Verified Reference Implementations of WS-Security Protocols

Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon

Microsoft Research

Abstract. We describe a new reference implementation of the web services security specifications. The implementation is structured as a library in the functional programming language F#. Applications written using this library can interoperate with other compliant web services, such as those written using Microsoft WSE and WCF frameworks. Moreover, the security of such applications can be automatically verified by translating them to the applied pi calculus and using an automated theorem prover. We illustrate the use of our reference implementation through examples drawn from the sample applications included with WSE and WCF. We formally verify their security properties. We also experimentally evaluate their interoperability and performance.

1 Introduction

XML web services offer a standards-based framework for deploying secure networked applications. Using SOAP [16] to serialize data, WS-Addressing [10] to identify endpoints, WS-Security [24] to protect messages, and HTTP or TCP as transport, programmers can deploy clients and servers that can operate across different platforms.

To this end, the WS-Security standard defines a security header for SOAP messages that may include signatures, ciphertexts, key identifiers, and tokens identifying particular principals. Environments such as Apache WSS4J [3], IBM WebSphere [17], and Microsoft Web Services Enhancements (WSE) [20] and Windows Communication Foundation (WCF) [21], provide tools and libraries for building web services that are secured via the mechanisms of WS-Security and related specifications.

In general, even if an attacker is unable to compromise the underlying cryptographic algorithms used in a protocol, there may be successful attacks based on intercepting, rewriting, and sending messages, as noted by Needham and Schroeder [25] and later formalized by Dolev and Yao [11]. Due to the flexibility of composable specifications and the semi-structured nature of the XML message format, WS-Security protocols are actually more prone to message rewriting attacks than protocols based on binary formats. In particular, studies of the usage of WS-Security reveal a wide range of vulnerabilities to message rewriting attacks [5,6,4,18,19]. Hence, it is essential to verify the security of WS-Security protocol implementations before deployment.

Almost all verification tools for cryptographic protocols analyze abstract models rather than implementations. For instance, the ProVerif [9,8] theorem prover takes a protocol model written in a variant of the pi calculus [23,2] plus target authentication and secrecy goals, and attempts to prove that the model satisfies these goals. So, to verify the security of a web services protocol implementation, one may write a detailed formal model for the protocol by studying the standards, by carefully observing the messages it sends, or by reading its source code. Using such models, previous analyses establish correctness theorems [14,5,4,18,19] and report attacks [5,6] on many WS-Security protocols. Still, writing formal models remains difficult and time-consuming; hence, this approach is typically applied only to common protocols. Even for these protocols, a precise and detailed formal model is lengthy, and its fidelity to the implementation is difficult to maintain.

In earlier work [7], we present an automated verification method for security protocol implementations written in F# [26], a dialect of ML. Our tool, named fs2pv, relies on the ProVerif theorem prover to verify that an F# program meets its security goals in the presence of an active attacker. The capabilities of the active attacker can be flexibly defined as a programming interface that lists all the values and functions of the protocol that the attacker may access. Our earlier work demonstrates the effectiveness of these tools on several protocol implementations, including protocol implementations based on WS-Security, and establishes a general theorem stating the correctness of our method.

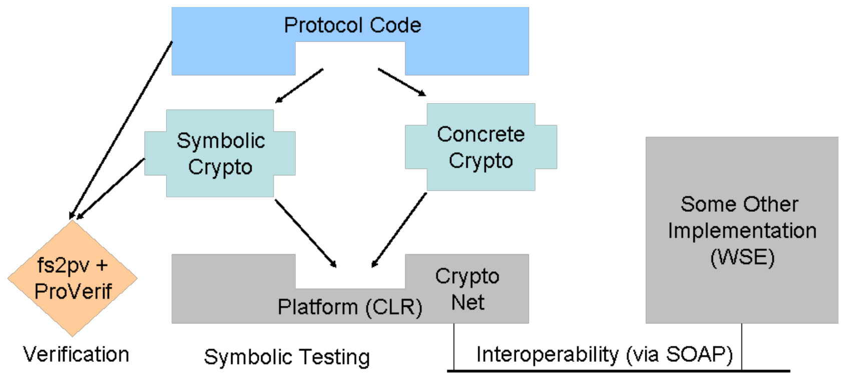
The present paper complements and extends this work by elaborating the details of our verifiable programming style for WS-Security. We propose to build reference implementations for WS-Security protocols in F#. We develop a verified library that partially implements WS-Security and its related specifications. With this library, we can quickly implement, test, and verify new protocol implementations. Our reference implementations are readable, succinct, and verified.

