution environment are well-typed. If $\mathit{methods}(c)(\ell) = (\mathit{sig}, b)$ and the signature $\mathit{sig} = B(A_1\, x_1, \dots, A_n\, x_n)$ we assume that the body b has type B given a typing environment $\mathit{this}{:}c, x_1{:}A_1, \dots, x_n{:}A_n$. The variable *this* refers to the object on which the ℓ method was invoked. The type system is given by a typing judgment $E \vdash a : A$, saying that a has type A in an environment E of the form $x_1{:}A_1, \dots, x_n{:}A_n$ that gives a type to the free variables in a. The domain $\mathit{dom}(E)$ of E is the set of variables $\{x_1, \dots, x_n\}$ given a type in E. The typing rules, which are standard, are given in Appendix B. We also assume the class $\mathit{class}(w)$ 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.) A formal account of this semantics, as well as the typing rules of the calculus, can be found in Appendix B.

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

To evaluate a let-expression $\mathit{let}\ x{=}a\ \mathit{in}\ b$ 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 $\mathit{if}\ u = v\ \mathit{then}\ a\ \mathit{else}\ b$ 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 $\mathit{new}\ c(v_1, \dots, v_n)$, we check f is the jth field of class c for some $j \in 1..n$ (that is, that $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$ 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 $v.\ell(u_1, \dots, u_n)$ as p, when v is an object $\mathit{new}\ c(v_1,\ \dots,\ v_n)$, we check ℓ is a method of class c (that is, that $\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)\ ^{i \in 1..m}$ and that $\ell = \ell_j$ for some $j \in 1..m$) and we check the arity of its signature is n (that is, that $\mathit{sig}_j = B(A_1\, x_1, \dots, A_n\, x_n)$) and then we evaluate the method body as p, but with the object v itself in place of the variable *this*, and actual parameters $u_1, \dots, u_n$ in place of the formal parameters $x_1, \dots, x_n$ (that is, we evaluate the expression $b_i\{\mathit{this}{\leftarrow}v, x_1{\leftarrow}u_1, \dots, x_n{\leftarrow}u_n\}$). If v is null or if either check fails, evaluation has gone wrong.

To evaluate a service call $w{:}\ell(u_1, \dots, u_n)$ as p, we evaluate the local method call $\mathit{new}\ c(p).\ell(u_1, \dots, u_n)$ as q, where $c = \mathit{class}(w)$ is the class implementing the service, and $q = \mathit{owner}(w)$ 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 calculus by translation to the spi-calculus [5, 22], 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 C contains the grammar of spi messages and processes. The grammar includes the type annotations that are required to appear in spi terms.

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 [22, 21].

For the sake of clarity, we defer some of the technical details to the appendices. Specifically, Appendix C contains more details on the spi-calculus and the type and effect sytem, as well as a formal definition of robust safety; Appendix D gives a proof of our technical results.

4.2. A Semantics for Local Computation

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

We use the notation $[\![\,]\!]$ to represent the translation of the types and terms of our object calculus to appropriate types, messages, and processes in the spi calculus. In many places, we also define abbreviations in the spi calculus (for instance, we define $\mathsf{let}\ x{=}\mathsf{call}_w(p, \mathit{args}); P$ as shorthand for a more complex spi calculus process); these do not use the $[\![\,]\!]$ notation.

We assume that *Prin* are spi-calculus names, and that $\mathit{Field} \cup \mathit{Meth} \cup \mathit{Class} \cup \{\mathit{null}\}$ are message tags. The translations for types is straightforward. Since principal identifiers are presumably known to the opponent, the type of identifiers corresponds to the spi type Un. A value of class c is either the value *null*, or a tagged tuple $\mathit{new}\ c(v_1, \ldots, v_n)$. As we shall see below, we translate *null* to a tagged empty tuple $\mathit{null}()$, and an object to a tagged tuple $c(v_1, \ldots, v_n)$. Thus, a class c translates to a tagged union type with components $\mathit{null}(\mathsf{Un})$ and $c(\mathsf{Un})$. (The types Un indicate that the content of the tuples are presumably known to the opponent.)

Type Translation:

$\mathsf{Prin} \triangleq \mathsf{Un}$
$[\![\mathit{Id}]\!] \triangleq \mathsf{Prin}$
$[\![c]\!] \triangleq \mathsf{Union}(\mathit{null}(\mathsf{Un}), c(\mathsf{Un}))$

Environment Translation:

$[\![x_1{:}A_1, \ldots, x_n{:}A_n]\!] \triangleq x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]$

If $\mathit{As} = A_1, \ldots, A_n$ and $\mathit{xs} = x_1, \ldots, x_n$ we sometimes write $B(\mathit{As}\ \mathit{xs})$ as shorthand for the signature $B(A_1\ x_1, \ldots, A_n\ x_n)$. We define two shorthands for types corresponding to web method calls. The type $\mathit{Req}(w)$ represents the type of possible calls to web methods provided by the service w; the type of a call is simply the translated type of the arguments of the web method, tagged with the name of the method. Similarly, the type $\mathit{Res}(w)$ represents the type of the results of web methods provided by the service w; the type of a result of a call is simply the translated type of the result of the web method, tagged once again with the name of the method.

Request and Response Types:

$[\![A_1, \ldots, A_m]\!] \triangleq [\![A_1]\!], \ldots, [\![A_m]\!]$
$\mathit{Req}(w) \triangleq \mathsf{Union}(\ell_i([\![\mathit{As}_i]\!])\ ^{i \in 1..n})$
  where $\mathit{class}(w) = c$ and $\mathit{methods}(c) = \ell_i \mapsto (B_i(\mathit{As}_i \mathit{xs}_i), b_i)\ ^{i \in 1..n}$
$\mathit{Res}(w) \triangleq \mathsf{Union}(\ell_i([\![B_i]\!])\ ^{i \in 1..n})$
  where $\mathit{class}(w) = c$ and $\mathit{methods}(c) = \ell_i \mapsto (B_i(\mathit{As}_i \mathit{xs}_i), b_i)\ ^{i \in 1..n}$

The translation of expressions really acts on the type derivation of an expression, not just the expression itself. This means that during the translation of an expression, we have access to the types of the subexpressions appearing in the expression. To reduce clutter, we write the translation as though it is acting on the expression itself, except that when we need access to the type of a subexpression, we annotate the appropriate subexpression with its type. For example, the translation of $\mathit{let}\ x{=}a\ \mathit{in}\ b$ depends on the type of a, which is available through the type derivation of $E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : B$. We write $\mathit{let}\ x{=}a_A\ \mathit{in}\ b$ to indicate that the type of a is A, according to the type derivation. 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 $[\![v]\!]$:

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

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. We defer the translation of web method calls until Section 4.3. 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 $[\![b]\!]^p_k$:

$[\![v]\!]^p_k \triangleq \mathsf{out}\ k\ [\![v]\!]$
$[\![\mathit{let}\ x{=}a_A\ \mathit{in}\ b]\!]^p_k \triangleq \mathsf{new}\ (k'{:}\mathsf{Un}); ([\![a]\!]^p_{k'} \mid \mathsf{inp}\ k'\ (x{:}\mathsf{Un}); [\![b]\!]^p_k)$
$[\![\mathit{if}\ u = v\ \mathit{then}\ a\ \mathit{else}\ b]\!]^p_k \triangleq \mathsf{if}\ [\![u]\!] = [\![v]\!]\ \mathsf{then}\ [\![a]\!]^p_k\ \mathsf{else}\ [\![b]\!]^p_k$
$[\![v_c.f_j]\!]^p_k \triangleq \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ \mathit{null}(y{:}\mathsf{Un}); \mathsf{stop}$
$\qquad \mathsf{is}\ c(y{:}\mathsf{Un}); \mathsf{split}\ y\ \mathsf{is}\ (x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]); \mathsf{out}\ k\ x_j$
$\qquad\qquad \text{where } \mathit{fields}(c) = f_i \mapsto A_i\ {}^{i\in 1..n}, \text{ and } j \in 1..n$
$[\![v_c.\ell(u_1, \ldots, u_n)]\!]^p_k \triangleq \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ \mathit{null}(y{:}\mathsf{Un}); \mathsf{stop}$
$\qquad \mathsf{is}\ c(y{:}\mathsf{Un}); \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k)$

Translation of Method ℓ of Class c:

$I_{\mathit{class}}(c, \ell) \triangleq \mathsf{repeat\ inp}\ c\_\ell\ (z{:}\mathsf{Un});$
$\qquad \mathsf{split}\ z\ \mathsf{is}\ (p{:}\mathsf{Prin}, \mathit{this}{:}\mathsf{Un}, x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!], k{:}\mathsf{Un}); [\![b]\!]^p_k$
$\qquad\quad \text{where } \mathit{methods}(c)(\ell) = (B(A_1\ x_1, \ldots, A_n\ x_n), b)$

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 $\mathit{class}(w)$, each web service also supports a method $\mathit{getnonce}$, which we implement specially.

$p \to q$ on $w : \mathit{req}(\mathit{getnonce}()), k_1$
$q \to p$ on $k_1 : \mathit{res}(\mathit{getnonce}(n_q))$
$p \to q$ on $w : p, \{\mathit{req}(w, \ell(u_1, \ldots, u_n), t, n_q)\}_{K_{pq}}, n_p, k_2$
$q \to p$ on $k_2 : q, \{\mathit{res}(w, \ell(r), t, n_p)\}_{K_{pq}}$

We are assuming there is a shared key K_{pq} for each pair of principals $p, q \in \mathit{Prin}$. For the sake of brevity, we omit the formal description of the type and effect system [21] we rely on, but see Appendix C 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:

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

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 $\mathit{req}(p,q,w,a,t)$ 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 $\mathit{res}(p,q,w,r,t)$ 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 $\mathit{req}(p,q,w,\ell([\![u_1]\!],\ldots,[\![u_n]\!]),t)$ to record the details of client p's call to server $q = \mathit{owner}(w)$. 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.

Translation of Web Method Call:

