Verified Implementations of the Information Card Federated Identity-Management Protocol

Karthikeyan Bhargavan* Cédric Fournet* Andrew D. Gordon* Nikhil Swamy†

*Microsoft Research †University of Maryland, College Park

ABSTRACT

We describe reference implementations for selected configurations of the user authentication protocol defined by the *Information Card Profile V1.0*. Our code can interoperate with existing implementations of the roles of the protocol (client, identity provider, and relying party). We derive formal proofs of security properties for our code using an automated theorem prover. Hence, we obtain the most substantial examples of verified implementations of cryptographic protocols to date, and the first for any federated identity-management protocols. Moreover, we present a tool that downloads security policies from services and identity providers and compiles them to a verifiably secure client proxy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Verification

Keywords

Cryptographic protocol verification, Verified implementations, Web Services Security, Federated identity management, CardSpace.

1. INTRODUCTION

Verified Reference Implementations of Protocols. One of the successes of formal methods in security is the wide range of tools now available for proving properties of cryptographic protocols. Given a description in the formal style of Dolev and Yao [1983], verifiers such as ProVerif [Blanchet, 2002] and AVISPA [et al, 2005] can check nearly automatically various security properties of protocols against realistic threat models.

A significant barrier to the adoption of these tools is the effort of writing a formal description and maintaining its consistency with the informal specification. A promising solution is to build tools to verify the code of reference implementations of security protocols. A *reference implementation* is one optimised for clarity and ease of verification and testing, not performance. The idea is (1) to check security properties by extracting a formal description from implementation code and passing this description to an existing verifier, and (2) to check consistency between the reference implementation and the informal specification in the same way as for any implementation, via interoperability testing with other implementations.

This paper reports the techniques underpinning the most complex examples to date of such verified implementations.

ASIACCS '08, March 18-20, Tokyo, Japan

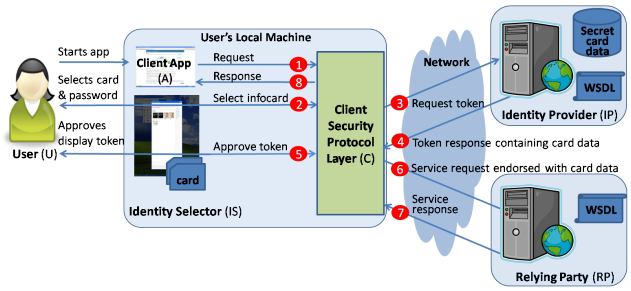

Figure 1: InfoCard: Card-based User Authentication

Information Card Profile V1.0 (InfoCard). We have built and verified reference implementations of various configurations of the card-based user authentication protocol defined by the *Information Card Profile V1.0* [InfoCard Guide, Nanda, 2006]. We refer to this protocol as InfoCard.

InfoCard is the core of a federated identity-management framework that allows users to collect *information cards* issued by various *identity providers* (IPs), such as web portals, governments, or companies, and to use the cards to sign into various *relying parties* (RPs), such as websites. An information card is a digital identity consisting of a set of attributes. Possible attributes include real names, pseudonyms, and email addresses. The client user interface for choosing and maintaining information cards is known as the *identity selector* (IS).

The InfoCard protocol is typically initiated by the web browser of a user that visits the website of a relying party [Jones, 2006]. The protocol allows the human user of the client computer to choose a suitable card via IS, to prove the attributes recorded in the card to RP, and to authenticate RP, via a message exchange with IP; Figure 1 depicts a typical run.

The protocol is *configurable* in that message formats are determined by configuration data such as policies set by RP and IP. In typical configurations, InfoCard amounts to a three-party authentication protocol between a client, an IP, and an RP, with various intended privacy guarantees. For example, RP should not learn more about the user identity than what is explicitly presented in the card.

There are multiple implementations of the various roles of the InfoCard protocol. Windows CardSpace is the Microsoft implementation of an identity selector for Microsoft Windows; CardSpace can be invoked from both Firefox (with a plugin) and Internet Ex-

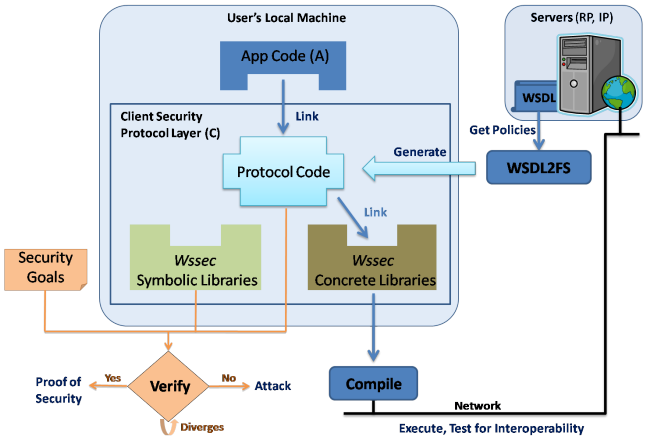

Figure 2: Verification Architecture

plorer. The OSIS [2006] working group exists to coordinate efforts to develop an open source identity selector. Microsoft makes available sample code for building identity provider and relying party servers, and maintains public servers for test purposes.

There are some indications of early usage. CardSpace is distributed with Windows Vista. Leading online retailers (including the second largest, the Otto group), and various banks have already adopted the protocols. The largest relying party to date is Windows Live; for example, Hotmail users can log in with InfoCard.

InfoCard is an important, worthwhile, and challenging example for verification. It is important because of the likelihood of large scale adoption. It is worthwhile because the protocols have yet to be standardised and are not yet widely used—so there is an opportunity for details to be corrected. It is challenging because InfoCard is highly configurable so that verification in full generality may be out of reach of automatic verifiers.

Our Approach. Figure 2 presents an overview of our verification architecture; it follows a framework first proposed by Bhargavan et al. [2006b]. At the core of our approach are two implementations of a set of libraries, called Wssec, that provide common cryptographic and networking functions used in web services security protocols. The *concrete* version implements real cryptographic operations and generates messages in a standardised XML format; the *symbolic* version implements an abstract model of cryptography, in the sense of Dolev and Yao [1983], and produces symbolic messages.

Our verified InfoCard implementations consist of protocol-specific code that uses functionality provided by the generic Wssec libraries. By linking application code with protocol code and the concrete Wssec libraries, we produce an executable that can run on the wire and exchange messages, and hence interoperate, with other implementations of the protocol. By linking with the symbolic libraries instead, we produce an executable model that can generate symbolic traces of the protocol, which are useful for debugging. More significantly, we can automatically verify that this symbolic model preserves the protocol security goals.

Our verifier is based on an existing tool chain consisting of the model extractor FS2PV [Bhargavan et al., 2006b], which compiles code written in F# (a dialect of ML) [Syme, 2005] to process models in an applied pi calculus [Abadi and Gordon, 1999, Abadi and Fournet, 2001], and the state-of-the-art verifier ProVerif, which analyses such models automatically. For many protocol implementations, the verifier either proves the security goals or produces a counter-example. In some cases, however, the analysis may not terminate; in others, it may take several gigabytes of memory. Hence, for verification of our examples to be feasible, we make various improvements. First, we implement additional inlining, dead-code elimination, partial evaluation transformations in the model extractor FS2PV, which result in smaller, more specialised process models to be passed to the verifier. For example, there are different but semantically equivalent representations of keys and nonces in ProVerif; we choose representations that lead to more efficient verification than in the original FS2PV. Second, we rewrite parts of the FS2PV libraries for web services security to optimise analysis with ProVerif. Third, to deal with the complexity of the layered web services security specifications, we write symbolic Wssec libraries that manipulate abstract messages and not XML documents.

The protocol code in Figure 2 can either be written by hand, or automatically generated based on metadata describing the protocol configuration. We formally define the set of possible InfoCard configurations as an F# datatype and implement a tool, called WSDL2FS, that downloads security policies from some given relying parties and identity providers, infers the InfoCard configuration, and generates protocol-specific F# code. The generated code can then be verified using our tools before being deployed.

Structure of the Paper. Section 2 informally presents information cards, the various protocol roles, the sequence of events making up the protocol, the configuration possibilities, and the security goals. Section 3 describes the web services security mechanisms upon which InfoCard depends. Section 4 presents our formal model for InfoCard configuration data, together with five example configurations. Section 5 describes message traces, shown in Figures 4 and 5, derived from two of these examples. Section 6 describes the construction of our implementations, including a technique for automatically generating protocol-specific code from policies. Section 7 presents verification results for our implementations. Using our newly improved version of FS2PV, we prove authentication and secrecy properties for our example configurations. Section 8 discusses related work and Section 9 concludes.

Contributions. To the best of our knowledge, the most substantial prior verification of security properties of protocol code is work by Bhargavan et al. [2006b,a], who verify trace properties of two-party web services protocols. The main technical contributions of the paper are as follows.

- Our verification of security properties of protocol code for four significant configurations of InfoCard is the most complex to date. Figures 4 and 5 indicate the numerous nested encryptions and signatures arising from the use of web service security. We verify trace properties of three party protocols; moreover, we verify privacy properties expressed as behavioural equivalences rather than trace properties.

- We introduce the Wssec abstraction for efficient verification of protocols layered on web services security. We need this abstraction, together with additional optimisations of the FS2PV model extractor and associated libraries, for our verifications to be feasible.