The contributions of this paper are as follows:

1. A description of the design and architecture of a reusable library for building web services and verifying their security. Our library supports a significant subset of the specifications for web services security and can interoperate with other web services implementations.
2. A detailed case study of the implementation and verification of a WS-Security security protocol. To the best of our knowledge, the thousand line pi calculus process we verify is the largest model of a cryptographic protocol to be extracted from code. We provide interoperability results and performance comparisons; as a benchmark, our implementations pass interoperability tests with at least two production implementations, Microsoft WSE and WCF. We also give formal security guarantees for this protocol, established by running verification tools and instantiating general theorems that justify our method.

Our earlier paper discusses related work, including tools that derive implementation code from models. We are aware of only one other tool that extracts models from cryptographic protocol implementations, Goubault-Larrecq and Parrenne's Csur [15]. Their tool extracts Horn clauses from C code; it has been applied successfully to the Needham-Schroeder protocol.

The structure of the rest of the paper is as follows. Section 2 recalls the verification method developed in our previous work. Section 3 details the implementation and verification of a WS-Security X.509 mutual authentication protocol. Section 4 presents ver-



ification results for some WS-Security protocol implementations. Section 5 describes the structure of our WS-Security library. Section 6 concludes.

2 Verifying Security Protocol Implementations in F# (Review)

The F# programming language [26] is a dialect of ML that executes on the Common Language Runtime (CLR). The figure above shows the structure of our formal method for verifying protocol models that are derived from the F# code of security protocols. This section outlines our method; the description draws in part on material included in our earlier paper [7].

Our tool fs2pv captures the semantics of an expressive subset of F# by translating F# implementation code to the dialect of the applied pi calculus [2] analyzed by the ProVerif theorem prover [8]. The core of our translation is Milner's interpretation of functions as pi calculus processes [22]. Still, we implement many optimizations to take advantage of features of ProVerif and to facilitate automated verification. Our translation, and the analysis performed by ProVerif, rely on a symbolic, algebraic representation of cryptography, as first proposed by Dolev and Yao [11]. We conjecture that our method could be adapted to other source languages whose semantics can be directly represented in the pi calculus, and that other tools could be used to analyze the translated pi calculus processes.

Dual Implementations for Trusted Libraries. Each of our protocol implementations is a composition of typed F# modules. Each module exports types, values, and functions, and may depend on other modules. We write standard F# interface files to describe the types and the typed values and functions provided by a module.

Ideally, we would construct our pi calculus model of a protocol entirely from the actual source code of its modules. For a few, trusted libraries, however, we instead write a dual, symbolic implementation. We assume (but do not formally verify) that the symbolic implementation of a library is an appropriate abstraction of its concrete implementation. These symbolic abstractions correspond to Dolev and Yao's algebraic treatment of cryptography and networking. For example, our protocols depend on an interface crypto.fsi, shown in Table 1, to perform cryptographic algorithms used for

crypto.fsi (excerpt)

```
type keybytes
val rsa_encrypt: keybytes → bytes → bytes
val rsa_decrypt: keybytes → bytes → bytes
val sha1: bytes → bytes
val rsa_sign: keybytes → bytes → bytes
val rsa_verify: keybytes → bytes → bytes → unit
```

prins.fsi (excerpt)

```
type principalX =
          {subject:str;
           cert: bytes;
           pubkey: keybytes;
           privkey: keybytes;}
val genX509: str → unit
val getX509Cert: str → bytes
val leakX509: str → principalX
```

net.fsi

```
val request: (str → str → item → item)
val accept: (str → item)
val respond: (item → unit)
```

Table 1. The Attacker's Interface to the Trusted Libraries

web services security. The concrete library implements the abstract type bytes as actual byte arrays, and the various functions as actual cryptographic algorithms, as provided by CLR libraries. The symbolic library implements bytes as an algebraic data type; a function such as rsa_encrypt becomes a constructor of this datatype, while the function rsa_decrypt is defined by pattern-matching on the datatype. We also define dual implementations for an interface prins.fsi, that provides access to the operating system security context, and an interface net.fsi, that provides networking capabilities.