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

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 $\mathit{class}(w)$ implementing the service w, with arguments $u_1, \ldots, u_n$, 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$_w(p, \mathit{args}); P \triangleq$
  new ($k$);
  (case $\mathit{args}$ (is $\ell_i(\mathit{xs}_i)$; new ($k'$); (out $c\_\ell_i$ ($q, c(p), \mathit{xs}_i, k'$) | inp $k'$ ($r$); out $k$ $\ell_i(r)$)) $^{i\in 1..n}$
    | inp $k$ ($x$); $P$)
  where $c = \mathit{class}(w)$, $q = \mathit{owner}(w)$,
   and $\mathit{methods}(c) = \ell_i \mapsto (B_i(\mathit{As}_i, \mathit{xs}_i), b_i)$ $^{i\in 1..n}$
```

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 $\mathit{req}(p,q,w,a,t)$, where p is the caller, $q = \mathit{owner}(w)$ 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 $\mathit{res}(p,q,w,r,t)$ 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 Implementation:

$$
\begin{array}{l}
I_{ws}(w) \triangleq \mathsf{repeat\ inp}\ w\ (bdy{:}\mathsf{Un}, k_1{:}\mathsf{Un});\\
\quad \mathsf{case}\ bdy\ \mathsf{is}\ req(getnonce());\\
\quad \mathsf{new}\ (n_q{:}\mathsf{Public\ Challenge}\ []);\\
\quad \mathsf{out}\ k_1\ (res(getnonce(n_q)));\\
\quad \mathsf{inp}\ w\ (p'{:}\mathsf{Un}, cipher{:}\mathsf{Un}, n_p{:}\mathsf{Un}, k_2{:}\mathsf{Un});\\
\quad \prod_{p \in Prin} \mathsf{if}\ p = p'\ \mathsf{then}\\
\quad \mathsf{decrypt}\ cipher\ \mathsf{is}\ \{req(plain)\}_{K_{pq}};\\
\quad \mathsf{match}\ plain\ \mathsf{is}\ (w, rest{:}\\
\quad\quad (a{:}Req(w), t{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,a,t)]));\\
\quad \mathsf{split}\ rest\ \mathsf{is}\ (a{:}Req(w), t{:}\mathsf{Un}, n'_q{:}\\
\quad\quad \mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,a,t)]);\\
\quad \mathsf{check}\ n_q\ \mathsf{is}\ n'_q; \mathsf{end}\ req(p,q,w,a,t);\\
\quad \mathsf{let}\ r{:}Res(w) {=} \mathsf{call}_w(p,a);\\
\quad \mathsf{begin}\ res(p,q,w,r,t);\\
\quad \mathsf{cast}\ n_p\ \mathsf{is}\ (n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t)]); \mathsf{out}\ k_2\ (q, \{res(w,r,t,n'_p)\}_{K_{pq}})\\
\quad \text{where } q = owner(w)
\end{array}
$$

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 process $Sys(b, p, k)$ to model a piece of code b being run by principal p (with continuation k) in the context of implementations of all the classes and web services in *Class* and *WebService*. The implementation of the classes and web services are given as follows.

Implementation of Classes and Web Services:

$$
\begin{array}{l}
ClMeth \triangleq \{(c, \ell)\ :\ c \in Class, \ell \in dom(methods(c))\}\\
I_{class} \triangleq \prod_{(c,\ell) \in ClMeth} I_{class}(c, \ell)\\
I_{ws} \triangleq \prod_{w \in WebService} I_{ws}(w)
\end{array}
$$

The process $Sys(b, p, k)$ is defined with respect to an environment that specifies the type of its free variables, such as the names of the web services, principals, classes and methods, and keys.

Top-Level Environments:

$$
\begin{array}{ll}
E_{class} \triangleq (c\_\ell{:}\mathsf{Un})^{\ (c,\ell) \in ClMeth} & \\
E_{keys} \triangleq (K_{pq}{:}\mathsf{CSKey}(p,q))^{\ p,q \in Prin} & \\
E_{ws} \triangleq (w{:}\mathsf{Un})^{\ w \in WebService} & \\
E_{prin} \triangleq p_1{:}\mathsf{Prin}, \ldots, p_n{:}\mathsf{Prin} & \text{where } Prin = \{p_1, \ldots, p_n\}\\
E_0 \triangleq E_{ws}, E_{prin}, E_{class}, E_{keys} &
\end{array}
$$

The process $Sys(b, p, k)$ is defined as follows:

$$Sys(b, p, k) \triangleq \mathsf{new}\ (E_{class}, E_{keys}); (I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![b]\!]^p_k)$$

We claim that the ways an opponent O can interfere with the behaviour of $Sys(b, p, k)$ 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 $Sys(b, p, k)$ 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 $\varnothing \vdash b : B$ and $p \in \mathit{Prin}$ and $k \notin \mathit{dom}(E_0)$ then the system $\mathit{Sys}(b, p, k)$ is robustly safe.

Proof. See Appendix D.1. □

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 Microsoft 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`.

6. A Semantics Using Asymmetric Cryptography

The security abstraction we describe in Section 2 relies on shared keys between principals. This is hardly a reasonable setup in modern systems. In this section, we show that our approach can easily accommodate public-key infrastructures.

6.1. Authenticated Web Methods

We start by describing the protocol and implementation for authenticated web methods. Hence, for now, we assume that all the exported methods of a web service are annotated with `Auth`.

Consider a simple public-key infrastructure for digital signatures. Each principal p has a signing key SKp and a verification key VKp. The signing key is kept private, while the verification key is public. To bind the name of a principal with their verification key, we assume a *certification authority* CA (itself with a signing key $SKCA$ and verification key $VKCA$) that can sign certificates $CertVKp$ of the form $\{\!|p, VKp|\!\}_{SKCA}$. (The notation $\{\!|\cdot|\!\}_K$ is used to represent both asymmetric encryption and signature, differentiating it from symmetric encryption. In the case where $\{\!|M|\!\}_K$ represent a signature, this is simply notation for M along with a token representing the signature of M with asymmetric key K.)

Here is a protocol that uses digital signatures to authenticate messages, 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. Again, we assume that in addition to the methods of $class(w)$, each web service also supports a method $getnonce$, which we implement specially.

$p \rightarrow q$ on $w : CertVKp, n_p, req(getnonce()), k_1$
$q \rightarrow p$ on $k_1 : CertVKq, res(getnonce(n_q))$
$p \rightarrow q$ on $w : p, \{\!|req(w, \ell(u_1, \ldots, u_n), t, q, n_q)|\!\}_{SKp}, k_2$
$q \rightarrow p$ on $k_2 : q, \{\!|res(w, \ell(r), t, p, n_p)|\!\}_{SKq}$

Type of Signing Keys:

$\mathsf{AuthMsg}(p) \triangleq$
$\quad \mathsf{Union}(req(w{:}\mathsf{Un}, a{:}\mathsf{Un}, t{:}\mathsf{Un}, q{:}\mathsf{Un}, n_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, a, t)]),$
$\qquad\quad res(w{:}\mathsf{Un}, r{:}\mathsf{Un}, t{:}\mathsf{Un}, q{:}\mathsf{Un}, n_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(q, p, w, r, t)]))$
$\mathsf{AuthKeys}(p) \triangleq \mathsf{KeyPair}(\mathsf{AuthMsg}(p))$
$\mathsf{AuthCert} \triangleq (p : \mathsf{Un}, \mathsf{Decrypt\ Key}(\mathsf{AuthMsg}(p)))$
$\mathsf{AuthCertKeys} \triangleq \mathsf{KeyPair}(\mathsf{AuthCert})$

We will represent the key pair of a signing key and verification key for principal p by a pair DSp, of type

AuthKeys(p). The key pair for the certification authority will be represented by a pair $DSCA$. We use the following abbreviations:

Key and Certificates Abbreviations:

$SKp \triangleq \mathsf{Encrypt}\ (DSp)$	p's signing key
$VKp \triangleq \mathsf{Decrypt}\ (DSp)$	p's verification key
$CertVKp \triangleq \{\!\mid p,\ VKp \mid\!\}_{SKCA}$	p's certificate

With that in mind, we can amend the translation of Section 4 to accommodate the new protocol. First, we give a new translation for a web method call $w{:}\ell(u_1, \ldots, u_n)$:

New Translation of Web Method Call:

```
$[\![w{:}\ell(u_1, \ldots, u_n)]\!]^p_k \triangleq$
  new $(k_1{:}\mathsf{Un}, k_2{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Challenge}\ [\,])$;
  begin $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$;
  out $w\ (CertVKp, n_p, req(getnonce()), k_1)$;
  inp $k_1\ (c{:}\mathsf{Un}, res(getnonce(n_q{:}\mathsf{Un})))$;
  decrypt $c$ is $\{\!\mid cert{:}(q'{:}\mathsf{Un}, \mathsf{Decrypt\ Key(AuthMsg}(q'))) \mid\!\}_{VKCA^{-1}}$;
  match $cert$ is $(q, vkq{:}\mathsf{Decrypt\ Key(AuthMsg}(q)))$;
  cast $n_q$ is $(n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)])$;
  out $w\ (p, \{\!\mid req(w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t, q, n'_q) \mid\!\}_{SKp}, k_2)$;
  inp $k_2\ (q''{:}\mathsf{Un}, bdy{:}\mathsf{Un})$;
  decrypt $bdy$ is $\{\!\mid res(plain{:}(w'{:}\mathsf{Un}, r{:}\mathsf{Un}, t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                                    $\mathsf{Public\ Response}\ [\mathsf{end}\ res(p', q, w', r, t')])) \mid\!\}_{vkq^{-1}}$;
  match $plain$ is $(w, rest{:}(r{:}Res(w), t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                                $\mathsf{Public\ Response}\ [\mathsf{end}\ res(p', q, w, r, t')]))$;
  split $rest$ is $(r{:}Res(w), rest'{:}(t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                                    $\mathsf{Public\ Response}\ [\mathsf{end}\ res(p', q, w, r, t')]))$;
  match $rest'$ is $(t, rest''{:}(p'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ res(p', q, w, r, t)]))$;
  match $rest''$ is $(p, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t)])$;
  check $n_p$ is $n'_p$;
  end $res(p, q, w, r, t)$;
  case $r$ is $\ell(x)$; out $k\ x$
where $q = owner(w)$
```

We also need to give a new implementation for web services, again to take into account the different messages being exchanged:

New Web Service Implementation:

```
$I_{ws}(w) \triangleq$
  repeat inp $w\ (c{:}\mathsf{Un}, n_p{:}\mathsf{Un}, bdy{:}\mathsf{Un}, k_1{:}\mathsf{Un})$;
  case $bdy$ is $req(getnonce())$;
  decrypt $c$ is $\{\!\mid p{:}\mathsf{Un}, vkp{:}\mathsf{Decrypt\ Key(AuthMsg}(p)) \mid\!\}_{VKCA^{-1}}$;
  new $(n_q{:}\mathsf{Public\ Challenge}\ [\,])$;
  out $k_1\ (CertVKq, res(getnonce(n_q)))$;
  inp $w\ (p'{:}\mathsf{Un}, cipher{:}\mathsf{Un}, k_2{:}\mathsf{Un})$;
  if $p = p'$ then
  decrypt $cipher$ is $\{\!\mid req(plain{:}(w{:}\mathsf{Un}, a{:}\mathsf{Un}, t{:}\mathsf{Un}, q'{:}\mathsf{Un},$
                                        $\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q', w, a, t)])) \mid\!\}_{vkp^{-1}}$;
  match $plain$ is $(w, rest{:}(a{:}Req(w), t{:}\mathsf{Un}, q'{:}\mathsf{Un},$
                                $\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q', w, a, t)]))$;
```

```
split $rest$ is $(a{:}Req(w),$
        $t{:}\mathsf{Un}, rest'{:}(q'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q',w,a,t)]))$;
match $rest'$ is $(q, n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,a,t)])$;
check $n_q$ is $n'_q$;
end $req(p,q,w,a,t)$;
let $r{:}Res(w){=}\mathsf{call}_w(p,a)$;
begin $res(p,q,w,r,t)$;
cast $n_p$ is $(n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t)])$;
out $k_2\ (q, \{\!|res(w,r,t,p,n'_p)|\!\}_{SKq})$
where $q = owner(w)$
```

Finally, we need to change the top-level environment to account for the new keys, and to add a channel through which we will publish the public keys.

Top-Level Environments:

$E_{class} \triangleq (c\_\ell{:}\mathsf{Un})\ ^{(c,\ell)\in ClMeth}$
$E_{keys} \triangleq DSCA{:}\mathsf{AuthCertKeys}, (DSp{:}\mathsf{AuthKeys}(p))\ ^{p\in Prin}$
$E_{ws} \triangleq (w{:}\mathsf{Un})\ ^{w\in WebService}$
$E_{prin} \triangleq p_1{:}\mathsf{Prin}, \ldots, p_n{:}\mathsf{Prin}$ where $Prin = \{p_1, \ldots, p_n\}$
$E_{net} \triangleq net{:}\mathsf{Un}$
$E_0 \triangleq E_{ws}, E_{prin}, E_{net}, E_{class}, E_{keys}$

Publishing can be achieved by simply sending the public keys on a public channel, here net:

Public Keys Publishing:

$I_{net} \triangleq \mathsf{out}\ net\ (VKCA, (VKp)\ ^{p\in Prin})$

We can now establish that the resulting system is robustly safe:

Theorem 2. If $\varnothing \vdash a : A$ and $p \in Prin$ and $k \notin dom(E_0)$ then the system

$$\mathsf{new}\ (E_{class}, E_{keys}); (I_{net} \mid I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k)$$

is robustly safe.

Proof. See Appendix D.2. □

The protocol we give above to provide authentication has some undesirable properties. Specifically, it requires the server to remember the certificate $CertVKp$ and nonce n_p at the time when a nonce is requested. Since anyone can request a nonce, and no authentication is performed at that stage of the protocol, this makes the server severely vulnerable to denial-of-service attacks. The following variation on the protocol achieves the same guarantees, but pushes the exchange of certificates and nonces to later messages, basically just when they are needed.

$p \to q$ on $w : req(getnonce()), k_1$
$q \to p$ on $k_1 : res(getnonce(n_q))$
$p \to q$ on $w : p, CertVKp, n_p, \{\!|req(w, \ell(u_1, \ldots, u_n), t, q, n_q)|\!\}_{SKp}, k_2$
$q \to p$ on $k_2 : q, CertVKq, \{\!|res(w, \ell(r), t, p, n_p)|\!\}_{SKq}$

6.2. Authenticated and Encrypted Web Methods

We now describe a protocol and implementation for authenticated and encrypted web methods. Hence, for now, we assume that all the exported methods of a web service are annotated with `AuthEnc`.

The public-key infrastructure we consider for this case is similar to the one for authenticated web methods, except that now we have encryption and decryption keys, as opposed to signing and verification keys. Each principal p has an encryption key EKp and a decryption key DKp. The decryption key is kept private, while the encryption key is public. To bind the name of a principal with their encryption key, we again assume a *certification authority CA* (with a signing key *SKCA* and verification key *VKCA*) that can sign certificates *CertEKp* of the form $\{\!|p, \mathit{EKp}|\!\}_{\mathit{SKCA}}$.

Here is a 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. Again, we assume that in addition to the methods of $\mathit{class}(w)$, each web service also supports a method *getnonce*, which we implement specially.

$$
\begin{array}{l}
p \to q \text{ on } w : \mathit{CertEKp}, \mathit{req}(\mathit{getnonce}()), k_1 \\
q \to p \text{ on } k_1 : \mathit{CertEKq}, \{\!|\mathit{msg_2}(q, n_K)|\!\}_{\mathit{EKp}}, \mathit{res}(\mathit{getnonce}(n_q)) \\
p \to q \text{ on } w : \{\!|\mathit{msg_3}(w, p, K, n_K)|\!\}_{\mathit{EKq}}, n_p, \{\mathit{req}(\ell(u_1, \ldots, u_n), t, n_q)\}_K, k_2 \\
q \to p \text{ on } k_2 : \{\mathit{res}(\ell(r), t, n_p)\}_K
\end{array}
$$

This protocol is similar to that for authenticated web methods, except that public key encryption is used to exchange a session-specific shared key K used to encrypt the actual method call. Specifically, in the third message, p chooses a session-specific shared key K, and sends it to q encrypted with q's public key EKq; this session key K is used to encrypt the web method call. The result of the web method call is also encrypted with this shared key. To prevent replay attacks, the shared key is bound to a nonce n_K sent by q in the second message.

Type of Keys:

$$
\begin{array}{l}
\mathsf{SKey}(p, q, w) \triangleq \\
\quad \mathsf{SharedKey}(\mathsf{Union}(\mathit{req}(a{:}\mathsf{Un}, t{:}\mathsf{Un}, n_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{req}(p, q, w, a, t)]), \\
\qquad\qquad\qquad\qquad \mathit{res}(r{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p, q, w, r, t)]))) \\
\mathsf{AuthEncMsg}(p) \triangleq \\
\quad \mathsf{Union}(\mathit{msg_2}(q{:}\mathsf{Un}, n_K{:}\mathsf{Private\ Challenge}\ []), \\
\qquad\qquad \mathit{msg_3}(w{:}\mathsf{Un}, q{:}\mathsf{Un}, K{:}\mathsf{Top}, \\
\qquad\qquad\qquad n_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p, q, w)])) \\
\mathsf{AuthEncKeys}(p) \triangleq \mathsf{KeyPair}(\mathsf{AuthEncMsg}(p)) \\
\mathsf{AuthEncCert} \triangleq (p{:}\mathsf{Un}, \mathsf{Encrypt\ Key}(\mathsf{AuthEncMsg}(p))) \\
\mathsf{AuthEncCertKeys} \triangleq \mathsf{KeyPair}(\mathsf{AuthEncCert})
\end{array}
$$

We will represent the key pair of an encryption key and decryption key for principal p by a pair *PKp*, of type $\mathsf{AuthEncKeys}(p)$. The signing key pair for the certification authority will be represented by a pair *DSCA*. We use the following abbreviations:

Key and Certificates Abbreviations:

$\mathit{EKp} \triangleq \mathsf{Encrypt}\ (\mathit{PKp})$	p's encryption key
$\mathit{DKp} \triangleq \mathsf{Decrypt}\ (\mathit{PKp})$	p's decryption key
$\mathit{CertEKp} \triangleq \{\!|p, \mathit{EKp}|\!\}_{\mathit{SKCA}}$	p's certificate

Again, we can amend the translation of Section 4 to accommodate the new protocol. First, we give a new translation for a web method call $w{:}\ell(u_1, \ldots, u_n)$:

New Translation of Web Method Call:

$$
\begin{array}{l}
[\![w{:}\ell(u_1, \ldots, u_n)]\!]^p_k \triangleq \\
\quad \mathsf{new}\ (k_1{:}\mathsf{Un}, k_2{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Challenge}\ []); \\
\quad \mathsf{begin}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t); \\
\quad \mathsf{out}\ w\ (\mathit{CertEKp}, \mathit{req}(\mathit{getnonce}()), k_1); \\
\quad \mathsf{inp}\ k_1\ (c{:}\mathsf{Un}, \mathit{cipher}{:}\mathsf{Un}, \mathit{res}(\mathit{getnonce}(n_q{:}\mathsf{Un})));
\end{array}
$$