- We introduce a framework for policy-based generation of implementation code for InfoCard clients, and its formal verification. Verification on demand is a response to the difficulty of verifying the protocol for all possible configurations. Generation of proxy code from metadata is a standard idea, but we are not aware of any prior work to apply verification techniques to confirm the correct construction of proxies.

This is the first paper to present formal descriptions of message exchanges in the InfoCard specification, and to establish formal

authentication and privacy properties of parts of a complete implementation. Moreover, we show that, as with many configurable security protocols, some configurations have potential vulnerabilities. Nonetheless, this paper is not a full security analysis of the specifications or implementations of InfoCard. A full analysis would need to examine the deployed implementation code, the storage of protocol data, the usability of identity selectors, and other factors.

Our verification results rely on the soundness of our symbolic abstraction of the concrete Wssec libraries, as well as the correctness of both FS2PV and ProVerif. We validate the concrete Wssec libraries in three ways: by interoperability testing; by checking (by hand) that symbolic traces conform to concrete traces on a series of configurations; and by appealing to earlier, independent verification results for protocols built on top of Wssec [Bhargavan et al., 2006a]. The correctness of the core translations and algorithms underpinning FS2PV and ProVerif are addressed elsewhere [Bhargavan et al., 2006b, Blanchet, 2002, Blanchet et al., 2005]. The additional optimisations for this work amount to standard, semantics-preserving program transformations; we omit their formalisation.

2. THE INFOCARD PROTOCOL

2.1 Cards as Fragments of Identities

An *information card* represents an identity for the user who holds the card; it is a container for a collection of *claims* that represent private attributes of the card holder, plus configuration data that controls the usage of the card—in particular the release of its contents. Each claim value is of a particular *claim type*, such as last name, gender, or email address.

There are two kinds of cards, *self-issued* and *managed*. A self-issued card is created by the user's client computer, and may contain up to 15 claims with standardised claim types. The claim data is stored at the client, typically protected by a password or a PIN. A managed card is created by an external identity provider, and may contain any kind of claims, encapsulating for instance application-specific data, much like browser cookies. The claim data is stored at the identity provider, but not at the client. Hence, the usage of the card requires online access to the identity provider, as configured within the card. In communications between the client and the card issuer, a card is identified by a *card reference*, or *cardId*, which uniquely identifies the card at the card issuer. The card reference is stored at the client, and is not revealed to the relying party.

A user may hold a collection of cards of both kinds, may create new self-issued cards, and may request managed cards from identity providers. A relying party may require its user to present cards with some specific claim types; the user may then obtain an *authentication token* containing the corresponding claims from the identity provider and present the token to the relying party.

2.2 Roles and Principals

Figure 1 depicts a typical usage of the InfoCard protocol. We describe the roles involved in the protocol; for each role, we give examples of principals that may instantiate the role.

User (U). The user is the human who owns a card and wishes to control its usage to sign on to relying parties, while preserving some privacy.

Identity Selector (IS). The identity selector is the secure subsystem that manages collections of cards on behalf of the user; it interacts with the user for card selection and communicates with identity providers. It is also the repository for the private data associated with self-issued cards. The IS could for instance be part of the user's operating system, as is the case for Windows CardSpace.

Client Application (A). The client application handles ordinary interactions between users and relying parties. It could for instance consist of an applet hosted by the browser, or could be a web service client. It is typically less trusted than IS; it usually has no access to the card-generated tokens carrying claim data.

Client Security Protocol Layer (C). We distinguish a security protocol layer running at the local machine as the role C. Both the IS and A roles communicate with remote parties through this security protocol layer. It must be trusted in that, depending on the protocol configuration, it may manipulate unencrypted forms of both application and card data.

Relying Party (RP). A relying party is a server that requires a card-based token. It may be a website (such as `http://relay.labs.live.com`) or a SOAP web service.

A relying party is identified and authenticated by its X.509 certificate. Servers using the same certificate are assumed to be part of the same entity, typically an organisation recognised by the user. InfoCard defines a procedure for extracting and presenting this organisation information from an X.509 certificate. A relying party publishes a policy, either as part of its web page, or from a metadata server, stating the user authentication requirements. This policy may require that the user provide a card-based token issued by a particular identity provider and asserting specific identity claims about the user. In addition, it may state its privacy policy concerning the usage of user information.

Identity Provider (IP). An identity provider issues and manages cards for users. It is typically implemented as a website with a web service for issuing security tokens (such as `http://sts.labs.live.com`).

An identity provider is identified and authenticated by its X.509 certificate. It stores user authentication material and claim data for the cards it has issued in a secure card database. It also publishes a policy that specifies its authentication requirements; for instance, it may require that the user secure her token request with a password. The design of InfoCard is "IP-centric": the identity provider must be online, and must be trusted to provide accurate card information to the user and secure card-based tokens to the relying party. However, the user can prevent information about the relying party from being sent to the identity provider, hence limiting the usage information that the identity provider can collect.

2.3 Messages and Events

We now describe the messages and events in a typical interaction between roles in the InfoCard protocol, as depicted in Figure 1. Section 4 defines several configurations of the protocol. The configuration used here is our running example; we refer to it as UserPassword-SOAP. For simplicity, we restrict our attention to the core three-party protocol described in the InfoCard specifications. In practice, this protocol may be integrated into more general scenarios that have, for instance, more exchanges between C and RP before or after card-based authentication, or that require C to contact several IPs.

We write $R : E$ for a local event E at a role R, representing an internal communication. We write $R_1 \rightarrow R_2 : M$ for a network message from R_1 to R_2 that carries the data in M. To protect M, the network message needs to be constructed and checked using cryptography. In this section we omit the cryptography so as to emphasise the message contents; Section 5 provides the details.

The protocol begins when the client application A, running on behalf of user U, wishes to make a service request at the relying party RP. We assume that U has already obtained a card, with reference *cardId* and some claims from an identity provider IP with whom U shares a password $pwd_{\mathrm{U,IP}}$. We further assume that the

client security protocol layer C has retrieved the security policies of RP and IP out-of-band. Given this pre-established configuration data, the core protocol proceeds as follows. (The numbering of events here corresponds to the numbering of interactions given in Figure 1.)

(1) C : *Request* (RP, M_{req})

The security protocol layer C receives a request message M_{req} from the client application A for a relying party RP.

(2) U : *Select InfoCard* (*cardId*, IP, RP, $pwd_{\mathrm{U,IP}}$, $types_{\mathrm{RP}}$)

From the previously retrieved policies of RP and IP, C extracts the claim types $types_{\mathrm{RP}}$ required by RP and triggers the identity selector, which prompts the user U to select an appropriate information card issued by IP for use at RP, and to provide the associated password $pwd_{\mathrm{U,IP}}$ as authentication credential. The user should carefully review the identities of IP and RP and the required $types_{\mathrm{RP}}$ before selecting a card.

(3) C → IP : *Request Token* (*cardId*, $pwd_{\mathrm{U,IP}}$, RP, $types_{\mathrm{RP}}$)

As the first network message, C sends a token request to IP. This request includes the card reference, RP's identity, and the types of the required claims. The message is secured in accordance with IP's policy, using U's password for authentication.

(4) IP : *Issue Token* (U, *cardId*, RP, *claims*, *display*)
IP → C : *Token Response* (RP, *claims*, *display*)

After authenticating and authorising the token request, IP issues a security token that contains attribute values *claims* for the required $types_{\mathrm{RP}}$, and returns it to C. IP encrypts the token for RP and provides a human-readable display token *display* that contains attributes of the issued token, such as data identifying the card and RP.

(5) U : *Approve Token* (*display*)

The identity selector presents the display token to the user for her approval. The user should approve the token only if the displayed attributes match her selected card and its intended usage.

(6) C → RP : *Service Request* (M_{req}, *claims*)

C sends a service request to RP for *A*'s original request M_{req}. It secures this message in accordance with RP's policy, and endorses the message with the issued token, which contains *claims*.

(7) RP : *Accept Request* (IP, *claims*, M_{req}, M_{resp})
RP → C : *Service Response* (M_{resp})

RP authorises the request based on the issuer IP and *claims*, and responds with M_{resp}.

(8) C : *Response* (RP, M_{req}, M_{resp})

The protocol ends with C forwarding RP's response to A.

2.4 Protocol Configuration Options

There are many configurable elements in the InfoCard protocol. We highlight some of the security-critical options below.

User Credentials. The protocol configuration in Figure 1 assumes the user shares a password with the identity provider, and uses this password to authenticate her request for a token for a managed card. Alternatively, the user may use other types of credentials, such as self-issued cards or smartcard-based X.509 certificates, to authenticate token requests.

Server Policies. Both the identity provider and relying party are identified and authenticated by their X.509 certificates. However, the details of their message security policies may differ; for example, they may or may not require message signatures to be encrypted. Section 4 describes some common server policies.