We write S for the *symbolic implementation* of a protocol in F#: the composition of all the modules of a protocol, but with the symbolic code instead of the concrete code for those trusted libraries with dual implementations. This is the code that fs2pv translates to the pi calculus. Our method does not verify the concrete code of the library modules with dual implementations, it is not included in S; it is trusted, not verified.

As well as verifying S, we can build a symbolic version of the protocol implementation by compiling S. Running this symbolic implementation generates readable messages, containing symbolic representations of cryptographic materials, useful for prototyping and debugging.

The Attacker Interface. The aim of the analysis is to prove security properties such as message authentication and secrecy in the face of an attacker able to monitor, rewrite, and substitute messages sent between the machines playing a role in a protocol.

We envisage the attacker as a top-level F# module that can call some but not all of the modules making up a protocol implementation. The *attacker interface*, I_{pub}, models the capabilities of the attacker; it is expressed as the concatenation of the interfaces for the modules that are deemed accessible by the attacker. (The list of these modules is an input of our verification tool; their selection is an important step of our method, and should reflect the informal threat model for the protocol.) This attacker interface typically includes the three interfaces in Table 1—to allow the attacker communication and cryptographic facilities—plus functions representing protocol roles—to allow the attacker to start arbitrary numbers of initiators and responders, for example. We write $S :: I_{pub}$ to mean that the symbolic implementation S correctly exports (at least) the types, values, and functions in I_{pub}. We can check $S :: I_{pub}$ with the F# typechecker.

Queries for Authentication and Secrecy. We express authentication properties as correspondences between protocol events, in the style of Woo and Lam [27]. For instance, suppose a principal A begins a protocol with some parameters P; before sending the first message, it logs an event Begin(P). Then, if a principal B ends the protocol, logging the event End(P′), an authentication goal would be that A and B agree on these parameters (P = P′). In particular, P may include the name of principal A (to ensure sender authentication), the contents of the message (to ensure message authentication), and the content of related messages (to ensure correlation and session integrity).

Similarly, we express syntactic secrecy properties as correspondences: whenever the attacker obtains a value s marked as secret, the attacker can trigger the logging of the event NotSecret(s); hence, s remains secret only if this event is not reachable.

In ProVerif syntax, these correspondences are represented by *queries*:

query ev:End(P) $\Longrightarrow$ **ev**:Begin(P).
query ev:NotSecret(s) $\Longrightarrow$ **ev**:Unreachable().

The first query says that in any run of the program, if event End(P) occurs, then event Begin(P) must have occurred before. The second query says that NotSecret(s) is unreachable. (We arrange that **ev**:Unreachable() occurs in no run of the program.) In general, queries may include conjunctions (&) and disjunctions (|) of events on the right hand side of the implication.

We say that S is *robustly safe* for q and I_{pub} to mean that, for every attacker module O that is well-typed against I_{pub}, the query q holds in all runs of the symbolic implementation S composed with the top-level module O. The attacker interface I_{pub} typically excludes the function for logging events, so the attacker O cannot log events itself. The formal details are elsewhere [7].

Automated Verification of Authentication and Secrecy. For any symbolic implementation S with attacker interface I_{pub}, our verification method consists of the following steps. First, we identify the attacker interface I_{pub} and represent our authentication and secrecy goals as ProVerif queries. Second, we run fs2pv to generate a ProVerif script, written $[\![S :: I_{pub}]\!]$. Third, we run ProVerif to check the script for each query q.

The following theorem states the correctness of our method. It follows as a corollary of the correctness of ProVerif [1] and the correctness of fs2pv [7]. The proof of the latter involves defining a direct semantics for the F# programs S accepted by fs2pv, and proving a correspondence between the direct semantics of each S and its pi calculus translation $[\![S :: I_{pub}]\!]$.

Theorem 1. *Suppose that $S :: I_{pub}$ and that $[\![S :: I_{pub}]\!]$ is the* ProVerif *script generated by* fs2pv *from S and I_{pub}. If* ProVerif *terminates having proved that $[\![S :: I_{pub}]\!]$ satisfies the query q, then S is robustly safe for q and I_{pub}.*

3 X.509 Mutual Authentication

As our main case study, we consider a mutual authentication protocol based on X.509 public key certificates. Both WSE and WCF already implement this protocol as part of their sample code.