```
  decrypt $c$ is $\{\!| cert{:}(q'{:}\mathsf{Un}, \mathsf{Encrypt\ Key(AuthEncMsg}(q'))) |\!\}_{VKCA^{-1}}$;
  match $cert$ is $(q, ekq{:}\mathsf{Encrypt\ Key(AuthEncMsg}(q)))$;
  decrypt $cipher$ is $\{\!| msg_2(q'{:}\mathsf{Un}, n_K{:}\mathsf{Un}) |\!\}_{DKp^{-1}}$;
  if $q = q'$ then
  cast $n_q$ is $(n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,\ell([\![u_1]\!],\ldots,[\![u_n]\!]),t)])$;
  new $(K{:}\mathsf{SKey}(p,q,w))$;
  witness $K{:}\mathsf{SKey}(p,q,w)$;
  cast $n_K$ is $(n'_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p,q,w)])$;
  out $w$ $(\{\!| msg_3(w,p,t,K,n'_K) |\!\}_{ekq}, n_p, \{req(w,\ell([\![u_1]\!],\ldots,[\![u_n]\!]),t,n'_q)\}_K, k_2)$;
  inp $k_2$ $(bdy{:}\mathsf{Un})$;
  decrypt $bdy$ is $\{res(plain{:}(r{:}Res(w), t'{:}\mathsf{Un},$
                                   $\mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t')])))\}_K$;
  match $plain$ is $(r{:}Res(w), rest{:}(t'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t')]))$;
  match $rest$ is $(t, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t)])$;
  check $n_p$ is $n'_p$;
  end $res(p,q,w,r,t)$;
  case $r$ is $\ell(x)$; out $k$ $x$
where $q = owner(w)$
```

We also need to give a new implementation for web services, again to take into account the different messages being exchanged:

New Web Service Implementation:

```
$I_{ws}(w) \triangleq$
  repeat inp $w$ $(c{:}\mathsf{Un}, bdy{:}\mathsf{Un}, k_1{:}\mathsf{Un})$;
  case $bdy$ is $req(getnonce())$;
  decrypt $c$ is $\{\!| p{:}\mathsf{Un}, ekp{:}\mathsf{Encrypt\ Key(AuthEncMsg}(p)) |\!\}_{VKCA^{-1}}$;
  new $(n_q{:}\mathsf{Public\ Challenge}\ [\,])$;
  new $(n_K{:}\mathsf{Private\ Challenge}\ [\,])$;
  out $k_1$ $(CertEKq, \{\!| msg_2(q, n_K) |\!\}_{ekp}, res(getnonce(n_q)))$;
  inp $w$ $(cipher_1{:}\mathsf{Un}, n_p{:}\mathsf{Un}, cipher_2{:}\mathsf{Un}, k_2{:}\mathsf{Un})$;
  decrypt $cipher_1$
        is $\{\!| msg_3(plain_1{:}(w{:}\mathsf{Un}, p'{:}\mathsf{Un}, K{:}\mathsf{Top},$
                              $\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p',q,w)])) |\!\}_{DKq^{-1}}$;
  match $plain_1$ is $(w, rest{:}(p'{:}\mathsf{Un}, K{:}\mathsf{Top},$
                                 $\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p',q,w)]))$;
  match $rest$ is $(p, rest'{:}(K{:}\mathsf{Top}, \mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p,q,w)]))$;
  split $rest'$ is $(K{:}\mathsf{Top}, n'_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p,q,w)])$;
  check $n_K$ is $n'_K$;
  trust $K$ is $(K'{:}\mathsf{SKey}(p,q,w))$;
  decrypt $cipher_2$ is $\{req(plain_2{:}(a{:}Req(w), t{:}\mathsf{Un},$
                                     $\mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,a,t)])))\}_{K'}$;
  split $plain_2$ is $(a{:}Req(w), t{:}\mathsf{Un}, n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p,q,w,a,t)])$;
  check $n_q$ is $n'_q$;
  end $req(p,q,w,a,t)$;
  let $r{:}Res(w)$=$\mathsf{call}_w(p,a)$;
  begin $res(p,q,w,r,t)$;
  cast $n_p$ is $(n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p,q,w,r,t)])$;
  out $k_2$ $\{res(r,t,n'_p)\}_{K'}$
where $q = owner(w)$
```

Finally, we need to change the top-level environment to account for the new keys, and to add a channel through which we will publish the public keys.

Top-Level Environments:

$E_{class} \triangleq (c\_\ell{:}\mathsf{Un})^{(c,\ell)\in ClMeth}$
$E_{keys} \triangleq DSCA{:}\mathsf{AuthEncCertKeys}, (PKp{:}\mathsf{AuthEncKeys}(p))^{p\in Prin}$
$E_{ws} \triangleq (w{:}\mathsf{Un})^{w\in WebService}$
$E_{prin} \triangleq p_1{:}\mathsf{Prin}, \ldots, p_n{:}\mathsf{Prin}$ where $Prin = \{p_1, \ldots, p_n\}$
$E_{net} \triangleq net{:}\mathsf{Un}$
$E_0 \triangleq E_{ws}, E_{prin}, E_{net}, E_{class}, E_{keys}$

Publishing can be achieved by simply sending the public keys on a public channel, here *net*:

Public Keys Publishing:

$I_{net} \triangleq \mathsf{out}\ net\ (VKCA, (EKp)^{p\in Prin})$

We can now establish that the resulting system is robustly safe:

Theorem 3. If $\varnothing \vdash a : A$ and $p \in Prin$ and $k \notin dom(E_0)$ then the system

$$\mathsf{new}\ (E_{class}, E_{keys}); (I_{net} \mid I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k)$$

is robustly safe.

Proof. See Appendix D.3. □

We can note some further possibilities, with respect to the protocols implemented in this section:

- The protocol implementing authenticated and encrypted invocation uses certificates to essentially negotiate a symmetric key with which to actually perform the encryption. It is straightforward to apply the same idea to the authenticated-only case, negotiating a symmetric key with which to hash the content of the method call (instead of relying on public-key signatures).
- In the above protocol, a new symmetric key is negotiated at every method invocation. A more efficient variation would be to re-use a negotiated symmetric key over multiple web method calls. Once a symmetric key has been negotiated, it can effectively act as a shared key between the two principals, which is the case we investigated in the body of this paper. We can therefore use the above protocol for the first web method call between a principal and a particular service, and the shared-key protocol for subsequent web method calls.

7. Related Work

There has been work for almost twenty years on secure RPC mechanisms, going back to Birrell [10]. 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 [11], and Java [7, 19].

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 [18] 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. Various forms of access control mechanisms have been formalized via π-calculi, [26, 33, 27], and other process calculi [13, 16]. An access control language based on temporal logic has been defined by Sirer and Wang [34] specifically for web services. Damiani *et al.* [15] describe an implementation of an access control model for SOAP; unlike our work, and the WS-Security proposal, it relies on an underlying secure channel, such as an SSL connection.

Since this work was completed, a series of specifications for web services security has been published, as laid out in a whitepaper from IBM and Microsoft [28]. In particular, WS-Security [6] defines how to add signatures, to apply encryption, and to add principal identities, such as usernames or certificates, to a SOAP envelope. It would be straightforward, for example, to adapt our implementation to produce WS-Security compliant SOAP envelopes. A recent paper shows how to formalize the authentication goals of protocols based on WS-Security using the applied π-calculus [9].

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.

8. 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.

While our approach is restricted to proving properties of protocols that can be established using the Cryptyc type and effect system, it is worth pointing out that it is compatible with alternative methods for protocol verification. For instance, it is possible to analyze the protocols we use to implement secure web method calls for security flaws beyond those that can be uncovered using Cryptyc (for instance, flaws involving malicious insiders).

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. (As with many formal analyses, these guarantees concern the design of our abstraction, and do not rule out code defects in its actual implementation.)

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.

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. 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 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">
  <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 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">
  <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 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">
  <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 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">
  <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. Semantics of the Object Calculus

In this appendix, we give a formal description of the operational semantics and typing rules of the object calculus. We first describe some encodings showing the expressiveness of the calculus.

B.1. Encoding Arithmetic

The calculus is simple enough that questions about whether or not it is sufficiently expressive to be of interest arise. This is especially likely since there are no recursive functions in the calculus, and it is not clear that it is even Turing complete. That the calculus indeed is Turing complete is a consequence of the fact that we can write recursive classes and methods, and that we have a null object. The following example shows an encoding of natural numbers as a class *Num*, with the typical recursive definition of addition:

```
class Num
  Num pred
  Num succ()
    new Num(this)
  Num add(Numx)
    if x.pred = null then
      this
     else this.add(x.pred).succ()
```

We define *zero* as *new Num*(*null*), *one* as *zero*.*succ*(), and so on.

B.2. Formalization of proxy objects

We mentioned in the text that we can easily express proxy objects within the calculus. For completeness, here is a detailed formalization of such proxy objects. First, we assume a map $proxy \in WebService \to Class$, assigning to every web service $w \in WebService$ a proxy class $proxy(w)$. We further assume that for each $w \in WebService$,

- $dom(methods(class(w))) \cup \{Id\} = dom(methods(proxy(w)))$,
- $fields(proxy(w)) = \varnothing$,
- $methods(proxy(w)(Id)) = (Id(), owner(w))$, and
- for all $\ell \in dom(methods(class(w)))$,

  $$methods(proxy(w))(\ell) = (B(A_1\, x_1, \ldots, A_n\, x_n), w{:}\ell(x_1, \ldots, x_n)),$$

  where $methods(class(w))(\ell) = (B(A_1\, x_1, \ldots, A_n\, x_n), b)$.

B.3. Operational Semantics

The operational semantics is defined by a transition relation, written $a \to^p a'$, where a and a' are method bodies, and p is the principal evaluating the body a.

To specify the semantics, we need to keep track of which principal is currently running a method body.

We add a new method body form to our object calculus, $p[a]$, meaning p running body a. This form does not appear in code written by the user, but only arises through the transitions of the semantics.

Extended Method Bodies:

$a, b \in \mathit{Body} ::=$	method body
$\quad\cdots$	as in Section 3
$\quad p[a]$	body a running as p

Transitions:

(Red Let 1)
$$\frac{a \to^p a'}{\mathit{let}\ x{=}a\ \mathit{in}\ b \to^p \mathit{let}\ x{=}a'\ \mathit{in}\ b}$$

(Red Let 2)
$$\frac{}{\mathit{let}\ x{=}v\ \mathit{in}\ b \to^p b\{x{\leftarrow}v\}}$$

(Red If)
$$\frac{}{\mathit{if}\ u = v\ \mathit{then}\ a_{\mathit{true}}\ \mathit{else}\ a_{\mathit{false}} \to^p a_{u=v}}$$

(Red Field)
$$\frac{\mathit{fields}(c) = f_i \mapsto A_i\ {}^{i\in 1..n} \quad j \in 1..n}{(\mathit{new}\ c(v_1, \ldots, v_n)).f_j \to^p v_j}$$

(Red Invoke)(where $v = \mathit{new}\ c(v_1, \ldots, v_n)$)
$$\frac{\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)\ {}^{i\in 1..n} \quad j \in 1..n \quad \mathit{sig}_j = B(A_1\ x_1, \ldots, A_m\ x_m)}{v.\ell_j(u_1, \ldots, u_m) \to^p b_j\{\mathit{this}{\leftarrow}v, x_k{\leftarrow}u_k\ {}^{k\in 1..m}\}}$$

(Red Remote)
$$\frac{\mathit{owner}(w) = q \quad \mathit{class}(w) = c}{w{:}\ell(u_1, \ldots, u_n) \to^p q[\mathit{new}\ c(p).\ell(u_1, \ldots, u_n)]}$$

(Red Prin 1)
$$\frac{a \to^q a'}{q[a] \to^p q[a']}$$

(Red Prin 2)
$$\frac{}{q[v] \to^p v}$$

B.4. Type System

The judgments of our type system all depend on an *environment* E, that defines the types of all variables in scope. An environment takes the form $x_1{:}A_1, \ldots, x_n{:}A_n$ and defines the type A_i for each variable x_i. The domain $\mathit{dom}(E)$ of an environment E is the set of variables whose types it defines.

Environments:

$D, E ::=$	environment
$\quad\varnothing$	empty
$\quad E, x{:}A$	entry
$\mathit{dom}(x_1{:}A_1, \ldots, x_n{:}A_n) \triangleq \{x_1, \ldots, x_n\}$	domain of an environment

The following are the two judgments of our type system. They are inductively defined by rules presented in the following tables.

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

$E \vdash \diamond$	good environment
$E \vdash a : A$	good expression a of type A

We write $E \vdash \mathcal{J}$ when we want to talk about both kinds of judgments, where $\mathcal{J}$ stands for either $\diamond$ or $a : A$.

The following rules define an environment $x_1{:}A_1, \ldots, x_n{:}A_n$ to be well-formed if each of the names $x_1, \ldots, x_n$ are distinct.

Rules for Environments:

(Env $\varnothing$) (Env x)(where $x \notin \mathit{dom}(E)$)

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash \diamond}{E, x{:}A \vdash \diamond}$$

We present the rules for deriving the judgment $E \vdash a : A$ that assigns a type A to a value or method body a. These rules are split into two tables, one for values, and one for method bodies.

Rules for Typing Values:

(Val x) (Val *null*)

$$\frac{E = E_1, x{:}A, E_2 \quad E \vdash \diamond}{E \vdash x : A} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{null} : c}$$

(Val Object) (Val Princ)

$$\frac{\mathit{fields}(c) = f_i \mapsto A_i \ ^{i \in 1..n} \quad E \vdash v_i : A_i \quad \forall i \in 1..n}{E \vdash \mathit{new}\ c(v_1, \ldots, v_n) : c} \qquad \frac{E \vdash \diamond}{E \vdash p : \mathit{Id}}$$

Rules for Typing Method Bodies:

(Body Let)

$$\frac{E \vdash a : A \quad E, x{:}A \vdash b : B}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : B}$$

(Body If)

$$\frac{E \vdash u : A \quad E \vdash v : A \quad E \vdash a : B \quad E \vdash b : B}{E \vdash \mathit{if}\ u = v\ \mathit{then}\ a\ \mathit{else}\ b : B}$$

(Body Field)

$$\frac{E \vdash v : c \quad \mathit{fields}(c) = f_i \mapsto A_i \ ^{i \in 1..n} \quad j \in 1..n}{E \vdash v.f_j : A_j}$$

(Body Invoke)

$$\frac{\begin{array}{c} E \vdash v : c \quad \mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i) \ ^{i \in 1..n} \quad j \in 1..n \\ \mathit{sig}_j = B(A_1\, x_1, \ldots, A_m\, x_m) \quad E \vdash u_k : A_k \quad \forall k \in 1..m \end{array}}{E \vdash v.\ell_j(u_1, \ldots, u_m) : B}$$

(Body Remote) (Body Princ)

$$\frac{\begin{array}{c} \mathit{class}(w) = c \quad \mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i) \ ^{i \in 1..n} \quad j \in 1..n \\ \mathit{sig}_j = B(A_1\, x_1, \ldots, A_m\, x_m) \quad E \vdash u_i : A_i \quad \forall i \in 1..m \end{array}}{E \vdash w{:}\ell_j(u_1, \ldots, u_m) : B} \qquad \frac{E \vdash a : A}{E \vdash p[a] : A}$$

We make the following assumption on the execution environment.

Assumptions on the Execution Environment:

(1) For each $w \in \mathit{WebService}$, $\mathit{fields}(\mathit{class}(w)) = \mathit{CallerId} : \mathit{Id}$.
(2) No tagged expression $p[a]$ occurs within the body of any method;
such expressions occur only at runtime, to track the call stack of principals.
(3) for each $c \in \mathit{Class}$ and each $\ell \in \mathit{dom}(\mathit{methods}(c))$,
if $\mathit{methods}(c)(\ell) = (B(A_1\, x_1, \ldots, A_n\, x_n), b)$,
then $\mathit{this}{:}c, x_1{:}A_1, \ldots, x_n{:}A_n \vdash b : B$.

It is straightforward to show that our type system is sound, that is, that the type system ensures that methods that typecheck do not get stuck when evaluating. Some care is needed to make this precise, since evaluation can block if one attempts to access a field of a null object, or to invoke a method on a null object. (We could introduce an *error token* in the semantics and propagate that error token when such a case is encountered, but this would needlessly complicate the semantics, at least for our purposes.) Soundness can be derived as usual via Preservation and Progress theorems. To establish these, we first need the following lemmas:

Lemma 1. The following properties of judgments hold:

(Exchange) if $E, x{:}A, y{:}B, E' \vdash \mathcal{J}$, then $E, y{:}B, x{:}A, E' \vdash \mathcal{J}$;
(Weakening) if $E \vdash \mathcal{J}$ and $x \notin \mathit{dom}(E)$, then $E, x{:}A \vdash \mathcal{J}$;
(Strengthening) if $E, x{:}B \vdash a : A$ and $x \notin \mathit{fv}(a)$, then $E \vdash a : A$.

Proof. Straightforward. $\square$

We often use the above properties silently in the course of proofs.

Lemma 2 (Substitution). If $E, x{:}B \vdash a : A$ and $E \vdash v : B$, then $E \vdash a\{x{\leftarrow}v\} : A$

Proof This is a straightforward proof by induction on the height of the typing derivation for $E \vdash a : A$. We proceed by case analysis on the form of a.

- Case $a = x$: Since $E, x{:}B \vdash x : A$, we must have $A = B$. Since $a\{x{\leftarrow}v\} = v$ and $E \vdash v : B$, we have $E \vdash v : A$, as required.
- Case $a = y$, where $y \neq x$: Since x is not free in y, $E, x{:}B \vdash y : A$ implies $E \vdash y : A$, by the Strengthening Lemma, as required.
- Case $a = \mathit{null}$: Since x is not free in null, $E, x{:}B \vdash \mathit{null} : A$ implies $E \vdash \mathit{null} : A$, as required.
- Case $a = p$: Since x is not free in p, $E, x{:}B \vdash p : A$ implies $E \vdash p : A$, as required.
- Case $a = newc(v_1, \ldots, v_n)$: We have the equation $newc(v_1, \ldots, v_n)\{x{\leftarrow}v\} = newc(v_1\{x{\leftarrow}v\}, \ldots, v_n\{x{\leftarrow}v\})$. We have $E, x{:}B \vdash \mathit{new}\ c(v_1, \ldots, v_n) : A$, hence $E, x{:}B \vdash v_i : A_i$ if $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$. By the induction hypothesis, we know $E \vdash v_i\{x{\leftarrow}v\} : A_i$ for all $i \in 1..n$. Hence, we can derive $E \vdash \mathit{new}\ c(v_1\{x{\leftarrow}v\}, \ldots, v_n\{x{\leftarrow}v\}) : A$, as required.
- Case $a = \mathit{let}\ y{=}a_0\ \mathit{in}\ b$: Without loss of generality, we can take $y \neq x$, since y is bound in b. Note that $(\mathit{let}\ y{=}a_0\ \mathit{in}\ b)\{x{\leftarrow}v\} = \mathit{let}\ y{=}a_0\{x{\leftarrow}v\}\ \mathit{in}\ b\{x{\leftarrow}v\}$. We have $E, x{:}B \vdash \mathit{let}\ y{=}a_0\ \mathit{in}\ b : A$, hence $E, x{:}B \vdash a_0 : A_0$ for some A_0, and $E, y{:}A_0, x{:}B \vdash b : A$. By the induction hypothesis, $E \vdash a_0\{x{\leftarrow}v\} : A_0$ and $E, y{:}A_0 \vdash b\{x{\leftarrow}v\} : A$, and hence $E \vdash \mathit{let}\ x{=}a_0\{x{\leftarrow}v\}\ \mathit{in}\ b\{x{\leftarrow}v\} : A$, as required.
- Case $a = \mathit{if}\ u_0 = u_1\ \mathit{then}\ a_0\ \mathit{else}\ a_1$: We have $(\mathit{if}\ u_0 = u_1\ \mathit{then}\ a_0\ \ \mathit{else}\ a_1)\{x{\leftarrow}v\} = \mathit{if}\ u_0\{x{\leftarrow}v\} = u_1\{x{\leftarrow}v\}\ \mathit{then}\ a_0\{x{\leftarrow}v\}\ \mathit{else}\ a_1\{x{\leftarrow}v\}$. We have $E, x{:}B \vdash \mathit{if}\ u_0 = u_1\ \mathit{then}\ a_0\ \mathit{else}\ a_1$, hence $E, x{:}B \vdash u_0 : A'$, $E, x{:}B \vdash u_1 : A'$, $E, x{:}B \vdash a_0 : A$ and $E, x{:}B \vdash a_1 : A$. Applying the induction hypothesis to these judgments, we can derive

  $$E \vdash \mathit{if}\ u_0\{x{\leftarrow}v\} = u_1\{x{\leftarrow}v\}\ \mathit{then}\ a_0\{x{\leftarrow}v\}\ \mathit{else}\ a_1\{x{\leftarrow}v\} : A$$

  as required.

The remaining cases are similar, upon noting that:

- $(u.f_j)\{x{\leftarrow}v\} = u\{x{\leftarrow}v\}.f_j$,
- $(u.\ell(u_1, \ldots, u_m))\{x{\leftarrow}v\} = u\{x{\leftarrow}v\}(u_1\{x{\leftarrow}v\}, \ldots, u_m\{x{\leftarrow}v\})$,
- $(w{:}\ell(u_1, \ldots, u_n))\{x{\leftarrow}v\} = w{:}(u_1\{x{\leftarrow}v\}, \ldots, u_n\{x{\leftarrow}v\})$, and
- $(p[a])\{x{\leftarrow}v\} = p[a\{x{\leftarrow}v\}]$. $\square$

Theorem 4 (Preservation). If $E \vdash a : A$ and $a \rightarrow^p a'$ then $E \vdash a' : A$.

Proof We proceed by induction on the height of the typing derivation for $E \vdash a : A$. Since $a \rightarrow^p a'$, a cannot be a value v.

- Case $a = \mathit{let}\ x{=}v\ \mathit{in}\ b$: Since $E \vdash a : A$, we have $E \vdash v : B$ and $E, x{:}B \vdash b : A$. We must have $a' = b\{x{\leftarrow}v\}$. Applying the Substitution Lemma, we have $E \vdash b\{x{\leftarrow}v\} : A$, as required.

- Case $a = \mathit{let}\ x{=}a_0\ \mathit{in}\ b$, where a_0 is not a value: We have $E \vdash a_0 : B$, and $E, x{:}B \vdash b : A$. Since $a \rightarrow^p a'$, we must have have $a_0 \rightarrow^p a_0'$. By induction hypothesis, $E \vdash a_0' : B$, and hence $E \vdash \mathit{let}\ x{=}a_0'\ \mathit{in}\ b : A$, as required.
- Case $a = \mathit{if}\ u = v\ \mathit{then}\ a_0\ \mathit{else}\ a_1$: Note that either $a \rightarrow^p a_0$ or $a \rightarrow^p a_1$. In both cases, since $E \vdash \mathit{if}\ u = v\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A$, we have $E \vdash a_0 : A$ and $E \vdash a_1 : A$, as required.
- Case $a = (\mathit{new}\, c(v_1, \dots, v_n)).f_j$: We have $\mathit{fields}(c) = f_i \mapsto A_i{}^{i\in 1..n}$. The type derivation for a is as follows:

$$\cfrac{\cfrac{E \vdash v_i : A_i{}^{i\in 1..n}}{E \vdash \mathit{new}\ c(v_1, \dots, v_n) : c}}{E \vdash (\mathit{new}\ c(v_1, \dots, v_n)).f_j : A_j}$$

  Since $a' = v_j$, we have $E \vdash v_j : A_j$, as required.
- Case $a = (\mathit{new}\ c(v_1, \dots, v_n)).\ell_j(u_1, \dots, u_m)$: Let $v = \mathit{new}\ c(v_1, \dots, v_n)$. We have $\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)\ {}^{i\in 1..n}$, where $\mathit{sig}_j = B(A_1\ x_1, \dots, A_m\ x_m)$. By the typing derivation for $E \vdash a : B$, we have $E \vdash u_k : A_k$ for all $k \in 1..m$, and $E \vdash v : c$. By assumption on the execution environment, we know $\mathit{this}{:}c, x_1{:}A_1, \dots, x_m{:}A_m \vdash b : B$. Applying the Substitution and the Weakening Lemmas, we get $E \vdash b\{\mathit{this}{\leftarrow}v, x_k{\leftarrow}u_k{}^{k\in 1..m}\} : B$, as required.
- Case $a = w{:}\ell_j(u_1, \dots, u_n)$: We have $\mathit{class}(w) = c$, $\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)\ {}^{i\in 1..n}$ where $\mathit{sig}_j = B(A_1\ x_1, \dots, A_m\ x_m)$. By the typing derivation for $E \vdash a : B$, we have $E \vdash u_i : A_i$ for all $i \in 1..m$. We can therefore derive the required type for $a' = q[\mathit{new}\ c(p).\ell(u_1, \dots, u_m)]$:

$$\cfrac{\cfrac{E \vdash \mathit{new}\ c(p) : c \quad E \vdash u_i : A_i \quad \forall i \in 1..m}{E \vdash \mathit{new}\ c(p).\ell(u_1, \dots, u_m) : B}}{E \vdash q[\mathit{new}\ c(p).\ell(u_1, \dots, u_m)] : B}$$

- Case $a = q[v]$: Since $E \vdash q[v] : A$, we have $E \vdash v : A$, and $q[v] \rightarrow^p v$, as required.
- Case $a = q[a_0]$, where a_0 is not a value: Since $E \vdash q[a_0] : A$, we have $E \vdash a_0 : A$, and since $a \rightarrow^p a'$, we must have $a_0 \rightarrow^q a_0'$. By induction hypothesis, $E \vdash a_0' : A$, and hence $E \vdash q[a_0'] : A$, as required. $\square$

To state the Progress Theorem, we need to recognize programs that are blocked because of a *null* in object position. We say a method body a is *null-blocked* if, essentially, it is stuck trying to access a field of a null object, or invoke a method on a null object. Formally, a is null-blocked if it is of the form $\mathit{null}.f_j$, $\mathit{null}.\ell(u_1, \dots, u_n)$, $\mathit{let}\ x{=}a\ \mathit{in}\ b$ (where a is null-blocked), or $q[a]$ (where a is null-blocked).

Theorem 5 (Progress). If $\varnothing \vdash a : A$ and a is not a value and is not null-blocked, and $p \in \mathit{Prin}$, then $a \rightarrow^p a'$ for some a'.

Proof Again, we proceed by induction on the height of the typing derivation for $\varnothing \vdash a : A$. We assume a is not a value, and a is not null-blocked.

- Case $a = \mathit{let}\ x{=}a_0\ \mathit{in}\ b$: We consider two subcases, depending on whether a_0 is a value or not.
  - Case a_0 is a value v: We have $a \rightarrow^p b\{x{\leftarrow}v\}$.
  - Case a_0 is not a value: Since $\varnothing \vdash a : A$, we have $\varnothing \vdash a_0 : B$ for some B, a_0 not a value. Since a is not null-blocked, a_0 is not null-blocked. Hence, by induction hypothesis, we have $a_0 \rightarrow^p a_0'$. Hence, we have $a \rightarrow^p \mathit{let}\ x{=}a_0'\ \mathit{in}\ b$.
- Case $a = \mathit{if}\ u = v\ \mathit{then}\ a_0\ \mathit{else}\ a_1$: We have $a \rightarrow^p a_0$ or $a \rightarrow^p a_1$ depending on the result of $u = v$.
- Case $a = v.f_j$: Since $\varnothing \vdash a : A$ and a is not null-blocked, we must have $v = \mathit{new}\ c(u_1, \dots, u_n)$, and $\mathit{fields}(c) = f_i \mapsto A_i{}^{i\in 1..n}$. Therefore, we have $v.f_j \rightarrow^p u_j$.
- Case $a = v.\ell_j(u_1, \dots, u_m)$: Since $\varnothing \vdash a : A$ and a is not null-blocked, we must have $v = \mathit{new}\, c(u_1, \dots, u_n)$, $\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)$, and $\mathit{sig}_j = B(A_1\ x_1, \dots, A_m\ x_m)$. Therefore, we have $v.\ell_j(u_1, \dots, u_m) \rightarrow^p b_j\{\mathit{this}{\leftarrow}v, x_k{\leftarrow}u_k{}^{k\in 1..m}\}$.
- Case $a = w{:}\ell(u_1, \dots, u_m)$: The following transition rule $w{:}\ell(u_1, \dots, u_m) \rightarrow^p q[\mathit{new}\ c(p).\ell(u_1, \dots, u_m)]$ applies, with $\mathit{owner}(w) = q$ and $\mathit{class}(w) = c$.
- Case $a = q[a_0]$: We consider two subcases, depending on whether a_0 is a value or not.

- Case a_0 is a value v: We have $q[v] \rightarrow^p v$.
- Case a_0 is not a value: Since $\varnothing \vdash q[a_0] : A$, we have $\varnothing \vdash a_0 : A$, a_0 not a value. Since a is not null-blocked, a_0 is not null-blocked. Hence, by induction hypothesis, we have $a_0 \rightarrow^q a_0'$, and $q[a_0] \rightarrow^p q[a_0']$. □

We can now state soundness formally. We say a method body a is stuck if a is not a value, a is not null-blocked, and there is no a' and p such that $a \rightarrow^p a'$. We write $a \rightarrow^* a'$ to mean that there exists a sequence $a_1, \ldots, a_n$ and principals $p_1, \ldots, p_{n+1}$ such that $a \rightarrow^{p_1} a_1 \rightarrow^{p_2} \cdots \rightarrow^{p_n} a_n \rightarrow^{p_{n+1}} a'$. (Hence, $\rightarrow^*$ is a kind of transitive closure of $\rightarrow^p$.)

Theorem 6 (Soundness). If $\varnothing \vdash a : A$, and $a \rightarrow^* a'$, then a' is not stuck.

Proof. A straightforward induction on the number of transitions in $a \rightarrow^* a'$. □

C. 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 [21], from which some of the following explanations are drawn. Some constructs primitive here are actually derived forms in the original calculus.

Names, Messages:

$k ::= \mathsf{Encrypt} \mid \mathsf{Decrypt}$	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, $M_1, \ldots, M_n$.

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 $\{\lvert M \rvert\}_N$ 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
Un	data known to the opponent
$(x_1{:}T_1, \ldots, x_n{:}T_n)$	dependent record, $n \geq 0$
$\mathsf{Union}(t_1(T_1), \ldots, t_n(T_n))$	tagged union
Top	top
$\mathsf{SharedKey}(T)$	shared-key type
$\mathsf{KeyPair}(T)$	asymmetric key-pair
$k\ \mathsf{Key}(T)$	encryption or decryption part
$\ell\ \mathsf{Challenge}\ es$	challenge type
$\ell\ \mathsf{Response}\ fs$	response type
$e, f ::=$	atomic effect

$\mathsf{end}\ L$	end-event labelled L
$\mathsf{check}\ \ell\ N$	name-check for a nonce N
$\mathsf{trust}\ M{:}T$	trust that $M{:}T$
$\mathit{es}, \mathit{fs} ::=$	effect
$[e_1, \ldots, e_n]$	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 ℓ $\mathsf{Challenge}$ es and ℓ $\mathsf{Response}$ 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 ℓ $\mathsf{Challenge}$ es can be cast into a response of type ℓ $\mathsf{Response}$ 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 ℓ $\mathsf{Challenge}$ es, and check that it equals an incoming response of type ℓ $\mathsf{Response}$ 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 $\mathsf{Public\ Challenge}$ es is public, or tainted, just if $\mathit{es} = [\,]$. Type $\mathsf{Public\ Response}$ fs is always public, but tainted just if $\mathit{es} = [\,]$. Neither $\mathsf{Private\ Challenge}$ es nor $\mathsf{Private\ Response}$ 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
$\mathsf{out}\ M\ N$	output
$\mathsf{inp}\ M\ (x{:}T); P$	input (x bound in P)
$\mathsf{repeat\ inp}\ M\ (x{:}T); P$	replicated input (x bound in P)
$\mathsf{split}\ M\ \mathsf{is}\ (x_1{:}T_1, \ldots, x_n{:}T_n); P$	record splitting
$\mathsf{match}\ M\ \mathsf{is}\ (N, y{:}T); P$	pair matching (y bound in P)
$\mathsf{case}\ M\ \mathsf{is}\ t_i(x_i{:}T_i); P_i\ ^{i \in 1..n}$	tagged union case (t_i distinct)
$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional (new)
$\mathsf{new}\ (x{:}T); P$	name generation (x bound in P)
$P \mid Q$	composition
stop	inactivity

$\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N;P$	symmetric decrypt (x bound in P)
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{\lvert x{:}T\rvert\}_{N^{-1}};P$	asymmetric decrypt (x bound in P)
$\mathsf{check}\ M\ \mathsf{is}\ N;P$	nonce-checking
$\mathsf{begin}\ L;P$	begin-assertion
$\mathsf{end}\ L;P$	end-assertion
$\mathsf{cast}\ M\ \mathsf{is}\ (x{:}T);P$	cast to nonce type
$\mathsf{witness}\ M{:}T;P$	witness testimony
$\mathsf{trust}\ M\ \mathsf{is}\ (x{:}T);P$	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 $\mathsf{match}\ M\ \mathsf{is}\ (N,y{:}U);P$ 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 $\mathsf{case}\ M\ \mathsf{is}\ t_i(x_i{:}T_i);P_i\ {}^{i\in 1..n}$ 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 $\mathsf{if}\ M=N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as P if M and N are the same message, and otherwise as Q. (This process is not present in the original calculus [21] but is a trivial and useful addition.)

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

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

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

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

The process $\mathsf{cast}\ M\ \mathsf{is}\ (x{:}T);P$ 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 $\ell\ \mathsf{Challenge}\ es$, and T is a response type $\ell\ \mathsf{Challenge}\ fs$. This is the only way to populate a response type.

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

The process $\mathsf{trust}\ M\ \mathsf{is}\ (x{:}T);P$ 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 $\mathsf{witness}\ M{:}T;Q$ 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 $\mathsf{begin}\ L$ event for every $\mathsf{end}\ L$ 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
$E \vdash es$	good effect es
$E \vdash T$	good type T
$E \vdash M : T$	good message M of type T
$E \vdash P : es$	good process P with effect es

We omit the rules defining these judgments, which can be found in [21]; our previous informal explanation of types should give some intuitions.

We made two additions to the language as defined in [21], namely the empty record type () (and corresponding empty record message ()), and the conditional form $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$. 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 [21]:

Extensions to Spi for the Conditional:

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

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

The type and effect system can guarantee the robust safety of a process, according to the following theorem [21]:

Theorem 7 (Robust Safety). If $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : [\,]$ then P is robustly safe.

D. Proofs

D.1. Proof of Theorem 1

A consequence of the types translation for our calculus is that $[\![A]\!]$ is isomorphic to Un for all types A. Formally,

Lemma 3. $[\![A]\!] <:> \mathsf{Un} \vdash$ for all types A.

In practice, this means that we can replace $[\![A]\!]$ by Un in type derivations, and vice versa.

Some general remarks on typing are in order. A consequence of Lemma 3, as well as our general use of types, reveals that we rely on typing exclusively to show security properties, not to establish standard safety results. For instance, we do not use types to ensure that the type of the arguments supplied at method invocation match the type of the parameters to the method. Indeed, the only channel type in our translation has itself type Un.

In order to prove Theorem 1, we first establish some lemmas.

Lemma 4.

(1) If $E \vdash v : A$ then $E_{prin}, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \llbracket A \rrbracket$.

(2) If $E \vdash a : A$ and $E_0, \llbracket E \rrbracket \vdash p : \mathsf{Prin}$ and $k \notin dom(E_0, \llbracket E \rrbracket)$ then:

$$E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash \llbracket a \rrbracket^p_k : [\,]$$

(3) If $c \in \mathit{Class}$ and $\ell \in dom(methods(c))$ then $E_0 \vdash I_{class}(c, \ell) : [\,]$.

(4) If $w \in \mathit{WebService}$ then $E_0 \vdash I_{ws}(w) : [\,]$.

Proof. (1) We prove this by induction on the height of the type derivation for $E \vdash v : A$:

- Case $v = x$: Since $E \vdash x : A$, we must have $x{:}A \in E$. By definition of the translation for environment, $x{:}\llbracket A \rrbracket \in \llbracket E \rrbracket$, hence $E_{prin}, \llbracket E \rrbracket \vdash x : \llbracket A \rrbracket$, as required.
- Case $v = null$: We have $E \vdash null : c$. Since $\llbracket c \rrbracket = \mathsf{Union}(null(), c(\mathsf{Un}))$ and $\llbracket null \rrbracket = null()$, we have $E_{prin}, \llbracket E \rrbracket \vdash null() : \mathsf{Union}(null(\mathsf{Un}), c(\mathsf{Un}))$, as required.
- Case $v = new\ c(v_1, \dots, v_n)$: Since $E \vdash v : A$, where $A = c$, we have $fields(c) = f_i \mapsto A_i{}^{i \in 1..n}$, and $E \vdash v_i : A_i$ for all $i \in 1..n$. Let $E' = E_{prin}, \llbracket E \rrbracket$. By induction hypothesis, $E' \vdash \llbracket v_i \rrbracket : \llbracket A_i \rrbracket$ for all $i \in 1..n$. We can now derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{E' \vdash \llbracket v_i \rrbracket : \llbracket A_i \rrbracket \quad \forall i \in 1..n}{E' \vdash (\llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket) : (\llbracket A_1 \rrbracket, \dots, \llbracket A_n \rrbracket)}}{E' \vdash \llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket : (\mathsf{Un}, \dots, \mathsf{Un})}}{E' \vdash \llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket : \mathsf{Un}}}{E' \vdash c(\llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket) : \mathsf{Union}(null(\mathsf{Un}), c(\mathsf{Un}))}$$

- Case $v = p$: Since $E \vdash p : A$ (with $A = \mathit{Id}$), we have $p \in \mathit{Prin}$, hence $p{:}\mathsf{Prin} \in E_{prin}$. Since $\llbracket \mathit{Id} \rrbracket = \mathsf{Prin}$, we have $E_{prin}, \llbracket E \rrbracket \vdash p : \mathsf{Prin}$, as required.

(2) Again, we proceed by induction on the height of the type derivation for $E \vdash a : A$.

- Case $a = v$: We can apply the result of part (1). Since $E \vdash v : A$, then $E_{prin}, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \llbracket A \rrbracket$. We can derive:

$$\dfrac{E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash k : \mathsf{Un} \qquad \dfrac{E_0, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \llbracket A \rrbracket}{E_0, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \mathsf{Un}}}{E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash \mathsf{out}\ k\ \llbracket v \rrbracket : [\,]}$$

- Case $a = let\ x{=}a_0\ in\ b$: We have $E \vdash a_0 : B$ for some B, and $E, x{:}B \vdash b : A$. Applying the induction hypothesis, we derive $E_0, \llbracket E \rrbracket, k'{:}\mathsf{Un} \vdash \llbracket a_0 \rrbracket^p_{k'} : [\,]$ and $E_0, \llbracket E \rrbracket, x{:}\llbracket B \rrbracket, k{:}\mathsf{Un} \vdash \llbracket b \rrbracket^p_k : [\,]$. Let $E' = E_0, \llbracket E \rrbracket, k{:}\mathsf{Un}$. We can now derive:

$$\dfrac{\dfrac{E', k'{:}\mathsf{Un} \vdash \llbracket a \rrbracket^p_{k'} : [\,] \qquad \dfrac{E', k'{:}\mathsf{Un} \vdash k' : \mathsf{Un} \qquad \dfrac{E', k'{:}\mathsf{Un}, x{:}\llbracket B \rrbracket \vdash \llbracket b \rrbracket^p_k : [\,]}{E', k'{:}\mathsf{Un}, x{:}\mathsf{Un} \vdash \llbracket b \rrbracket^p_k : [\,]}}{E', k'{:}\mathsf{Un} \vdash \mathsf{inp}\ k'\ (x{:}\mathsf{Un}); \llbracket b \rrbracket^p_k : [\,]}}{E', k'{:}\mathsf{Un} \vdash \llbracket a \rrbracket^p_{k'} \mid \mathsf{inp}\ k'\ (x{:}\mathsf{Un}); \llbracket b \rrbracket^p_k : [\,]}}{E' \vdash \mathsf{new}\ (k'{:}\mathsf{Un}); (\llbracket a \rrbracket^p_{k'} \mid \mathsf{inp}\ k'\ (x{:}\mathsf{Un}); \llbracket b \rrbracket^p_k) : [\,]}$$

- Case $a = if\ u = v\ then\ a_0\ else\ a_1$: We have $E \vdash u : B$, $E \vdash v : B$, $E \vdash a_0 : A$, and $E \vdash a_1 : A$. Applying the induction hypothesis, we derive $E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash \llbracket a_0 \rrbracket^p_k : [\,]$ and $E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash \llbracket a_0 \rrbracket^p_k : [\,]$. By (1), we also have $E_0, \llbracket E \rrbracket \vdash \llbracket u \rrbracket : \llbracket B \rrbracket$ and $E_0, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \llbracket B \rrbracket$. This gives us $E_0, \llbracket E \rrbracket, k{:}\mathsf{Un} \vdash \mathsf{if}\ \llbracket u \rrbracket = \llbracket v \rrbracket\ \mathsf{then}\ \llbracket a_0 \rrbracket^p_k\ \mathsf{else}\ \llbracket a_1 \rrbracket^p_k : [\,]$, as required.
- Case $a = v.f_j$: We have $E \vdash v.f_j : A_j$, where $E \vdash v : c$ and $fields(c) = f_i \mapsto A_i{}^{i \in 1..n}$. By (1), $E_0, \llbracket E \rrbracket \vdash \llbracket v \rrbracket : \llbracket c \rrbracket$. Let $E' = E_0, \llbracket E \rrbracket, k{:}\mathsf{Un}$. First, let us derive that $E', y : \mathsf{Un} \vdash \mathsf{split}\ y\ \mathsf{is}\ (x_1{:}\llbracket A_1 \rrbracket, \dots, x_n{:}\llbracket A_n \rrbracket);$

$\mathsf{out}\ k\ x_j : [\,]$. Let $E'' = x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]$. (We trim environments where possible to reduce clutter.)

$$\dfrac{E', y : \mathsf{Un} \vdash y : \mathsf{Un} \qquad \dfrac{E' \vdash k : \mathsf{Un} \qquad \dfrac{E', y : \mathsf{Un}, E'' \vdash x_j : [\![A_j]\!]}{E', y : \mathsf{Un}, E'' \vdash x_j : \mathsf{Un}}}{E', y : \mathsf{Un}, E'' \vdash \mathsf{out}\ k\ x_j : [\,]}}{E', y : \mathsf{Un} \vdash \mathsf{split}\ y\ \mathsf{is}\ (x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]); \mathsf{out}\ k\ x_j : [\,]}$$

We can now derive:

$$\dfrac{\begin{array}{l} E' \vdash [\![v]\!] : \mathsf{Union}(null(\mathsf{Un}), c(\mathsf{Un})) \\ E', y : \mathsf{Un} \vdash \mathsf{stop} : [\,] \\ E', y : \mathsf{Un} \vdash \mathsf{split}\ y\ \mathsf{is}\ (x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]); \mathsf{out}\ k\ x_j : [\,] \end{array}}{\begin{array}{l} E' \vdash \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ null(y{:}\mathsf{Un}); \mathsf{stop} \\ \qquad\qquad \mathsf{is}\ c(y); \mathsf{split}\ y\ \mathsf{is}\ (x_1{:}[\![A_1]\!], \ldots, x_n{:}[\![A_n]\!]); \mathsf{out}\ k\ x_j : [\,] \end{array}}$$

- Case $a = v.\ell_j(u_1, \ldots, u_m)$: We have $E \vdash v.\ell_j(u_1, \ldots, u_m) : B$, where $E \vdash v : c$, $methods(c) = \ell_i \mapsto (sig_i, b_i)\ ^{i \in 1..n}$, $sig_j = B(A_1\ x_1,\ \ldots,\ A_m\ x_m)$, and $E \vdash u_k : A_k$ for all $k \in 1..m$. By (1), $E_0, [\![E]\!] \vdash [\![u_k]\!] : [\![A_k]\!]$ for all $k \in 1..m$. Let $E' = E_0, [\![E]\!], k{:}\mathsf{Un}$. First, let us derive that $E', y{:}\mathsf{Un} \vdash \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k) : [\,]$.

$$\dfrac{E', y{:}\mathsf{Un} \vdash c\_\ell : \mathsf{Un} \qquad E', y{:}\mathsf{Un} \vdash (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k) : \mathsf{Un}}{E', y{:}\mathsf{Un} \vdash \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k) : [\,]}$$

We can derive:

$$\dfrac{\begin{array}{l} E' \vdash [\![v]\!] : \mathsf{Union}(null(\mathsf{Un}), c(\mathsf{Un})) \\ E', y{:}\mathsf{Un} \vdash \mathsf{stop} : [\,] \\ E', y{:}\mathsf{Un} \vdash \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k) : [\,] \end{array}}{\begin{array}{l} E' \vdash \mathsf{case}\ [\![v]\!]\ \mathsf{is}\ null(y{:}\mathsf{Un}); \mathsf{stop} \\ \qquad\qquad \mathsf{is}\ c(y); \mathsf{out}\ c\_\ell\ (p, [\![v]\!], [\![u_1]\!], \ldots, [\![u_n]\!], k) : [\,] \end{array}}$$

- Case $a = w{:}\ell_j(u_1, \ldots, u_m)$: We have $E \vdash w{:}\ell_j(u_1, \ldots, u_m) : B$, where $class(w) = c$, $owner(w) = q$, $methods(c) = \ell_i \mapsto (sig_i, b_i)\ ^{i \in 1..n}$, $sig_j = B(A_1\ x_1, \ldots, A_m\ x_m)$, and $E \vdash u_k : A_k$ for all $k \in 1..m$. By (1), $E_0, [\![E]\!] \vdash [\![u_k]\!] : [\![A_k]\!]$ for all $k \in 1..m$. Rather than giving the full type derivation for the translation of a web service call, we outline the derivation of effects:

```
new $(k_1{:}\mathsf{Un}, k_2{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Challenge}\ [\,])$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
begin $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$;
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
out $w\ (req(getnonce()), k_1)$;
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
inp $k_1\ (res(getnonce(n_q{:}\mathsf{Un})))$;
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
cast $n_q$ is $(n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)])$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
out $w\ (p, \{req(w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t, n'_q)\}_{K_{pq}}, n_p, k_2)$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
inp $k_2\ (q'{:}\mathsf{Un}, bdy{:}\mathsf{Un})$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
decrypt $bdy$ is $\{res(plain)\}_{K_{pq}}$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
match $plain$ is $(w, rest$:
  $(r{:}Res(w), t'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t')]))$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
split $rest$ is $(r{:}Res(w), rest'$:
  $(t'{:}\mathsf{Un}, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t')]))$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
match $rest'$ is $(t, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t)])$;
// Effect: $[\mathsf{check\ Public}\ n_p]$
check $n_p$ is $n'_p$;
// Effect: $[\mathsf{end}\ res(p, q, w, r, t)]$
end $res(p, q, w, r, t)$;
// Effect: $[\,]$
case $r$ is $\ell(x)$; out $k\ x$
// Effect: $[\,]$
```

(3) Recall that we assume that method bodies are well-typed, that is, we assume for c, ℓ_j with $methods(c) = \ell_i \mapsto (sig_i, b_i)$ and $sig_j = B(A_1\ a_1, \ldots, A_m\ x_m)$, that $this{:}c, x_1{:}A_1, \ldots, x_m{:}A_m \vdash b_j : B$. By clause (2) above, this means that $E_0, this{:}[\![c]\!], x_1{:}[\![A_1]\!], \ldots, x_m{:}[\![A_m]\!], k{:}\mathsf{Un} \vdash [\![b_j]\!]^p_k : [\,]$. Applying Lemma 3, we derive $E_0, this{:}\mathsf{Un}, x_1{:}\mathsf{Un}, \ldots, x_m{:}\mathsf{Un}, k{:}\mathsf{Un} \vdash [\![b_j]\!]^p_k : [\,]$. We can now easily derive the following:

$$\dfrac{\dfrac{E_0 \vdash c\_\ell : \mathsf{Un} \qquad \dfrac{\begin{array}{c} E_0, z{:}\mathsf{Un} \vdash z : (\mathsf{Prin}, \mathsf{Un}, \ldots, \mathsf{Un}) \\ E_0, z{:}\mathsf{Un}, p{:}\mathsf{Prin}, this{:}\mathsf{Un}, x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un}, k{:}\mathsf{Un} \vdash [\![b_j]\!]^p_k : [\,] \end{array}}{E_0, z{:}\mathsf{Un} \vdash \mathsf{split}\ z\ \mathsf{is}\ (p{:}\mathsf{Prin}, this{:}\mathsf{Un}, x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un}, k{:}\mathsf{Un}); [\![b_j]\!]^p_k : [\,]}}{E_0 \vdash \mathsf{repeat\ inp}\ c\_\ell\ (z); \mathsf{split}\ z\ \mathsf{is}\ (p{:}\mathsf{Prin}, this{:}\mathsf{Un}, x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un}, k{:}\mathsf{Un}); [\![b_j]\!]^p_k : [\,]}}{E_0 \vdash I_{class}(c, \ell) : [\,]}$$

(4) Let $w \in WebService$, with $owner(w) = q$. First, note that the following derivation is admissible:

$$\dfrac{E_0, E \vdash p : Prin \quad E_0, E \vdash a : Req(w) \quad E_0, E, r{:}Res(w) \vdash P : es}{E_0, E \vdash \mathsf{let}\ r{:}Res(w){=}\mathsf{call}_w(p, a); P : es}$$

(The proof is a straightforward, if longish, type derivation.) Rather than giving the full type derivation for the implementation of web service w, we outline the derivation of effects:

```
repeat inp $w$ ($bdy$:Un, $k_1$:Un);
// Effect: [ ]
case $bdy$ is $req(getnonce())$;
// Effect: [ ]
new ($n_q$:Public Challenge [ ]);
// Effect: [check Public $n_q$]
out $k_1$ ($res(getnonce(n_q))$);
// Effect: [check Public $n_q$]
inp $w$ ($p'$:Un, $cipher$:Un, $n_p$:Un, $k_2$:Un);
// Effect: [check Public $n_q$]
$\prod_{p \in Prin}$ if $p = p'$ then
// Effect: [check Public $n_q$]
decrypt $cipher$ is $\{req(plain)\}_{K_{pq}}$;
// Effect: [check Public $n_q$]
match $plain$ is ($w$, $rest$:($a$:$Req(w)$, $t$:Un,
                          Public Response [end $req(p,q,w,a,t)$]));
// Effect: [check Public $n_q$]
split $rest$ is ($a$:$Req(w)$, $t$:Un, $n'_q$:Public Response [end $req(p,q,w,a,t)$]);
// Effect: [check Public $n_q$]
check $n_q$ is $n'_q$;
// Effect: [end $req(p,q,w,a,t)$]
end $req(p,q,w,a,t)$;
// Effect: [ ]
let $r$:$Res(w)$=call$_w(p,a)$;
// Effect: [ ]
begin $res(p,q,w,r,t)$;
// Effect: [end $res(p,q,w,r,t)$]
cast $n_p$ is ($n'_p$:Public Response [end $res(p,q,w,r,t)$]);
// Effect: [ ]
out $k_2$ ($q$, $\{res(w,r,t,n'_p)\}_{K_{pq}}$)
// Effect: [ ]
```

□

Lemma 5. If $\varnothing \vdash a : A$ and $p \in Prin$ and $k \notin dom(E_0)$ then:

$$E_{ws}, E_{prin} \vdash \mathsf{new}\ (E_{class}, E_{keys}); (I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k) : [\,]$$

Proof. This is a corollary of Lemma 4. Specifically, we can derive:

$$\dfrac{\dfrac{\dfrac{E_0 \vdash I_{class}(c,\ell) : [\,]\ ^{(c,\ell)\in ClMeth}}{E_0 \vdash I_{class} : [\,]} \quad \dfrac{E_0 \vdash I_{ws}(w) : [\,]\ ^{w \in WebService}}{E_0 \vdash I_{ws} : [\,]} \quad \dfrac{E_0, k{:}\mathsf{Un} \vdash [\![a]\!]^p_k : [\,]}{E_0 \vdash \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k : [\,]}}{E_0 \vdash I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k : [\,]}}{E_{ws}, E_{prin} \vdash \mathsf{new}\ (E_{class}, E_{keys}); (I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k) : [\,]}$$

□

We can now prove Theorem 1.

Theorem. If $\varnothing \vdash a : A$ and $p \in Prin$ and $k \notin dom(E_0)$ then the system

$$\mathsf{new}\ (E_{class}, E_{keys}); (I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k)$$

is robustly safe.

Proof. By Lemma 5,

$$E_{ws}, E_{prin} \vdash \mathsf{new}\ (E_{class}, E_{keys}); (I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k) : [\,].$$

Robust safety of the system follows by Theorem 7. □

D.2. Proof of Theorem 2

Theorem. If $\varnothing \vdash a : A$ and $p \in \mathit{Prin}$ and $k \notin \mathit{dom}(E_0)$ then the system

$$\mathsf{new}\ (E_{class}, E_{keys}); (I_{net} \mid I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k)$$

is robustly safe.

Proof. Rather than giving a full proof, we point out the parts of the proof of Theorem 1 that need to be updated. Essentially, we need to show that the new semantics for web method invocations is effect-free, and similarly for the new implementation of web services. These occur in the proof of Lemma 4, part (2) and (4).

As we did in Lemma 4, rather than giving the full type derivation for the translation of a web service call, we outline the derivation of effects:

```
$\mathsf{new}\ (k_1{:}\mathsf{Un}, k_2{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Challenge}\ [\,]);$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{begin}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t);$
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
$\mathsf{out}\ w\ (\mathit{CertVKp}, n_p, \mathit{req}(\mathit{getnonce}()), k_1);$
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
$\mathsf{inp}\ k_1\ (c{:}\mathsf{Un}, \mathit{res}(\mathit{getnonce}(n_q{:}\mathsf{Un})));$
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
$\mathsf{decrypt}\ c\ \mathsf{is}\ \{\!|\mathit{cert}{:}(q'{:}\mathsf{Un}, \mathsf{Decrypt\ Key}(\mathsf{AuthMsg}(q')))|\!\}_{VKCA^{-1}};$
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
$\mathsf{match}\ \mathit{cert}\ \mathsf{is}\ (q, \mathit{vkq}{:}\mathsf{Decrypt\ Key}(\mathsf{AuthMsg}(q)));$
// Effect: $[\mathsf{check\ Public}\ n_p, \mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]$
$\mathsf{cast}\ n_q\ \mathsf{is}\ (n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{req}(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)]);$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{out}\ w\ (p, \{\!|\mathit{req}(w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t, q, n'_q)|\!\}_{SKp}, k_2);$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{inp}\ k_2\ (q''{:}\mathsf{Un}, \mathit{bdy}{:}\mathsf{Un});$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{decrypt}\ \mathit{bdy}\ \mathsf{is}\ \{\!|\mathit{res}(\mathit{plain}{:}(w'{:}\mathsf{Un}, r{:}\mathsf{Un}, t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                              $\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p', q, w', r, t')]))|\!\}_{vkq^{-1}};$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{match}\ \mathit{plain}\ \mathsf{is}\ (w, \mathit{rest}{:}(r{:}\mathit{Res}(w), t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                           $\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p', q, w, r, t')]));$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{split}\ \mathit{rest}\ \mathsf{is}\ (r{:}\mathit{Res}(w), \mathit{rest}'{:}(t'{:}\mathsf{Un}, p'{:}\mathsf{Un},$
                              $\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p', q, w, r, t')]));$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{match}\ \mathit{rest}'\ \mathsf{is}\ (t, \mathit{rest}''{:}(p'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p', q, w, r, t)]));$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{match}\ \mathit{rest}''\ \mathsf{is}\ (p, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ \mathit{res}(p, q, w, r, t)]);$
// Effect: $[\mathsf{check\ Public}\ n_p]$
$\mathsf{check}\ n_p\ \mathsf{is}\ n'_p;$
// Effect: $[\mathsf{end}\ \mathit{res}(p, q, w, r, t)]$
$\mathsf{end}\ \mathit{res}(p, q, w, r, t);$
// Effect: $[\,]$
$\mathsf{case}\ r\ \mathsf{is}\ \ell(x); \mathsf{out}\ k\ x$
// Effect: $[\,]$
```

For the new implementation of web service w, rather than giving the full type derivation, we outline the derivation of effects:

```
repeat inp $w$ ($c$:Un, $n_p$:Un, $bdy$:Un, $k_1$:Un);
// Effect: [ ]
case $bdy$ is $req(getnonce())$;
// Effect: [ ]
decrypt $c$ is $\{\!|p$:Un, $vkp$:Decrypt Key(AuthMsg($p$))$|\!\}_{VKCA^{-1}}$;
// Effect: [ ]
new ($n_q$:Public Challenge [ ]);
// Effect: [check Public $n_q$]
out $k_1$ ($CertVKq$, $res(getnonce(n_q))$);
// Effect: [check Public $n_q$]
inp $w$ ($p'$:Un, $cipher$:Un, $k_2$:Un);
// Effect: [check Public $n_q$]
if $p = p'$ then
// Effect: [check Public $n_q$]
decrypt $cipher$ is $\{\!|req(plain$:($w$:Un, $a$:Un, $t$:Un, $q'$:Un,
                                Public Response [end $req(p, q', w, a, t)$]))$|\!\}_{vkp^{-1}}$;
// Effect: [check Public $n_q$]
match $plain$ is ($w$, $rest$:($a$:$Req(w)$, $t$:Un, $q'$:Un,
                          Public Response [end $req(p, q', w, a, t)$]));
// Effect: [check Public $n_q$]
split $rest$ is ($a$:$Req(w)$,
                 $t$:Un, $rest'$:($q'$:Un, Public Response [end $req(p, q', w, a, t)$]));
// Effect: [check Public $n_q$]
match $rest'$ is ($q$, $n'_q$:Public Response [end $req(p, q, w, a, t)$]);
// Effect: [check Public $n_q$]
check $n_q$ is $n'_q$;
// Effect: [end $req(p, q, w, a, t)$]
end $req(p, q, w, a, t)$;
// Effect: [ ]
let $r$:$Res(w)$=call$_w(p, a)$;
// Effect: [ ]
begin $res(p, q, w, r, t)$;
// Effect: [end $res(p, q, w, r, t)$]
cast $n_p$ is ($n'_p$:Public Response [end $res(p, q, w, r, t)$]);
// Effect: [ ]
out $k_2$ ($q$, $\{\!|res(w, r, t, p, n'_p)|\!\}_{SKq}$)
// Effect: [ ]
```

□

D.3. Proof of Theorem 3

Theorem. If $\varnothing \vdash a : A$ and $p \in Prin$ and $k \notin dom(E_0)$ then the system

$$\mathsf{new}\ (E_{class}, E_{keys}); (I_{net} \mid I_{class} \mid I_{ws} \mid \mathsf{new}\ (k{:}\mathsf{Un}); [\![a]\!]^p_k)$$

is robustly safe.

Proof. Rather than giving a full proof, we point out the parts of the proof of Theorem 1 that need to be updated. Essentially, we need to show that the new semantics for web method invocations is effect-free, and similarly for the new implementation of web services. These occur in the proof of Lemma 4, part (2) and (4).

As we did in Lemma 4, rather than giving the full type derivation for the translation of a web service call, we outline the derivation of effects:

```
new $(k_1{:}\mathsf{Un}, k_2{:}\mathsf{Un}, t{:}\mathsf{Un}, n_p{:}\mathsf{Public\ Challenge}\ [\,])$;
// Effect: [check Public $n_p$]
begin $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
out $w$ $(CertEKp, req(getnonce()), k_1)$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]inp $k_1$ $(c{:}\mathsf{Un}, cipher{:}\mathsf{Un}, res(getnonce(n_q{:}\mathsf{Un})))$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
decrypt $c$ is $\{\![cert{:}(q'{:}\mathsf{Un}, \mathsf{Encrypt\ Key}(\mathsf{AuthEncMsg}(q')))]\!\}_{VKCA^{-1}}$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
match $cert$ is $(q, ekq{:}\mathsf{Encrypt\ Key}(\mathsf{AuthEncMsg}(q)))$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
decrypt $cipher$ is $\{\![msg_2(q'{:}\mathsf{Un}, n_K{:}\mathsf{Un})]\!\}_{DKp^{-1}}$;
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
if $q = q'$ then
// Effect: [check Public $n_p$, end $req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)$]
cast $n_q$ is $(n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t)])$;
// Effect: [check Public $n_p$]
new $(K{:}\mathsf{SKey}(p, q, w))$;
// Effect: [check Public $n_p$]
witness $K{:}\mathsf{SKey}(p, q, w)$;
// Effect: [check Public $n_p$, trust $K{:}\mathsf{SKey}(p, q, w)$]
cast $n_K$ is $(n'_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p, q, w)])$;
// Effect: [check Public $n_p$]
out $w$ $(\{\![msg_3(w, p, t, K, n'_K)]\!\}_{ekq}, n_p, \{req(w, \ell([\![u_1]\!], \ldots, [\![u_n]\!]), t, n'_q)\}_K, k_2)$;
// Effect: [check Public $n_p$]
inp $k_2$ $(bdy{:}\mathsf{Un})$;
// Effect: [check Public $n_p$]
decrypt $bdy$ is $\{res(plain{:}(r{:}Res(w), t'{:}\mathsf{Un},$
                                  $\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t')]))\}_K$;
// Effect: [check Public $n_p$]
match $plain$ is $(r{:}Res(w), rest{:}(t'{:}\mathsf{Un}, \mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t')]))$;
// Effect: [check Public $n_p$]
match $rest$ is $(t, n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t)])$;
// Effect: [check Public $n_p$]
check $n_p$ is $n'_p$;
// Effect: [end $res(p, q, w, r, t)$]
end $res(p, q, w, r, t)$;
// Effect: [ ]
case $r$ is $\ell(x)$; out $k$ $x$
// Effect: [ ]
```

For the new implementation of web service w, rather than giving the full type derivation, we outline the derivation of effects:

```
repeat inp $w$ $(c{:}\mathsf{Un}, bdy{:}\mathsf{Un}, k_1{:}\mathsf{Un})$;
// Effect: [ ]
case $bdy$ is $req(getnonce())$;
// Effect: [ ]
decrypt $c$ is $\{\![p{:}\mathsf{Un}, ekp{:}\mathsf{Encrypt\ Key}(\mathsf{AuthEncMsg}(p))]\!\}_{VKCA^{-1}}$;
// Effect: [ ]
new $(n_q{:}\mathsf{Public\ Challenge}\ [\,])$;
// Effect: [check Public $n_q$]
new $(n_K{:}\mathsf{Private\ Challenge}\ [\,])$;
// Effect: [check Public $n_q$, check Private $n_K$]
out $k_1$ $(CertEKq, \{\![msg_2(q, n_K)]\!\}_{ekp}, res(getnonce(n_q)))$;
// Effect: [check Public $n_q$, check Private $n_K$]
```