Self-Issued vs Managed Cards. Figure 1 assumes the use of a managed card. For self-issued cards, the identity selector (IS) itself acts as IP. To create a self-issued card, IS generates a fresh card reference (*cardId*) and some card-specific entropy (η_{cardId}, known only to IS) and stores them with the claims in an encrypted card store. When issuing a token for a self-issued card, IS signs the token with a key $k_{cardId,\mathrm{RP}}$ generated using η_{cardId} and the X.509 certificate of the target relying party. We refer to the key generation algorithm as K and write:

$$k_{cardId,\mathrm{RP}} = K(\eta_{cardId}, \mathrm{RP}) \qquad \text{(computed by IS)}$$

To create a managed card, the identity provider (IP) generates the card reference and a card master key (k_{cardId}, known only to IP) and stores them with the claims; it reusing of a

not. In particular, a relying party RP may ask for a *private personal identifier* (PPID) that it can use to correlate different requests that use the same card, for example to retrieve server-state associated with the user. Informally, a PPID acts as a pseudonym for the card user interacting with RP. To protect user privacy, the PPID should prevent coalitions of relying parties from correlating visits to different RPs by the same user (identified by the same card). Hence, there are two desired properties of a PPID; first, the PPIDs computed for a given card and a given relying party must always be the same; second, the PPIDs computed for a given card and different relying parties must be different and difficult to correlate.

In practice, a PPID may be computed in three different ways. First, when IP issues a token, based on a managed card *cardId*, with token scope limited to RP, it computes a cryptographic hash based on the card master key k_{cardId} and RP's X.509 certificate, possibly with some additional entropy stored with the card:

$$ppid_{cardId,\mathrm{RP}} = H_1(k_{cardId}, \mathrm{RP}) \qquad \text{(computed by IP)}$$

Second, when IP issues a token, based on a managed card *cardId*, but with unlimited scope, C must first send it a PPID seed specific to RP, computed as a hash of the card salt η_{cardId} and RP's certificate; IP uses this seed to compute a PPID using the card master key:

$$seed_{cardId,\mathrm{RP}} = H_2(\eta_{cardId}, \mathrm{RP}) \qquad \text{(computed by IS)}$$
$$ppid_{cardId,\mathrm{RP}} = H_3(k_{cardId}, seed_{cardId,\mathrm{RP}}) \qquad \text{(computed by IP)}$$

Third, when IS generates a token based on a self-issued card, it computes the PPID based on *cardId* and RP's certificate:

$$ppid_{cardId,\mathrm{RP}} = H_4(cardId, \mathrm{RP}) \qquad \text{(computed by IS)}$$

2.5 Security Goals

An implementation of the InfoCard protocol consists of principals playing each of the roles U, A, IS, C, RP, and IP; the first is played by a human, while the others are played by programs. The behaviour of these roles varies depending on the choice of configuration options. However, irrespective of the configuration, every implementation of InfoCard is expected to satisfy the authentication and secrecy goals stated in this section. The formal statements below are our interpretation of properties expressed informally in the specifications. We assume the symbolic threat model of Dolev and Yao [1983]: the adversary may intercept, compute upon, and send messages; it controls some principals; and it may actively interact with any number of runs of the protocol. We say that a principal is *compromised* when it is controlled by the adversary; otherwise we say that it is *compliant*. We assume that the adversary is given the secrets, such as passwords and keys, of all compromised principals, but cannot guess the secrets of compliant principals.

Our formal statements rely on the following assumptions. Let there be five compliant principals: a user A running a client security protocol layer C, an identity provider I, and two relying parties R and R'. The principal A uses C to make service requests at both R and R', and both relying parties accept tokens issued by I with claim types $types_R$. The principals A and I share a password $pwd_{A,I}$, and I has issued exactly two cards to A, with references *cardId* and *cardId*$'$, both with the same secret data $claims_A$ of type $types_R$. The master keys, entropy, and salt corresponding to the two cards are kept secret. There may be other compromised users, identity providers, and relying parties, but we assume that A does not have any interaction with them.

Authentication Properties. We specify the target authentication properties as correspondence assertions [Woo and Lam, 1993] between protocol events. We write the assertion: $E \Rightarrow E_1 \wedge \ldots \wedge E_n$ to mean that in any run of a protocol, whenever the event E occurs, the events $E_1, \ldots, E_n$ must have previously occurred. All quantifications of variables are explicit. Elements that are not quantified are constants. We write $[v]$ to indicate an optional element v in a message.

In every run of an InfoCard implementation, even in the presence of an active adversary, the properties **A1**–**A3** must hold:

A1 Whenever I issues a token with $claims_A$ for any relying party RP (possibly R or R'), A must have selected the card and claim types for use at RP.

$$\begin{array}{l}\forall cid, \mathrm{RP}, display.\\ \quad I : \mathit{Issue\ Token}(A, cid, [\mathrm{RP}], claims_A, [display]) \Rightarrow\\ \quad A : \mathit{Select\ InfoCard}(cid, I, \mathrm{RP}, pwd_{A,I}, types_R)\end{array}$$

Since both RP and *display* are optional in the *Issue Token* event and the subsequent *Token Response* message, this assertion should be read as a disjunction; either I specifically issues a token for RP $= R$, and then A must have selected R (and not say R') as the relying party; or A may have selected any relying party.

A2 Whenever R accepts a request M_{req} with a token carrying $claims_A$, the client C must have made the request, the user A must have selected the card and approved the token, and I must have issued the token.

$$\begin{array}{l}\forall M_{req}, M_{resp}.\\ \quad R : \mathit{Accept\ Request}(I, claims_A, M_{req}, M_{resp}) \Rightarrow\\ \quad \exists cid, display.\\ \quad\quad C : \mathit{Request}(R, M_{req}) \wedge\\ \quad\quad A : \mathit{Select\ InfoCard}(cid, I, R, pwd_{A,I}, types_R) \wedge\\ \quad\quad I : \mathit{Issue\ Token}(A, cid, [R], claims_A, [display]) \wedge\\ \quad\quad A : \mathit{Approve\ Token}([display])\end{array}$$

A3 Whenever C accepts from R a service response M_{resp}, R must have sent it in response to its request M_{req}.

$$\begin{array}{l}\forall M_{req}, M_{resp}.\\ \quad C : \mathit{Response}(R, M_{req}, M_{resp}) \Rightarrow\\ \quad\quad C : \mathit{Request}(R, M_{req}) \wedge\\ \quad\quad R : \mathit{Accept\ Request}(I, claims_A, M_{req}, M_{resp})\end{array}$$

Property **A1** specifies A's authentication at I after processing message (3); **A2** specifies card-based authentication at R after processing message (6); and **A3** specifies response authentication and correlation at C after processing message (7).

Secrecy Properties. Our target secrecy properties encode various user privacy goals of the InfoCard protocol. Let A, C, I, R, R' be principals as before, but now let R' be compromised.

The first privacy goal is that $claims_A$ must never be released without explicit approval from A. Moreover, if the issued token scope is limited to R, then only R may obtain $claims_A$. In the following, *Attacker knows*(X) means that the attacker is able to syntactically derive X by performing standard cryptographic operations on the messages that it has seen on the wire.

S1 The attacker may obtain $claims_A$ only if A released it to some relying party RP.

$$\begin{array}{l}\mathit{Attacker\ knows}(claims_A) \Rightarrow\\ \quad \exists cid, \mathrm{RP}, display.\\ \quad\quad A : \mathit{Select\ InfoCard}(cid, I, \mathrm{RP}, pwd_{A,I}, types_R) \wedge\\ \quad\quad I : \mathit{Issue\ Token}(A, cid, [\mathrm{RP}], claims_A, [display]) \wedge\\ \quad\quad A : \mathit{Approve\ Token}([display])\end{array}$$

S2 If issued token scopes are limited, then the adversary may obtain $claims_A$ only if A released it to R':

$$\begin{array}{l} \mathit{Attacker\ knows}\,(\mathit{claims}_A) \Rightarrow \\ \quad \exists \mathit{cid}, \mathit{display}. \\ \qquad A : \mathit{Select\ InfoCard}\,(\mathit{cid}, I, R', \mathit{pwd}_{\mathrm{A,IP}}, \mathit{types}_{\mathrm{RP}}) \wedge \\ \qquad I : \mathit{Issue\ Token}\,(A, \mathit{cid}, [R'], \mathit{claims}_A, [\mathit{display}]) \wedge \\ \qquad A : \mathit{Approve\ Token}\,([\mathit{display}]) \end{array}$$

The following privacy goal is that if a card does not contain personally identifiable data about the user, then a coalition of relying parties cannot link uses of the card. For this property we assume that both R and R' are compromised, and that the issued token includes a PPID computed for the card and relying party. The property is stated in terms of the equivalence, from the viewpoint of the adversary, between two different versions of the user A.

S3 An implementation of the protocol where A always presents the same card *cardId* to both R and R' is observationally equivalent to an implementation where A always presents *cardId* to R and *cardId*$'$ to R'.

The fourth privacy goal is that if the token scope is not limited, an IP should not be able to tell which RP is being used. To state this property, we assume that I is compromised but both R and R' are compliant, and that A always uses the same card *cardId* at both R and R'.

S4 If the issued token scope is not limited, then an implementation of the protocol where A obtains a token from I for use at R is observationally equivalent to an implementation where A obtains a token from I for the same card for use at R'.