We begin with an informal narration of the protocol, then provide a complete implementation in F#. The code is quite short, as it mostly relies on our WS-Security libraries. We describe executions of the protocol, both symbolically (to produce readable message traces) and concretely (to evaluate its performance). We also report on interoperability testing with the WSE and WCF implementations. Finally, we present verification results for this implementation.

Protocol Narration. The protocol has two roles, a client and a server. Every session of the protocol involves a principal A acting as client and a principal B acting as server. Each principal is associated with an RSA key-pair, consisting of a private key and a corresponding public key; A's key-pair is written (sk_A, pk_A), and B's key-pair is written (sk_B, pk_B). We assume that the principals have already exchanged their public key certificates. Hence, the principals can identify one another using their public keys.

The goal of the protocol is to exchange two XML messages: a request and a response, such that both the client and server can authenticate the two-message session and keep the messages secret, even in the presence of an active attacker. To accomplish this goal, we rely on XML digital signatures and XML Encryption. The abstract message sequence of the protocol can be written as follows (where $|$ denotes concatenation):

$$\begin{array}{rl}
A \rightarrow B : & \mathit{TS}\ | \\
& \text{RSA-SHA1}\{\mathit{sk}_A\}[\mathit{request} \mid \mathit{TS}]\ | \\
& \text{RSA-Encrypt}\{\mathit{pk}_B\}[\mathit{symkey}_1]\ | \\
& \text{AES-Encrypt}\{\mathit{symkey}_1\}[\mathit{request}] \\
B \rightarrow A : & \text{RSA-SHA1}\{\mathit{sk}_B\}[\mathit{response} \mid \text{RSA-SHA1}\{\mathit{sk}_A\}[\mathit{request} \mid \mathit{TS}]]\ | \\
& \text{RSA-Encrypt}\{\mathit{pk}_B\}[\mathit{symkey}_2]\ | \\
& \text{AES-Encrypt}\{\mathit{symkey}_2\}[\mathit{response}]
\end{array}$$

The client acting for principal A sends a message *request* at time *TS* to the server acting for B. To support message authentication, the client jointly signs *request* and *TS* using the signature algorithm RSA-SHA1 keyed with A's private key sk_A. To protect the secrecy of the message, the client uses AES-Encrypt to encrypt it under a fresh symmetric key $symkey_1$. The symmetric key is in turn encrypted using RSA-Encrypt under pk_B. (This standard, two-step encryption is motivated by the relative costs of symmetric and asymmetric encryptions for large messages.)

The server repeatedly processes request messages. After accepting a request, the server returns a *response* to the client. Like the request, the response is signed (using sk_B) then encrypted (using a fresh $symkey_2$ encrypted under pk_A). To correlate requests and responses, the server jointly signs the response and the signature value of the request. (Otherwise, since clients and servers may run several sessions in parallel, an attacker may confuse the client by swapping two responses.) This correlation mechanism is called *signature confirmation*.

The security goals of the protocol are:

Request Authentication: B accepts a *request* from A with timestamp *TS* only if A sent such a *request* with timestamp *TS*.

Response Authentication and Correlation: A accepts a *response* to its *request* only if B sent *response* on receiving A's *request*.

Secrecy: The message payloads *request* and *response* are kept secret from all principals other than A and B.

Implementation. Our protocol implementation is listed as X509MutualAuth.fs. The module consists of four functions: mkEnvelope and isEnvelope generate and check the protocol messages, while client and server implement the two protocol roles.

To parse and generate standards-compliant SOAP envelopes, and to sign and encrypt XML elements, we rely on functions of the web services security library. As an example, consider the mkEnvelope function. Depending on its arguments, mkEnvelope constructs either a request message or a response message. To construct a request, it takes a message body containing the *request*, the X.509 entry snd for the sending principal A, the X.509 certificate rcvcert for the receiving principal B, and an empty list corr. (When constructing a response, snd is the X.509 entry for B, rcvcert is the X.509 certificate for A, and corr contains the signature value of the request.) The code for mkEnvelope successively calls the following library functions, defined in modules wssecurity.fs and soap.fs:

- mkTimestamp and genTimestamp create a new timestamp and serialize it to XML;
- mkX509Signature generates the XML digital signature for the message;
- mkX509Encdatakey generates the two encrypted components;
- mkX509SecurityHeader generates the security header;
- genEnvelope generates the whole SOAP envelope for the message.