```
inp $w$ ($cipher_1$:Un, $n_p$:Un, $cipher_2$:Un, $k_2$:Un);
// Effect: [check Public $n_q$, check Private $n_K$]
decrypt $cipher_1$
    is $\{\!|msg_3(plain_1{:}(w{:}\mathsf{Un}, p'{:}\mathsf{Un}, K{:}\mathsf{Top},$
                    $\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p', q, w)]))|\!\}_{DKq^{-1}}$;
// Effect: [check Public $n_q$, check Private $n_K$]
match $plain_1$ is $(w, rest{:}(p'{:}\mathsf{Un}, K{:}\mathsf{Top},$
                    $\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p', q, w)]))$;
// Effect: [check Public $n_q$, check Private $n_K$]
match $rest$ is $(p, rest'{:}(K{:}\mathsf{Top}, n'_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p, q, w)]))$;
// Effect: [check Public $n_q$, check Private $n_K$]
split $rest'$ is $(K{:}\mathsf{Top}, n'_K{:}\mathsf{Private\ Response}\ [\mathsf{trust}\ K{:}\mathsf{SKey}(p, q, w)])$;
// Effect: [check Public $n_q$, check Private $n_K$]
check $n_K$ is $n'_K$;
// Effect: [check Public $n_q$, trust $K$:SKey$(p, q, w)$]
trust $K$ is $(K'{:}\mathsf{SKey}(p, q, w))$;
// Effect: [check Public $n_q$]
decrypt $cipher_2$ is $\{req(plain_2{:}(a{:}Req(w), t{:}\mathsf{Un},$
                    $\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, a, t)]))\}_{K'}$;
// Effect: [check Public $n_q$]
split $plain_2$ is $(a{:}Req(w), t{:}\mathsf{Un}, n'_q{:}\mathsf{Public\ Response}\ [\mathsf{end}\ req(p, q, w, a, t)])$;
// Effect: [check Public $n_q$]
check $n_q$ is $n'_q$;
// Effect: [end $req(p, q, w, a, t)$]
end $req(p, q, w, a, t)$;
// Effect: [ ]
let $r{:}Res(w) = \mathsf{call}_w(p, a)$;
// Effect: [ ]
begin $res(p, q, w, r, t)$;
// Effect: [end $res(p, q, w, r, t)$]
cast $n_p$ is $(n'_p{:}\mathsf{Public\ Response}\ [\mathsf{end}\ res(p, q, w, r, t)])$;
// Effect: [ ]
out $k_2$ $\{res(r, t, n'_p)\}_{K'}$
// Effect: [ ]
```

□

E. First-Class Web Services

The model of web services captured by our calculus in Section 3 does not consider web services to be values. This reflects the fact that current WSDL does not allow for web services to be passed as requests or results. On the other hand, a web service has a simple representation as a string, namely the URL used to access the web service, and this string *can* be passed as a request or a result. Hence, it is possible, in a sense, to pass web services as values given the current web services infrastructure. In this section, we explore an extension of our object calculus that allows web services as first-class values. The main point here is to show that there is no real difficulty in modelling this aspect of the web services infrastructure. Our main result is type safety. We expect it would be straightforward to translate this extended calculus into the spi-calculus, but we do not describe this in detail.

For the sake of keeping this section essentially self-contained, we give the full syntax and semantics of the extended object calculus.

E.1. Syntax

We assume finite sets *Prin*, *WebService*, *Class*, *Field*, *Meth* of principal, web service, class, field, and method names, respectively.

Classes, Fields, Methods, Principals, Web Services:

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

There are now three kinds of data type: *Id* is the type of principal identifiers, $c \in \mathit{Class}$ is the type of instances of class c, and $\mathit{WS}(c)$ is the type of web services with implementation class $c \in \mathit{Class}$. A method signature specifies the types of its arguments and result.

Types and Method Signatures:

$A, B \in \mathit{Type} ::=$	type
$\quad \mathit{Id}$	principal identifier
$\quad c$	object
$\quad \mathit{WS}(c)$	web service
$\mathit{sig} \in \mathit{Sig} ::= B(A_1\, x_1, \dots, A_n\, x_n)$	method signature (x_i distinct)

As in Section 3, an execution environment defines the services and code available in the distributed system.

Execution Environment: (*fields*, *methods*, *owner*, *class*)

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

The owner and implementation class of a web service need not be globally known. We can assume that the representation of a web service w carries representations of its owner and its implementation class, which *class* and *owner* simply read off. Since we assume web services are given, and we do not provide for ways to actually create new web services, there is no loss of generality in taking this particular approach.

The syntax of method bodies and values is that of the original object calculus, with the differences that web services are values, and that we do not assume that web service invocations require a fixed web service.

Values and Method Bodies:

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

We again require a method body of the form $p[a]$, meaning p running body a, to keep track of which principal is running a method body in the upcoming operational semantics.

E.2. Operational Semantics

The operational semantics is defined by a transition relation, written $a \rightarrow^p a'$, where a and a' are method bodies, and p is the principal evaluating the body a.

Transitions:

(Red Let 1)
$$\frac{a \rightarrow^p a'}{\mathit{let}\ x{=}a\ \mathit{in}\ b \rightarrow^p \mathit{let}\ x{=}a'\ \mathit{in}\ b}$$

(Red Let 2)
$$\frac{}{\mathit{let}\ x{=}v\ \mathit{in}\ b \rightarrow^p b\{x{\leftarrow}v\}}$$

(Red If)
$$\frac{}{\mathit{if}\ u = v\ \mathit{then}\ a_{\mathit{true}}\ \mathit{else}\ a_{\mathit{false}} \rightarrow^p a_{u=v}}$$

(Red Field)
$$\frac{\mathit{fields}(c) = f_i \mapsto A_i\ {}^{i\in 1..n} \quad j \in 1..n}{(\mathit{new}\ c(v_1,\ldots,v_n)).f_j \rightarrow^p v_j}$$

(Red Invoke)(where $v = \mathit{new}\ c(v_1,\ldots,v_n)$)
$$\frac{\mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)\ {}^{i\in 1..n} \quad j \in 1..n \quad \mathit{sig}_j = B(A_1\,x_1,\ldots,A_m\,x_m)}{v.\ell_j(u_1,\ldots,u_m) \rightarrow^p b_j\{\mathit{this}{\leftarrow}v, x_k{\leftarrow}u_k\ {}^{k\in 1..m}\}}$$

(Red Remote)
$$\frac{\mathit{owner}(w) = q \quad \mathit{class}(w) = c}{w{:}\ell(u_1,\ldots,u_n) \rightarrow^p q[\mathit{new}\ c(p).\ell(u_1,\ldots,u_n)]}$$

(Red Prin 1)
$$\frac{a \rightarrow^q a'}{q[a] \rightarrow^p q[a']}$$

(Red Prin 2)
$$\frac{}{q[v] \rightarrow^p v}$$

E.3. Type System

The judgments of our type system all depend on an *environment* E, that defines the types of all variables in scope. An environment takes the form $x_1{:}A_1,\ldots,x_n{:}A_n$ and defines the type A_i for each variable x_i. The domain $\mathit{dom}(E)$ of an environment E is the set of variables whose types it defines.

Environments:

$D, E ::=$	environment
$\varnothing$	empty
$E, x{:}A$	entry
$\mathit{dom}(x_1{:}A_1,\ldots,x_n{:}A_n) \triangleq \{x_1,\ldots,x_n\}$	domain of an environment

The following are the two judgments of our type system. They are inductively defined by rules presented in the following tables.

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

$E \vdash \diamond$	good environment
$E \vdash a : A$	good expression a of type A

We write $E \vdash \mathcal{J}$ when we want to talk about both kinds of judgments, where $\mathcal{J}$ stands for either $\diamond$ or $a : A$.

The following rules define an environment $x_1{:}A_1,\ldots,x_n{:}A_n$ to be well-formed if each of the names $x_1,\ldots,x_n$ are distinct.

Rules for Environments:

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env x)(where $x \notin \mathit{dom}(E)$)
$$\frac{E \vdash \diamond}{E, x{:}A \vdash \diamond}$$

We present the rules for deriving the judgment $E \vdash a : A$ that assigns a type A to a value or method body a. These rules are split into two tables, one for values, and one for method bodies.

Rules for Typing Values:

(Val x)
$$\frac{E = E_1, x{:}A, E_2 \quad E \vdash \diamond}{E \vdash x : A}$$

(Val *null*)
$$\frac{E \vdash \diamond}{E \vdash \mathit{null} : c}$$

(Val WS)
$$\frac{E \vdash \diamond \quad \mathit{class}(w) = c}{E \vdash w : \mathit{WS}(c)}$$

(Val Object)
$$\frac{\mathit{fields}(c) = f_i \mapsto A_i{}^{i \in 1..n} \quad E \vdash v_i : A_i \quad \forall i \in 1..n}{E \vdash \mathit{new}\ c(v_1, \ldots, v_n) : c}$$

(Val Princ)
$$\frac{E \vdash \diamond}{E \vdash p : \mathit{Id}}$$

Rules for Typing Method Bodies:

(Body Let)
$$\frac{E \vdash a : A \quad E, x{:}A \vdash b : B}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : B}$$

(Body If)
$$\frac{E \vdash u : A \quad E \vdash v : A \quad E \vdash a : B \quad E \vdash b : B}{E \vdash \mathit{if}\ u = v\ \mathit{then}\ a\ \mathit{else}\ b : B}$$

(Body Field)
$$\frac{E \vdash v : c \quad \mathit{fields}(c) = f_i \mapsto A_i{}^{i \in 1..n} \quad j \in 1..n}{E \vdash v.f_j : A_j}$$

(Body Invoke)
$$\frac{\begin{array}{c} E \vdash v : c \quad \mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)^{\,i \in 1..n} \quad j \in 1..n \\ \mathit{sig}_j = B(A_1\ x_1, \ldots, A_m\ x_m) \quad E \vdash u_k : A_k \quad \forall k \in 1..m \end{array}}{E \vdash v.\ell_j(u_1, \ldots, u_m) : B}$$

(Body Remote)
$$\frac{\begin{array}{c} E \vdash v : \mathit{WS}(c) \\ \mathit{methods}(c) = \ell_i \mapsto (\mathit{sig}_i, b_i)^{\,i \in 1..n} \quad j \in 1..n \\ \mathit{sig}_j = B(A_1\ x_1, \ldots, A_m\ x_m) \quad E \vdash u_i : A_i \quad \forall i \in 1..m \end{array}}{E \vdash v{:}\ell_j(u_1, \ldots, u_m) : B}$$

(Body Princ)
$$\frac{E \vdash a : A}{E \vdash p[a] : A}$$

We make the following assumption on the execution environment.

Assumptions on the Execution Environment:

(1) For each $w \in \mathit{WebService}$, $\mathit{fields}(\mathit{class}(w)) = \mathit{CallerId} : \mathit{Id}$.
(2) No tagged expression $p[a]$ occurs within the body of any method;
such expressions occur only at runtime, to track the call stack of principals.
(3) for each $c \in \mathit{Class}$ and each $\ell \in \mathit{dom}(\mathit{methods}(c))$,
if $\mathit{methods}(c)(\ell) = (B(A_1\ x_1, \ldots, A_n\ x_n), b)$,
then $\mathit{this}{:}c, x_1{:}A_1, \ldots, x_n{:}A_n \vdash b : B$.

We can establish the soundness of the type system of this extended object calculus by essentially the same way we established the soudness of the type system of the original object calculus. Recall that a method body is null-blocked if it is of the form $\mathit{null}.f_j$, $\mathit{null}.\ell(u_1, \ldots, u_n)$, $\mathit{let}\ x{=}a\ \mathit{in}\ b$ (where a is null-blocked), or $q[a]$ (where a is null-blocked). A method body is stuck if a is not a value, a is not null-blocked, and there is no a' and p such that $a \rightarrow^{p} a'$. We write $a \rightarrow^{*} a'$ to mean that there exists a sequence $a_1, \ldots, a_n$ and principals $p_1, \ldots, p_{n+1}$ such that $a \rightarrow^{p_1} a_1 \rightarrow^{p_2} \cdots \rightarrow^{p_n} a_n \rightarrow^{p_{n+1}} a'$.

Theorem 8 (Soundness). If $\varnothing \vdash a : A$, and $a \rightarrow^{*} a'$, then a' is not stuck.

Proof. A straightforward adaptation of the proof of Theorem 6, via corresponding Preservation and Progress theorems. □

To illustrate the usefulness of first-class web services, consider the following simple example, where the fact that web services can be passed as arguments to methods is quite natural. Suppose, as we did in Section 3, that there are two principals $\mathit{Alice}, \mathit{Bob} \in \mathit{Prin}$, and a web service *cal = http://mycalendar.com/CalendarService*, where we have $class(cal) = CalendarServiceClass$. The web service *cal* maintains an appointment calendar for principals. It offers web methods to query a principal's calendar for a free time slot, and to reserve time slots. More precisely, the service has the following interface:

```
class CalendarServiceClass
  Id CallerId
  Bool Available(Id account, Time from, Time to)
    ⟨check if selected time slot if free for account⟩
  Void Reserve(Id account, Time from, Time to)
    ⟨reserve time slot for account⟩
```

(We assume that the classes *Bool*, *Time*, and *Void* are provided in the execution environment. The details of their implementation are irrelevant to our discussion.)

Suppose that Alice has an account on *cal*, and that she wants to make an appointment with a calendar-enabled banking service—that is, a banking service that offers a web method for scheduling appointments with a bank advisor via a calendar service. Consider a calendar-enabled version of the banking service of Section 3. Let $w = \mathit{http://bob.com/BankingService}$, where we have $owner(w) = Bob$ and $class(w) = BankingServiceClass$. We add a web method *MakeAppt* to *BankingServiceClass* that takes as argument a time period during which the appointment is sought, and a calendar service that the banking service can query to confirm that a common free time slot is available between the client and the bank advisor. The interface of the augmented banking service is as follows:

```
class BankingServiceClass
  Id CallerId
  Num Balance(Num account)
    if account = 12345 then
      if this.CallerId = Alice then 100 else null
     else ...
  Time MakeAppt(Time from, Time to, WS(CalendarService) cs)
    ... cs.Available(CallerId, ...) ...
```

Hence, if Alice wants to make an appointment sometime within the next week, she could issue the web method call *w*:*MakeAppt*(*18/11/02:08:00*, *23/11/02:17:00*, *cal*). (We assume appropriate syntax for constants of type *Time*.) During the evaluation of this web method invocation, the implementation of *MakeAppt* will make calls to *cal*:*Available* to find a time slot suitable to Alice, and finally a call to *cal*:*Reserve* to reserve a time slot. A principal with an account on a different calendar service c would call *w*:*MakeAppt* passing in c as the calendar service.

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

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[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 A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.

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