3. WEB SERVICES SECURITY (REVIEW)

The InfoCard protocol is constructed upon the existing framework for web services security. This framework consists of a suite of specifications that include SOAP [Gudgin et al., 2003], WS-Security [Nadalin et al., 2004], WS-SecurityPolicy [Kaler et al., 2005], SAML [Cantor et al., 2005], XML-DSIG [Eastlake et al., 2002a] and XML-ENC [Eastlake et al., 2002b]. The protocols described by each of these specifications are highly configurable and the resulting suite of protocols is flexible in the extreme. In this section, we provide a brief review of the necessary concepts from web services security.

Message formats and security. Messages are formatted in accordance with the SOAP messaging framework, which relies on XML for the underlying representation. To secure a SOAP message between two principals, WS-Security defines mechanisms to sign the message body and chosen headers (using XML-DSIG) and to encrypt the message body and sensitive cryptographic materials (using XML-ENC). Alternatively, a SOAP message can also be secured in its entirety by transmitting it over channels secured by other means, such as TLS [Dierks and Rescorla, 2006]. In this paper, we focus primarily on precisely modelling the former, and simply assume that TLS provides a secure transport.

End-point References. Servers are usually identified by an end-point reference (EPR). These consist of a URI for the service, plus either an X.509 certificate or a Kerberos identity associated with the server principal.

Authentication Tokens. WS-Security sets XML formats for the security tokens used to identify and authenticate principals. Many existing authentication credentials can be encapsulated within security tokens, including username/password pairs, X.509 certificates, Kerberos tickets, and SAML tokens. SAML tokens in particular are commonly used within the InfoCard protocol to convey claims about the client that are asserted by the identity provider.

Security Policies. WS-SecurityPolicy permits a server to use the WSDL language [Christensen et al., 2002] to define and publish its security policy together with a description of the service it provides. The policy specifies the kinds of security tokens acceptable to the service, how message signing and encryption keys are derived from these tokens, which parts of a message should be signed or encrypted, the ordering of signatures and encryptions, and whether a session and associated security context should be established.

To associate a principal with a SOAP message, a security token for the principal is included as a SOAP header. For each type of token, WS-Security specifies ways to derive signing and encryption keys, used to sign or encrypt a message. Moreover, if there is already a message signature, a token can be used to *endorse* the message by adding a secondary signature that covers the primary message signature.

4. MODELLING CONFIGURATIONS

In this section, we present the F# data structures that constitute our formal model of an InfoCard configuration. A configuration sets the parameters for all instances of the roles that may be involved in runs of the protocol. Given the security policies published by RP and IP, their X.509 certificates, and a list of information cards available to the user, we can automatically generate an F# model of the configuration. We illustrate these data structures by describing the UserPassword-SOAP example configuration. We also briefly describe the other configurations used in the paper.

In overview, the UserPassword-SOAP configuration bases user identity on managed information cards issued by IP; RP requests a symmetric proof key to be associated with the issued token; and IP expects U to authenticate herself using a username and password associated with the managed card.

Protocol configurations. We model protocol configurations as records with, for each role of the protocol, a field that sets the configuration for the role. Our example configuration is as follows. (We present the definition top-down, defining each element of this configuration in turn.)

```
let config = {RP = [(rpEpr, rpPolicy)];
              IP = [(ipEpr, ipPolicy, [cardsecret])];
              IS = ([infocard], [])}
```

In the syntax of F#, a record is written as a sequence of equations between { and }; each equation binds a field label, such as RP, to a term, such as [(rpEpr, rpPolicy)]. A list is written as a sequence of terms between [and], while a tuple is a sequence of terms between (and). A constructor application is written as the constructor name followed by parameters between (and), such as Some(ipEpr, ipPolicy, [cardsecret]).

For both RP and IP, the configuration is an association list that maps EPRs (consisting of a URI plus an X.509 certificate) to policies. For example, configurations with only self-issued cards have no IP, so they set IP = []; configurations with a single IP (as is the case for our UserPassword-SOAP example) set IP to a singleton. In addition to a policy, each configuration entry for IP models its database of issued cards as a list of card entries. In the example, only one card has been issued, so the card list associated with ipEpr has a single entry cardsecret associated with the user.

Finally, the configuration for the identity selector IS consists of two lists: a list of card information for the managed cards, as well as a list of card entries for the self-issued cards. In the example, IS specifies a single managed card and no self-issued cards.

Name	Issuer	User credential	Token scope	Token key	Encrypt Sigs
SelfIssued-SOAP (Figure 4)	Self	None	RP	Asymmetric	**true**
UserPassword-TLS	IP	Password	RP	Symmetric	**true**
UserPassword-SOAP (Figure 5)	IP	Password	RP	Symmetric	**true**
UserCertificate-SOAP	IP	X.509 Certificate	Any	Asymmetric	**true**
UserCertificate-SOAP-v	IP	X.509 Certificate	Any	Asymmetric	**false**

Figure 3: Summary of Selected Example Configurations

Our model for EPRs is straightforward. We omit the (standard) representation of X.509 certificates. In the example, we let:

```
let rpEpr = {address="http://rp.com/"; identity=rpX509Cert}
let ipEpr = {address="http://ip.com/"; identity=ipX509Cert}
```

Policies for IP and RP. A web services security policy specifies the tokens that a client must include in a message in order to access a service. Tokens can be included in messages either as *endorsing tokens* or as *supporting tokens*. WS-Security has specific rules for including and proving possession of tokens for each token type. The policy also includes a set of WS-SecurityPolicy options that further qualify the message format. For simplicity, our configurations have at most one endorsing token and one supporting token.

The record below defines the RP policy in the UserPassword-SOAP configuration.

```
let rpPolicy = {endorse = IssuedToken(rpIssuedTokenPolicy);
                support = NoToken;
                options = options}
```

Hence, RP requires an endorsing token but no supporting token. The term rpIssuedTokenPolicy, detailed next, specifies the form of the issued token. The options field sets WS-SecurityPolicy options.

The rpIssuedTokenPolicy identifies the token issuer, and states the claims and type of the proof key required by RP. In the example, we define:

```
let rpIssuedTokenPolicy = {
    issuer = IP(ipEpr);
    rstTemplate = {
        claims = [ClaimType("GivenName"); PPID];
        proofKeyType = SymKey } }
```

This policy identifies IP as the token issuer using its EPR; it requires a token with two claims: a given name and a PPID; it also requires a symmetric proof key associated with the issued token.

The record below defines the IP policy.

```
let ipPolicy = {endorse = NoToken;
                support = UsernameToken;
                options = options; }
```

Unlike RP, IP requires the client to identify itself using a supporting token rather than an endorsing token. In this case, the supporting token is a token derived from the user's username and password.

In the example, RP and IP rely on the same WS-SecurityPolicy, recorded below:

```
let options = {deriveKeys = true; encryptSigs = true;
               signatureConf = false; entropy = Both}
```

Card data. In addition to the policy, the state of IP includes a database of cards that it has issued. In our example, there is a single card, so the IP database is just a singleton list.

The terms below illustrate an entry in this database.

```
let infocard = {reference = "cardid";
                issuer=ip(ipEpr);
                reqAppTo=true;
                userauth=UserPwdCred;
                hashsalt=...}
```

```
let cardsecret = {usercard = infocard;
                  useridentity = UserPassword("user", "****");
                  claims = [(claimtype("name"), "alice")];
                  masterkey = ...}
```

The information card downloaded by the user from IP and locally maintained by IS is represented by values such as infocard, with five fields. The reference field contains a string that identifies the card for the user and IP. The issuer field contains the EPR of the card issuer. The reqAppTo field indicates whether or not the scope of issued tokens based on this card is limited to a specific RP. The userauth field states the kind of credential that the user must present to IP to authenticate herself and retrieve a token associated with the card; in our example, this credential type mandates the use of a username and a password, as is the case in IP's ipPolicy.

The secrets associated with cards are held in a database at IP; the term cardsecret models a database entry. Its fields record: a reference to the card issued to the user; the credentials that the user must present to authenticate herself to IP; a list of claims associated with the card; and a card master secret.

Limits of our Model. Our programming model leaves out some details of the actual InfoCard configuration management. We assume that all policies have been gathered and all discrepancies between them have been resolved. We do not model the indirection of logical EPRs, opting instead for network EPRs only. We assume that all InfoCards support all claims. We ignore the contents of display tokens. We assume that options such as derived keys apply to all tokens in a message or none of them. We support only the SAML 1.1 token type.

Selected configurations. Figure 3 summarises the main options for the five protocol configurations that we formalize and verify. Except for UserPassword-SOAP detailed above, we omit their direct coding as F# data structures. The configuration UserPassword-TLS differs from UserPassword-SOAP only in that it uses TLS to secure messages rather than using SOAP message security.

We have selected these configurations based on two main criteria. First, we aim to cover much of the variation in our model of the issued token and InfoCard policy configurations. Second, as far as possible, we model the scenarios that are presented in the Information Card specifications. In the following sections we give detailed traces for two of these configurations. In Section 5, we prove security properties for the first four. In Section 6, we discuss how certain configuration choices, such as not encrypting signatures in the fifth configuration, can lead to vulnerabilities.

5. DETAILED PROTOCOL NARRATIONS

We present protocol narrations for the first and third configurations of Figure 3. These two configurations of the InfoCard protocol exercise the default recommended security policy options, and are expected to be commonly deployed.