Finally, the function returns the envelope (for sending) paired with its signature value (kept for correlating the response).

Unlike mkEnvelope and isEnvelope, the client and server functions are part of the attacker interface; both these functions are included in the interface X509MutualAuth.fsi for the protocol module X509MutualAuth.fs. Hence, an attacker can call these functions to initiate sessions and instantiate roles.

The four arguments to client are the name of the client and server principals (clPrin, srvPrin), and the HTTP URI and SOAP action (servUri, servAction) that identify the server location. The client first calls the request function from the service.fs module (described in the next subsection) to compute the XML request payload (req). It then instantiates both principals; it gets the X.509 entry (cl) for clPrin from a private database;

X509MutualAuth.fs

```
(* Opening Library Modules *)
open Data (* Standard datatypes: str, bytes, item *)
open Events (* Protocol Events *)

(* Constructing Messages *)
let mkEnvelope (body:item) (snd:Prins.principalX) (rcvcert:bytes)
               (corr:item list) : item*bytes =
  let ts = Wssecurity.genTimestamp(Wssecurity.mkTimestamp()) in
  let (dsig,sv) = Wssecurity.mkX509Signature snd (body::ts::corr) in
  let (ed,ek) = Wssecurity.mkX509Encdatakey rcvcert body in
  let sec = Wssecurity.mkX509SecurityHeader (Prins.cert snd) ek ts dsig in
  let envXml = Soap.genEnvelope {Soap.header=[sec]; Soap.body=ed} in
  (envXml,sv)

(* Checking Messages *)
let isEnvelope (envXml:item) (sndcert:bytes) (rcv:Prins.principalX)
               (corr:item list) : item*bytes =
  let env = Soap.parseEnvelope envXml in
  let ([sec],ed) = (env.header,env.body) in
  let (ts,ek,dsig) = Wssecurity.isX509SecurityHeader sec in
  let body = Wssecurity.isX509Encdatakey rcv ek ed in
  let sv = Wssecurity.isX509Signature dsig sndcert (body::ts::corr) in
  (body,sv)

(* Client Role *)
let client (clPrin: str) (srvPrin:str) (servUri:str) (servAction:str) =
  let req = Service.request() in
  let cl = Prins.getX509 clPrin in
  let srvCert = Prins.getX509Cert srvPrin in
  let (reqXml,sv) = mkEnvelope req cl srvCert [] in
  log (ClientSend(clPrin,srvPrin,req));
  let respXml = Net.request servUri servAction reqXml in
  let sc = Wssecurity.genSigConf sv in
  let (resp,_) = isEnvelope respXml srvCert cl [sc] in
  log (ClientCorr(clPrin,srvPrin,req,resp))

(* Server Role *)
let server (clPrin:str) (srvPrin:str) (servUri:str) =
  let clCert = Prins.getX509Cert clPrin in
  let srv = Prins.getX509 srvPrin in
  let reqXml = Net.accept servUri in
  let (req,sv) = isEnvelope reqXml clCert srv [] in
  log (ServerRecv(clPrin,srvPrin,req));
  let resp = Service.response(req) in
  let sc = Wssecurity.genSigConf sv in
  let (respXml,_) = mkEnvelope resp srv clCert [sc] in
  log (ServerCorr(clPrin,srvPrin,req,resp));
  Net.respond respXml
```

the entry consists of an X.509 certificate and its associated private key; it then extracts the certificate (srvCert) for the server principal srvPrin. Next, it prepares the request message (reqXml), using mkEnvelope, logs an event ClientSend(clPrin,srvPrin,req) to indicate that it is sending the first message, and makes an HTTP request to the server, using Net.request. The client remembers the signature value (sv) of the request for correlating the response. When the client receives a response (respXml), it uses isEnvelope to check that the response message is valid and that it includes a signature confirmation (sc) echoing sv. It then logs the event ClientCorr(clPrin,srvPrin,req,resp) indicating that a valid response has been received and correlated with the request.

The server proceeds symmetrically: it uses the client certificate and the server X.509 entry to check requests and issue responses. After accepting a request, the server logs an event ServerRecv(clPrin,srvPrin,req); it then calls Service.response(req) to compute the response resp, and logs the event ServerCorr(clPrin,srvPrin,req,resp) before issuing the response.

Protocol Execution. To run the protocol, we write a main module X509Main.fs, listed below. (This module is not used for verification; formally, it is just a simple instance of the attackers considered in our theorems.)

```
let clntPrin = S "client.com"
let srvPrin = S "localhost"
do match Sys.argv.(1) with
  | "client" → client clntPrin srvPrin Service.uri Service.action;
  | "server" → server clntPrin srvPrin Service.uri;
  | "local" → Pi.fork (fun () → server clntPrin srvPrin Service.uri);
                client clntPrin srvPrin Service.uri Service.action
```

This module first instantiates the client and server principals (identified by their X.509 common names “client.com” and “localhost”), and then runs either the client, or the server, or both, depending on the command-line argument. The X509Main.fs module is used only for executing the protocol; they are not used for verification.

We also write a module service.fs to encode an exemplary addition service. The module consists of two functions: Service.request extracts two numbers from the command line and returns them in a request body; Service.response computes the sum of the two numbers in a request and returns it in a response body.

For verification, we write a dual, symbolic implementation of this module that generalizes the two functions by allowing the attacker to choose some payloads: the symbolic version of Service.request (Service.response) returns a request (response) body that it either received from the attacker or it computed from a secret value. Hence, our security goals require request and response authentication even when the attacker is allowed to choose arbitrary payloads, and require secrecy of the secret payloads.

Symbolic runs. To run the protocol symbolically, we compile the X509MutualAuth.fs and X509Main.fs modules with the web services library and the symbolic version of the modules crypto.fs, net.fs, prins.fs, and service.fs to generate an executable run.exe. We can then execute the command run local 100 15.99, for example. Our implementation pretty-prints the communicated messages, using an abbreviated XML-like format with embedded symbolic expressions. The first message has 304 symbols while the second has 531. Both messages are listed and described in the appendix.

Concrete runs and Performance. To run the protocol concretely, we compile X509MutualAuth.fs, X509Main.fs, and the web services library with the concrete versions of crypto.fs, net.fs, prins.fs, and service.fs to generate a new run.exe. We can then execute the command run server on one machine, and execute run client 100 15.99 on another. The resulting 4-kilobyte messages are instances of the symbolic messages, where each symbol expression is replaced by a concrete, string-encoded value.

To test our concrete implementation for interoperability, we run our client with servers implemented with WSE and WCF. The response message generated by the WCF server does not include the X.509 certificate of the server, since the client is expected to have it already. We easily modify our client to ignore this difference and it successfully executes the protocol with WCF. The WSE server, however, does not support the <SignatureConfirmation> mechanism. Moreover, the key-sizes and encryption algorithms supported by WSE are different from and more limited than WCF. After disabling correlation and using WSE's key sizes and algorithms, our client successfully executes the protocol with the WSE server.

Each session of our implementation takes 1.2 seconds to complete the protocol. We expect that this is comparable to the performance of the WSE and WCF implementations because all three implementations use the same .NET cryptography libraries, XML parsers, and X.509 certificate stores. Indeed, in the default configuration, both WSE and WCF take around one second per session for our protocol. A direct comparison of the performance of the three protocol implementations has little significance, because WCF, and to a lesser extent WSE, is a full web services implementation running within a web server, whereas ours is a partial implementation focusing on security. The WSE implementation consists of around 185 lines of C# code, while the WCF implementation consists of around 70 lines of C# code and 160 lines of security-related XML configuration. In contrast, our implementation consists of 104 lines of F# code that can be executed concretely or symbolically, as well as automatically verified.

Security Goals and Theorem. We use the fs2pv/ProVerif tool chain to verify our protocol implementation against its security goals. Recall the three security goals for our protocol. Let G be these security goals expressed as ProVerif queries:

```
query ev:ServerRecv(u,s,x) ⟹ ev:ClientSend(u,_,x) | ev:Leak(u).
query ev:ClientCorr(u,s,x,y) ⟹ ev:ServerCorr(u,s,x,y) | ev:Leak(s).
query ev:NotSecret(v) ⟹
        (ev:ClientSend(u,s,DataTxt(DataBase64(DataFresh(v)))) & ev:Leak(s))
      | (ev:ServerCorr(u,s,r,DataTxt(DataBase64(DataFresh(v)))) & ev:Leak(u)).
```

The first query formalizes request authentication: it says that, if the server principal s accepts a request x from a client principal u (ServerRecv(u,s,x)), then u has sent the request x (ClientSend(u,_,x)) or else u has been compromised. The second query formalizes response authentication and correlation: if the client principal u accepts a response y for request x from server principal s (ClientCorr(u,s,x,y)), then s must have sent the response y to u for request x (ServerCorr(u,s,x,y)).