Cryptography. Each narration is a trace of protocol messages and events corresponding to the abstract description in Section 2.3. Our narrations consist of one or two message exchanges. All the optional elements are fixed, in accordance with the specific configura-

Initially,	**C has:** $Card(cardId, claims_{\mathrm{U}}), \mathtt{PK}(k_{\mathrm{RP}})$; **RP has:** k_{RP}	
(1) C :	*Request* (RP, M_{req})	C *receives an application request*
(2) U :	*Select InfoCard* (*cardId*, C, RP, $types_{\mathrm{RP}}$)	*User selects card*
(4) C(IP) :	*Issue Token* (U, *cardId*, $claims_{\mathrm{U}}$, RP, *display*)	C *generates a self-issued token*
(5) U :	*Approve Token* (*display*)	*User approves token*
(6) C :	*generate fresh* $k, \eta_1, \eta_2, (k_{proof}, \mathtt{PK}(k_{proof}))$	*Fresh session key, two nonces, and asymmetric key-pair*
C → RP :	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), k)$ in	*Encrypt session key for* RP
	let $k_{sig} = \mathtt{PSHA1}(k, \eta_1)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k, \eta_2)$ in	*Derive message encryption key*
	let $ppid_{cardId,\mathrm{RP}} = H_4(cardId, \mathrm{RP})$ in	*Compute PPID using card reference,* RP*'s identity*
	let $k_{cardId,\mathrm{RP}} = K(cardId, \mathrm{RP})$ in	*Compute token signing key using card,* RP*'s identity*
	let $M_{tok} = \mathtt{Assertion}(\mathsf{Self}, \mathtt{PK}(k_{proof}), claims_{\mathrm{U}}, \mathrm{RP}, ppid_{cardId,\mathrm{RP}})$ in	*SAML assertion with public key, claims, and PPID*
	let $M_{toksig} = \mathtt{RSASHA1}(k_{cardId,\mathrm{RP}}, M_{tok})$ in	*Self-signed SAML assertion*
	let $M_{saml} = \mathtt{SAML}(M_{tok}, M_{toksig})$ in	*Issued token*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{req})$ in	*Message signature*
	let $M_{proof} = \mathtt{RSASHA1}(k_{proof}, M_{mac})$ in	*Endorsing signature proving possession of* k_{proof}
	Service Request $(M_{ek}, \eta_1, \eta_2, \mathtt{PK}(k_{cardId,\mathrm{RP}}),$ $\mathtt{AESEnc}(k_{enc}, M_{saml}), \mathtt{AESEnc}(k_{enc}, M_{mac}),$ $\mathtt{AESEnc}(k_{enc}, M_{proof}), \mathtt{AESEnc}(k_{enc}, M_{req}))$	*Request, with encrypted token, signatures and body*
(7) RP :	*Accept Request* (C, $claims_{\mathrm{U}}, M_{req}, M_{resp}$)	RP *accepts request and authorizes a response*
RP :	*generate fresh* η_3, η_4	*Fresh nonces*
RP → C :	let $k_{sig} = \mathtt{PSHA1}(k, \eta_3)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k, \eta_4)$ in	*Derive message encryption key*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{resp})$ in	*Message Signature*
	Service Response $(\eta_3, \eta_4, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{resp}))$	*Service Response, with encrypted signatures and body*
(8) C :	*Response* (M_{resp})	C *accepts response and sends it to application*

Figure 4: Protocol Narration for the SelfIssued-SOAP Configuration

tion, and the network messages display all their cryptographic components, as required by the security policies at the identity provider and the relying party. The cryptographic algorithms used in our narrations follow from web services security:

- for key derivation, we use the PSHA1 function that takes a symmetric key and a nonce to generate a new symmetric key;
- for key encryption, we use RSA encryption (RSAEnc) and write $\mathtt{PK}(k)$ for the public key corresponding to private key k;
- for message encryption, we use AES (AESEnc);
- for message signatures and endorsing signatures based on symmetric keys, we use HMACSHA1;
- for endorsing signatures based on asymmetric keys, we use RSA signature (RSASHA1).

The computations of PPIDs and token signing keys use the algorithms K, H_1, H_2, H_3, and H_4 described in Section 2.4.

A common pattern for exchanging messages. Exchanges secured using web services security follow a common pattern (called a SymmetricBinding) that relies on the server having an X.509 certificate. Under this binding, a request sent by an initiator A to a responder B is of the form

$$(M_{ek}, \eta_1, \eta_2, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{req}))$$

where M_{ek} is an encrypted key token that contains a fresh symmetric session key k generated by A and encrypted under $\mathtt{PK}(k_B)$. The initiator A also derives two keys, k_{sig} and k_{enc} from k plus fresh nonces, η_1 and η_2; it then signs the message body M_{req} with k_{sig}, which yields the message signature M_{mac}; it finally encrypts both M_{req} and M_{mac} with k_{enc}. Upon receiving this message, B decrypts k, derives k_{sig} and k_{enc} using the nonces, decrypts M_{mac} and M_{req} using k_{enc}, and verifies the signature M_{mac} over the message body M_{req} using k_{sig}.

The response message from B has a similar structure; it includes two fresh nonces used by B to derive keys, an encrypted signature of the message body M_{resp}, and the encrypted message body. Since A knows the session key k, it decrypts the message and verifies the signature before accepting it.

This pattern of message exchange authenticates B using its X.509 public key, and provides secrecy and correlation between request and response. However, it does not authenticate A to B. To this end, the request may contain additional authentication tokens for A, and endorsing signatures based on these tokens. These tokens and signatures are then also encrypted using k_{enc}.

SelfIssued-SOAP Configuration. Figure 4 depicts the protocol narration for the client security protocol layer C (on behalf of a user U and the client application A) sending a request to a relying party RP, based on a self-issued card. The numbers on the left refer to steps in the abstract description of the protocol in Section 2.3. The narration consists of a single message exchange between C and RP. (There is no separate IP.) C begins with a card with reference *cardId* and some secret attributes $claims_{\mathrm{U}}$; RP starts with an X.509 certificate with public key $\mathtt{PK}(k_{\mathrm{RP}})$ known to C.

The narration is divided into three stages. The first stage details the interaction between the user U and the identity selector, as U selects and approves a card to send to RP. The events in the first stage correspond to the protocol events numbered (1), (2), (4), and (5) in Section 2.3; here, C itself acts as the issuer, so the network messages *Request Token* and *Token Response* are not present (and so there is no counterpart to event (3) from Section 2.3). The second stage details the *Service Request* from C to RP, while the third stage details the *Service Response*.

The message exchange follows the SymmetricBinding pattern described above; in addition, the *Service Request* message from C to RP contains a self-signed card-based SAML token M_{saml} and a corresponding endorsing signature M_{proof}. C constructs M_{saml} as follows: it first generates a fresh asymmetric key-pair (k_{proof}, $\mathtt{PK}(k_{proof})$) for the token; it computes a PPID $H_4(cardId, \mathrm{RP})$ and a token signing key $k_{cardId,\mathrm{RP}}$ specific to the card and RP; it then constructs a SAML assertion M_{tok} that identifies the token issuer, the token public key $\mathtt{PK}(k_{proof})$, $claims_{\mathrm{U}}$, RP, and $ppid_{cardId,\mathrm{RP}}$; the full SAML token M_{saml} consists of this assertion and its signature using key $k_{cardId,\mathrm{RP}}$. To prove possession of the token key k_{proof}, C signs the message signature M_{mac} using k_{proof} creating an en-

Initially,	**C has:** $cardId$, $\mathtt{PK}(k_{\mathrm{IP}})$, $\mathtt{PK}(k_{\mathrm{RP}})$; **IP has:** k_{IP}, $\mathtt{PK}(k_{\mathrm{RP}})$, $Card(cardId, claims_{\mathrm{U}}, pwd_{\mathrm{U,IP}}, k_{cardId})$; **RP has:** k_{RP}, $\mathtt{PK}(k_{\mathrm{IP}})$	
(1) C :	$Request(\mathrm{RP}, M_{req})$	C *receives an application request*
(2) U :	$Select\ InfoCard(cardId, \mathrm{C}, \mathrm{RP}, pwd_{\mathrm{U,IP}}, types_{\mathrm{RP}})$	*User selects card and provides password*
(3) C :	*generate fresh* $k_1, \eta_1, \eta_2, \eta_{ce}$	*Fresh session key, two nonces, and client entropy for token key*
C → IP :	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{IP}}), k_1)$ in	*Encrypt session key for* IP
	let $k_{sig} = \mathtt{PSHA1}(k_1, \eta_1)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_1, \eta_2)$ in	*Derive message encryption key*
	let $M_{rst} = \mathtt{RST}(cardId, types_{\mathrm{RP}}, \mathrm{RP}, \eta_{ce})$ in	*Token request message body*
	let $M_{user} = (\mathrm{U}, pwd_{\mathrm{U}})$ in	*User authentication token*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, (M_{rst}, M_{user}))$ in	*Message signature*
	$Request\ Token(M_{ek}, \eta_1, \eta_2, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{user}), \mathtt{AESEnc}(k_{enc}, M_{rst}))$	*Token Request, with encrypted signatures, token and body*
(4) IP :	$Issue\ Token(\mathrm{U}, cardId, claims_{\mathrm{U}}, \mathrm{RP}, display)$	IP *issues token for* U *to use at* RP
IP :	*generate fresh* $\eta_3, \eta_4, \eta_{se}, k_t$	*Fresh nonces, server entropy, token encryption key*
IP → C :	let $k_{sig} = \mathtt{PSHA1}(k_1, \eta_3)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_1, \eta_4)$ in	*Derive message encryption key*
	let $M_{tokkey} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), \mathtt{PSHA1}(\eta_{ce}, \eta_{se}))$ in	*Compute token key from entropies, encrypt for* RP
	let $ppid_{cardId,\mathrm{RP}} = H_1(k_{cardId}, \mathrm{RP})$ in	*Compute PPID using card master key,* RP*'s identity*
	let $M_{tok} = \mathtt{Assertion}(\mathrm{IP}, M_{tokkey}, claims_{\mathrm{U}}, \mathrm{RP}, ppid_{cardId,\mathrm{RP}})$ in	*SAML assertion with token key, claims, and PPID*
	let $M_{toksig} = \mathtt{RSASHA1}(k_{\mathrm{IP}}, M_{tok})$ in	*SAML assertion signed by* IP
	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), k_t)$ in	*Token encryption key, encrypted for* RP
	let $M_{enctok} = (M_{ek}, \mathtt{AESEnc}(k_t, \mathtt{SAML}(M_{tok}, M_{toksig})))$ in	*Encrypted issued token*
	let $M_{rstr} = \mathtt{RSTR}(M_{enctok}, \eta_{se})$ in	*Token response message body*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{rstr})$ in	*Message Signature*
	$Token\ Response(\eta_3, \eta_4, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{rstr}))$	*Token Response, with encrypted signature and body*
(5) U :	$Approve\ Token(display)$	*User approves token*
(6) C :	*generate fresh* $k_2, \eta_5, \eta_6, \eta_7$	*Fresh session key, three nonces*
C → RP :	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), k_2)$ in	*Encrypt session key for* RP
	let $k_{sig} = \mathtt{PSHA1}(k_2, \eta_5)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_2, \eta_6)$ in	*Derive message encryption key*
	let $k_{proof} = \mathtt{PSHA1}(\eta_{ce}, \eta_{se})$ in	*Compute token key from entropies*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{req})$ in	*Message signature*
	let $k_{endorse} = \mathtt{PSHA1}(k_{proof}, \eta_7)$ in	*Derive a signing key from the issued token key*
	let $M_{proof} = \mathtt{HMACSHA1}(k_{endorse}, M_{mac})$ in	*Endorsing signature proving possession of token key*
	$Service\ Request(M_{ek}, \eta_5, \eta_6, \eta_7, M_{enctok}, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{proof}), \mathtt{AESEnc}(k_{enc}, M_{req}))$	*Service Request, with issued token, encrypted signatures and body*
(7) RP :	$Accept\ Request(\mathrm{IP}, claims_{\mathrm{U}}, M_{req}, M_{resp})$	RP *accepts request and authorizes a response*
RP :	*generate fresh* η_8, η_9	*Fresh nonces*
RP → C :	let $k_{sig} = \mathtt{PSHA1}(k_2, \eta_8)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_2, \eta_9)$ in	*Derive message encryption key*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{resp})$ in	*Message signature*
	$Service\ Response(\eta_8, \eta_9, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{resp}))$	*Service Response, with encrypted signatures and body*
(8) C :	$Response(M_{resp})$	C *accepts response and sends it to application*

Figure 5: Protocol Narration for the UserPassword-SOAP Configuration

dorsing signature M_{proof}. Hence, the *Service Request* consists of the encrypted session key M_{ek}, two nonces, the encrypted SAML token, an encrypted message signatures, an encrypted endorsing signature, and the encrypted body.

After receiving and decrypting this message, RP verifies M_{mac}, verifies the token issuer's signature M_{toksig} on the SAML assertion M_{tok}, uses the public key $\mathtt{PK}(k_{proof}))$ in M_{tok} to verify M_{proof}, and checks that the $claims_{\mathrm{U}}$ in M_{tok} satisfies its required $types_{\mathrm{RP}}$, before accepting the request M_{req} and sending the response M_{resp}. The response is signed and encrypted according to the SymmetricBinding pattern.

UserPassword-SOAP Configuration. Figure 5 depicts the protocol narration for C sending a request to RP based on a managed card issued by IP. Client C begins with a reference for the managed card; IP begins with its own private key and a single card in its database; RP begins with its own private key. Each principal also knows the others' public keys. The first stage of the narration is similar to the SelfIssued-SOAP case, where U selects a particular card, this time including a password for that card. The remainder of the narration consists of two exchanges, first between C and IP to obtain an issued token and then between C and RP, where C authenticates itself to RP using the issued token.

The first message follows a SymmetricBinding pattern, with the addition of a supporting username token M_u that serves to authenticate U to IP. M_u is included in the M_{mac} along with the body of the message M_{rst} and is included after encryption in the message. M_{rst} identifies RP to limit the scope of the issued token. It also contains the card reference, the claims requested by RP, and the client entropy contribution η_{ce} to the symmetric proof key. In addition to the SymmetricBinding checks, IP authenticates the user using the supporting username token.

The construction by IP of a SAML token is similar to the self-issued case. However, since this configuration requires a symmetric proof key, IP first derives a key from the entropies η_{se} and η_{ce}, then encrypts the resulting key for RP in M_{tokkey}. Additionally, the token signature uses IP's X.509 private key instead of a key derived from the card. The entire SAML token is encrypted for RP following a standard pattern in WS-Security for encrypting data—a fresh symmetric key k_t is encrypted to RP's public key (M_{ek}) and the SAML token is encrypted using k_t. The response body M_{rstr} includes this encrypted token as well as the server entropy η_{se}.

After verifying IP's response, C computes the proof key for the

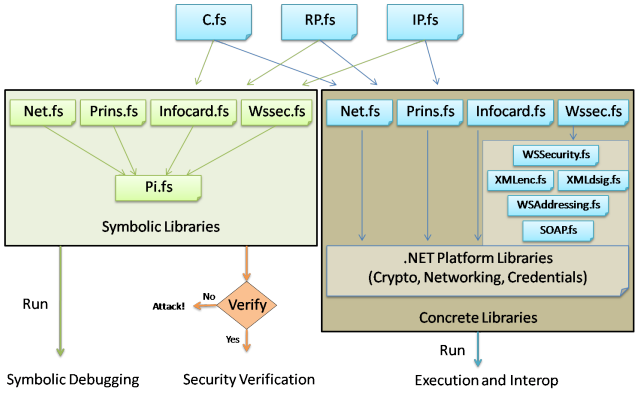

Figure 6: Dual Structure of our Verified Implementation

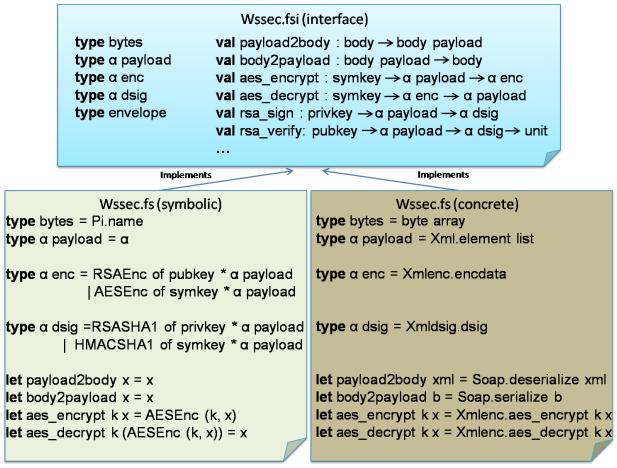

Figure 7: Dual Implementations of Wssec

token, k_{proof}, from the entropies. To prove to RP that it possesses this key, C derives an endorsing key $k_{endorse}$ from k_{proof}, and uses it to produce an endorsing signature M_{proof}. Hence, the message to RP includes M_{proof} as well as the encrypted issued token M_{enctok}. Finally, RP's response follows the SymmetricBinding pattern.

6. VERIFIED IMPLEMENTATIONS

To obtain a verifiable implementation of each protocol configuration, we follow an approach proposed by Bhargavan et al. [2006b]. This approach (illustrated in Figure 6) involves developing dual implementations of the protocol in F#; a *concrete* implementation generates XML envelopes and uses concrete cryptography, while a *symbolic* implementation uses an abstract model for message formats and cryptography.

The top of Figure 6 mentions three files implementing the protocol-specific code for the roles C, RP, and IP, respectively, for some given configuration. This code is written using libraries that implement various networking and cryptographic functions defined by the web services security specifications. Each of these libraries has two implementations.

The libraries on the right of Figure 6 represent concrete implementations. When protocol-specific code is linked with these libraries we obtain a reference implementation of the protocol that produces standards-conformant SOAP envelopes. We validate this implementation by testing interoperability with other independent implementations of the protocol.

The libraries on the left represent a symbolic model for cryptography, a concurrency model based on the pi calculus for communications, and an abstract message format. When protocol-specific code is linked and executed with these libraries, we obtain a symbolic trace of the protocol, as illustrated in Figures 4 and 5; such traces are useful for debugging. More importantly, we can extract and verify a model of the protocol implementation (by using an enhanced version of the FS2PV tool and ProVerif, respectively).

An Abstract API for WS-Security.

Figure 7 shows fragments of the main F# interface Wssec.fsi and of its symbolic and concrete implementations. The interface declares several abstract types and exposes functions that manipulate these types. For instance, the type bytes is implemented concretely as a byte array; in the symbolic library we model bytes as pi calculus names. Thus, for instance, when our code calls the library to generate a fresh nonce, its implementation either uses a pseudo-random number generator to generate an array of bytes, or symbolically creates a new name in the pi calculus.

In both implementations, SOAP envelopes are represented by the envelope datatype: a record with fields for the various components of an message. We abstract the wire formats of envelopes by introducing the polymorphic type α payload. This type represents values of type α after they have been serialised; concretely, this type is implemented as a list of XML elements. For instance, the interface exposes the functions payload2body and body2payload. In the concrete case, these functions use our SOAP library for (de)serialising the message body. In the symbolic case, no serialisation is performed; both of them are modelled as identity functions. Similarly, other functions enable (de)serialisation of all message components.

The types α enc and α dsig represent encrypted and signed values. In the concrete case, they are implemented using our libraries for XML encryption and XML digital signatures. In the symbolic case, they are modelled as algebraic types, with a different constructor for each kind of cryptographic operation. Figure 7 shows a partial list of functions in the interface (such as aes_encrypt and rsa_sign) that perform cryptographic operations. The types of these functions ensure that only serialised data (of type α payload) can be encrypted or signed.

Our protocol implementations use additional libraries with dual implementations: the Prins.fs library implements functions for manipulating credentials such as X.509 certificates; the Infocard.fs library implements card stores; the Net.fs library implements networking functions, with interface

```
type connection
val send: (connection → envelope → unit)
val receive: (connection → envelope)
```

In this library, a connection is a TCP or TLS channel between principals, which can be used to send and receive SOAP envelopes.

Infocard-specific Role Implementations. The code for roles C, RP, and IP implement functions with the following types (defined in the interfaces C.fsi, RP.fsi, IP.fsi):

```
val C: request → response
val RP: response → unit
val IP: unit → unit
```

The function C is a proxy for the client application; it expects to be called with the request as a parameter and returns the relying party's response. Similarly, RP is a proxy for the relying party service. Finally, IP starts a process that serves token requests according to the secret card data of an IP database (see Section 4).

For the configuration UserPassword-SOAP, the term shown below represents the envelope constructed by the function C for the *Request Token* message:

Configuration	LOC	Cryptographic operations	Verified security properties	Clauses	Verification time
SelfIssued-SOAP(Figure 4)	1410 (80)	9, 3	**A2**-**A3**, **S1**–**S3**	800	38s
UserPassword-TLS	1426 (96)	0, 5, 17, 6	**A1**–**A3**, **S1**–**S3**	3800	24m 40s
UserPassword-SOAP(Figure 5)	1429 (99)	9, 11, 17, 6	**A1**–**A3**, **S1**–**S3**	3000	20m 53s
UserCertificate-SOAP	1429 (99)	13, 7, 11, 6	**A1**–**A3**, **S1**, **S3**–**S4**	2400	66m 21s
UserCertificate-SOAP-v	1429 (99)	7, 5, 7, 4	**A3** *fails*	1200	10s

Figure 8: Verification Results for the Example Configurations

```
{headers = hdrs;
 token = encKeyTok;
 dsig = aes_encrypt kenc (hmacsha1 ksig (RST(r), hdrs, userTok));
 supptok = aes_encrypt kenc userTok;
 endorsements = [];
 body = aes_encrypt ksig (RST(r))}
```

It uses signing and encryption functions from Wssec.fs, and relies on some precomputed values: hdrs contains addressing headers, encKeyTok is the encrypted session key M_{ek}, ksig and kenc are the derived signing and encryption keys, userTok is the username token, and RST(r) is the token request.

The next IP code fragment runs after it has received the *Request Token* message, decrypted the body RST(r) and the username token (u,p), and verified the signature:

```
let card = lookupCard u r.cardid in
  check (card.user = u);
  check (card.password = p);
  let rstr = mkSamlRSTR r.keyinfo card in ...
```

The code looks up a card in its database (using the lookupCard function from Infocard.fs), checks that the username and password correspond to the card, and constructs a token response with a SAML token for the card.

The next RP code fragment runs after it has received a message including a SAML token; it checks the token as follows:

```
let Saml(assertion, asig) = decryptSamlTok rpKey encSamlTok in
  rsasha1_verify ipPubKey assertion asig;
  let symProofKey = decryptAssertionKey rpKey assertion in ...
```

It decrypts the token using its private key rpKey, verifies that the assertion is signed by IP, and decrypts the symmetric token key embedded within the SAML assertion.

These code fragments illustrate that the Wssec.fs abstraction for web services security enables simple, compact protocol-specific coding. Every line expresses a cryptographic operation or checks a protocol requirement. There is no need to mention any detail of the underlying XML formats.

Automated Generation of Protocol-Specific Code. We have constructed a tool, called WSDL2FS, which we use to automatically generate the protocol-specific role programs illustrated above.

As explained in Section 3, WS-SecurityPolicy permits a server to publish a WSDL specification of its security policy. Given an EPR for a relying party, WSDL2FS downloads its WSDL policy, resolves the EPRs for all identity providers mentioned in this policy, and recursively downloads the IP policies.

WSDL2FS also attempts to parse the policies and the collection of information cards that the user possesses into the data structures described in Section 4. In order for this step to succeed, it must be able to solve several potential discrepancies between the downloaded policies and the information cards. For instance, the relying party may require an endorsement using an issued token with scope limited to itself, whereas the identity provider may not support such a facility for any of the information cards that it has issued to the user. If no discrepancies remain, our tool generates configuration-specific F# programs for each of the three roles RP, IP and C.

Given a set of security goals (such as those of Section 2.5), we translate the generated role programs together with the symbolic WS-Security libraries to a model in ProVerif. If ProVerif is able to discharge the proofs, the user can choose to link her application with the implementation of C and the concrete libraries, and access the service provided by the relying party.

Experimental Results. We generate full protocol implementations for all the configurations of Section 4; each implementation has around 100 lines of protocol-specific code; the concrete code for our libraries is around 5000 lines of F#, while the symbolic version has around 1300 lines. (The bulk of the concrete code involves serialising and de-serialising XML formats.)

We can run the same protocol code both symbolically, to generate symbolic traces, and concretely, to send messages over the network. The symbolic traces generated by our SelfIssued-SOAP and UserPassword-SOAP implementations coincide exactly to Figures 4 and 5. Our concrete implementations interoperate with IPs and RPs that use other web services libraries, such as Microsoft Windows Communication Foundation (WCF). As a first experiment, we target the protocol between Windows Cardspace as client, the identity provider `http://sts.labs.live.com`, and a relying party implemented as a WCF web service. This is one of the first publicly available identity providers for managed cards. The protocol configuration corresponds to our running example UserPassword-SOAP; our client implementation produces and consumes all four SOAP messages, with sizes 14KB, 10KB, 15KB, and 5KB, respectively. We aim to experiment with other IPs and ISs as they become available.

7. VERIFICATION RESULTS

We establish that our implementations meets the authentication and secrecy properties given in Section 2.5.

We define the attacker interface I_{pub} as C.fsi, RP.fsi, IP.fsi, Net.fsi, and Wssec.fsi. Hence, the attacker can initiate any number of sessions between the client C, IP, and RP; she can intercept and send messages on the network using functions in Net.fsi; and she can decrypt, encrypt, and sign messages using functions in Wssec.fsi. Our target security assertions are stated in terms of principals U, C, IP, and RP that are not compromised by the attacker. However, the interface I_{pub} includes the X.509 certificates of IP and RP, and the user credential and card belonging to a bad user. Moreover, the attacker can generate new certificates and deploy her own servers.

Our first security theorem deals with correspondences. Formally, we say that a program S (consisting of several modules) is *robustly safe* for a correspondence assertion q and an attacker interface I_{pub} to mean that, for every attacker module O that is well-typed against I_{pub}, and hence only uses the functions in this interface, the assertion q holds in all runs of the program S composed with O.

For each InfoCard configuration C, let S_C be the program consisting of the symbolic libraries (Prins.fs, Infocard.fs, Net.fs, and Wssec.fs) and the protocol-specific C.fs, RP.fs, and IP.fs; let I_{pub} be defined as above.

THEOREM 1 (CORRESPONDENCES). *For each of the first four configurations* C *of Figure 3, the program S_C is robustly safe for the correspondence assertions **A1–A3**, **S1–S2** and I_{pub}.*

This theorem is established using an automated tool chain: a model extractor FS2PV parses the program S_C, optimises it using transformations such as inlining, dead-code elimination, and partial evaluation, and then compiles it to an applied pi calculus model representing all the roles of the protocol. The correspondence assertions are encoded as queries over protocol events in the syntax of ProVerif, which then automatically verifies that the script satisfies these queries.

THEOREM 2 (EQUIVALENCES). *For each of the first four configurations* C *in Figure 3, the program S_C with interface I_{pub} satisfies the equivalence properties **S3–S4**.*

To prove these properties, we use FS2PV to compile two versions of the pi calculus script from S_C, and use ProVerif to verify that these versions are behaviourally equivalent [Blanchet et al., 2005].

Figure 8 summarises our verification results for our five example configurations. Not all security properties apply to every configuration; for instance, **A1** applies only to configurations with managed cards, and vacuously holds with self-issued cards since there is no IP. For each configuration, we list the security properties verified using our tools; the others trivially hold. The column LOC refers to the number of lines of verified F# code (the protocol-specific code is in parentheses); For each message of the protocol, Cryptographic operations gives the number of cryptographic calls used to build the message; Clauses refers to the number of logical rules generated by ProVerif during verification; and Verification time gives the total runtime of ProVerif to verify all the security properties (using a machine with 2 GB of memory and an Intel Pentium 4 at 3.60 GHz).

Vulnerable Configurations and Attacks. The last configuration UserCertificate-SOAP-v in Figure 8 fails to verify property **A3**. This configuration uses weak security policies for IP and RP that do not require signatures to be encrypted. Messages with unencrypted signatures are known to be vulnerable to man-in-the-middle attacks in some cases. Indeed, ProVerif discovers such an attack on our implementation. We describe the attack in terms of the message elements in Figure 5.

The adversary intercepts the *Service Request* message from C to RP, removes the issued token M_{saml} and its endorsing signature M_{proof}, inserts her own card-based token M'_{saml} (which she may have obtained from IP using her own card), and signs the unencrypted message signature M_{mac} with the token key k'_{proof} for M'_{saml}. When RP receives this message, it performs all the checks as before, and then accepts the request M_{req} as being associated with the adversary's card and not the user's. (This already constitutes an attack on user card-based authentication.) It then computes M_{resp}, possibly relying on details of the incorrect card claims in M'_{saml}, secures it as usual and sends it to the client C, who accepts it, since nothing in the response tells C about the card accepted at RP. Hence, the protocol concludes without error, with C and RP having inconsistent states: RP has an incorrect card associated with the request, and C has a response computed using someone else's card.

A similar attack is found when IP's policy does not require encrypted signatures: an adversary can then modify the *Request Token* message in the same way. However, if IP checks that the *cardId* corresponds to the user and password (as our implementation does), then the attack does not succeed. Conversely, if IP does not perform these checks (and some existing IPs do not) it may issue a card-based token based on an incorrect user credential. Even then, if it provides an accurate *display*, the user should be able to detect the attack and should refuse to approve the token.

In configurations with self-issued cards, card references should be generated with sufficient entropy to be unguessable (rather than, say, as predictable sequence numbers) and should be kept secret. Otherwise, ProVerif finds a violation of the secrecy property **S3**. In the configuration SelfIssued-SOAP (Figure 4), the PPID is computed using the function H_4 using the card reference and information from RP's X.509 certificate. If the *cardId* were guessable or known to RP, then since the other component of this computation is a publicly known certificate, two colluding RPs can easily correlate usages of the same card at different RP. Hence, the privacy of the user depends on the *cardId* being a strong secret.

Each of these three vulnerabilities stems from incorrect usages of the protocol; we describe them here to warn users of potential pitfalls. On the basis of our analysis, we recommend that IP and RP use strong security policies, and that identity selectors generate cryptographically random *cardId*s and keep them secret.

8. RELATED WORK

Cameron [2005] frames the social and technological background to InfoCard in terms of a collection of informal Laws of Identity.

Formal models for protocols based on the web services standards WS-Security, WS-SecurityPolicy, and WS-Trust were developed during the standardisation process, with various design bugs being discovered and fixed [Bhargavan et al., 2004, Bhargavan et al., Kleiner and Roscoe, 2004, 2005, Backes et al., 2006]. Bhargavan et al. [2004] also describe how to compile and verify declarative web services policies from high-level security goals.

Pfitzmann and Waidner [2005] describe the security properties of some earlier federated identity protocols, including Passport, Liberty, Shibboleth, and SAML, although they do not develop a formal analysis. Hansen et al. [2006] model SAML in the process calculus LySa. Via static analysis, they check some security properties of some configurations, and report vulnerabilities in others.

We verify secrecy and authentication properties of symbolic protocol models, derived directly from a reference implementation that interoperates with existing implementations. Several tools extract implementation code from verified formal models [Perrig et al., 2001, Muller and Millen, 2001] or generate implementation code using verified protocol compilers [Corin et al., 2007]—as opposed to extracting models from code. We are unaware of any such code-generation tools that have been developed sufficiently to interoperate on any standard protocols.

Goubault-Larrecq and Parrennes [2005] are the first to derive symbolic protocol models from implementation code, although they report no verification results. Their code is a C implementation of the Needham-Schroeder protocol. Although this protocol is a standard example in the research literature, it is not used in practice and has no standard binary format. Hence, their code does not pass any interoperability tests.

Bhargavan et al. [2006b,a] are the first to extract verifiable symbolic models from interoperable implementation code. They also consider security protocols based on web services, though the protocols implemented by their code are considerably simpler than the CardSpace protocols in our study.

Giambiagi and Dam [2004] and Poll and Schubert [2007] check conformance between the implementation code of security protocols and abstract models. They do not verify any security properties of the abstract models. In Schubert and Poll's study, the abstract model is a state machine for SSH, and the code is a pre-existing (and presumably interoperable) Java implementation of SSH.

9. CONCLUSION

InfoCard is a good example of the trend toward composition of protocols from large collections of independently standardised specifications. Regarding this trend, Pfitzmann and Waidner [2005] argue that "from a security point of view, the standards cannot currently be considered modular, because the first layer for which one can specify and prove usual security goals is typically the highest one". They describe this trend as a "challenge to the security-research community". Our work shows that the idea of verified reference implementations goes some way to meeting this challenge.

Acknowledgments

We thank Martín Abadi and Arun Nanda for detailed discussions on drafts of this paper.

References

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

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

M. Backes, S. Mödersheim, B. Pfitzmann, and L. Viganò. Symbolic and cryptographic analysis of the secure WS-ReliableMessaging scenario. In *Foundations of Software Science and Computation Structures (FOSSACS)*, LNCS. Springer, 2006.

K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Computer Science*, 340(1):102–153.

K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, October 2004.

K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-Security protocols. In *WS-FM '06*, volume 4184 of *LNCS*. Springer, 2006a.

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

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

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340, 2005.

K. Cameron. The laws of identity. At `http://www.identityblog.com/stories/2005/05/13/TheLawsOfIdentity.pdf`, 2005.

Scott Cantor, John Kemp, Rob Philpott, and Eve Maler. Assertions and protocols for the oasis security assertion markup language (saml) v2.0, 2005.

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

R. Corin, P-M. Dénielou, C. Fournet, K. Bhargavan, and J.J. Leifer. Secure implementations of typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF20)*, pages 170–186, 2007.

T. Dierks and E. Rescorla. The transport layer security (tls) protocol, version 1.1, April 2006. URL `http://www.ietf.org/rfc/rfc4346.txt`.

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

D. Eastlake, J. Reagle, D. Solo, et al. *XML-Signature Syntax and Processing*, 2002a. URL `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`. W3C Recommendation.

D. Eastlake, J. Reagle, et al. *XML Encryption Syntax and Processing*, 2002b. URL `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/`. W3C Recommendation.

A. Armando et al. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *17th Conference on Computer Aided Verification (CAV)*, LNCS, pages 281–285. Springer, 2005.

P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Science of Computer Programming*, 50:73–99, 2004.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.

M. Gudgin et al. *SOAP Version 1.2*, 2003. URL `http://www.w3.org/TR/soap12`. W3C Recommendation.

S. M. Hansen, J. Skriver, and H. Riis Nielson. Using static analysis to validate the SAML single sign-on protocol. In *Workshop on Issues in the Theory of Security (WITS'06)*, pages 27–40, 2006.

M. Jones. *A Guide to Supporting Information Cards within Web Applications and Browsers as of the Information Card Profile V1.0*. Microsoft Corporation, December 2006. At `http://go.microsoft.com/fwlink/?LinkId=88956`.

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

E. Kleiner and A. W. Roscoe. Web services security: A preliminary study using Casper and FDR. In *Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.

E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. In *Mathematical Foundations of Programming Semantics (MFPS XXI)*, 2005.

InfoCard Guide. *A Guide to Interoperating with the Information Card Profile V1.0*. Microsoft Corporation and Ping Identity Corporation, December 2006. At `http://go.microsoft.com/fwlink/?LinkId=87446`.

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

A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, March 2004. OASIS Standard 200401.

A. Nanda. *A Technical Reference for the Information Card Profile V1.0*. Microsoft Corporation, December 2006. At `http://go.microsoft.com/fwlink/?LinkId=87444`.

OSIS: The Open-Source Identity System. OSIS, 2006. At `http://osis.netmesh.org/wiki/`.

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

B. Pfitzmann and M. Waidner. Federated identity-management protocols. In *11th International Workshop on Security Protocols (2003)*, volume 3364 of *LNCS*, pages 153–174. Springer, 2005.

E. Poll and A. Schubert. Verifying an implementation of SSH. In *Workshop on Issues in the Theory of Security (WITS'07)*, 2007.

D. Syme. *F#*, 2005. Project website at `http://research.microsoft.com/fsharp/`.

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.