The third query expresses the secrecy of the request and response. It says that the only secrets v available to the attacker (NotSecret(v)) are those that have been sent within requests or responses to compromised servers or clients, respectively.

Let S be the F# system consisting of the X509MutualAuth.fs module, the web services library, and the symbolic implementations for the modules crypto.fs, net.fs, prins.fs, and service.fs. Let I_{pub} be the attacker interface from Table 1 extended with the protocol interface X509MutualAuth.fsi. We use fs2pv to compile S to a script consisting of 988 lines of pi calculus code. Then we run ProVerif to verify all three queries in G above. By Theorem 1, we obtain:

Theorem 2. *For each $q \in G$, the system S is robustly safe for q and I_{pub}.*

Hence, we verify the security of our protocol implementation and all the functions it uses from the web services library against a powerful attacker model. The only modules we trust to be correct, and do not verify, are crypto.fs, net.fs, prins.fs, and service.fs.

Vulnerabilities and Attacks. Theorem 2 applies to our protocol implementation before modifying it for interoperation with WCF or WSE. The modification for WCF makes no difference to protocol correctness: we automatically establish Theorem 2 for the modified implementation.

The modification for WSE, however, weakens the protocol: the second query (response authentication) fails and ProVerif reports an attack. Indeed, since the modified protocol does not use signature confirmation, an attacker can forward to the client a response generated by the server in reply to another request by the same client. As a result, requests and responses are not securely correlated—this is a known issue in WS-Security 1.0, which led to the design of signature confirmation in WS-Security 1.1. More precisely, we can still capture a weaker notion of response authentication that holds for WSE, using the following, weaker variant of the second query:

query ev:ClientCorr(u,s,x,y) $\Longrightarrow$ **ev**:ServerCorr(_,s,_,y) | **ev**:Leak(s).

We then verify that all variants of our protocol implementation satisfy this query.

The X.509 mutual authentication protocol presented in this section meets our specific set of authentication and secrecy goals, but is not unconditionally secure. We discuss two of its limitations.

– The protocol fails to guarantee certain other security properties. For instance, it fails to protect (stronger variants of) secrecy of *request* or *response* against guessing attacks, when these messages have low entropy. If such protection is required, we can either encrypt the signature in addition to the message content, or we can add a nonce to the message content.
– The protocol also fails to prevent certain replay attacks on the server. If the client produces a new timestamp for each request and if the server maintains a cache of these timestamps, then replays can be detected and discarded. Indeed, our formal model generates fresh timestamps for each message. Alternatively, we can include a unique message identifier in each request.

We also coded stronger variants of the protocol that meet at least the requirements of Theorem 2 and also address these limitations, and verified their implementation using additional queries. We omit the details for simplicity.

Protocol	Implementation				Security Goals and Verification			
	LoC	msgs	bytes	symbols	queries	secrecy	authentication	time
Password-based auth	85	1	3835	394	5	no	msg, sender	5.3 s
X.509 auth	85	1	4650	389	5	no	msg, sender	2.6 s
Pwd-X.509 mutual auth	149	2	6206; 3187	486; 542	15	no	session	44m
X.509 mutual auth	117	2	4533; 4836	304; 531	18	msg	session	51m

Table 2. Verification results for example protocols

Trusted Library			Verified Web Services Library		Protocol Module
Modules	Concrete LoC	Symbolic LoC	Modules	LoC	LoC
4	793 + CLR	575	5	1648	85-149

Table 3. Comparative sizes of implementation modules

4 Other Protocols and Verification Results

In addition to the X.509 Mutual Authentication protocol, we have implemented several other sample WSE and WCF protocols in F# and verified them. Table 2 reports our experimental results. For each protocol, Table 2 states the program size for the implementation (in lines of F# code, excluding interfaces and code for shared libraries), the number of messages exchanged, and the size of each message, measured both in bytes for concrete runs and in number of constructors for symbolic runs. Concerning verification, it gives the number of queries and the kinds of security properties they express. A secrecy query requires that the message body be protected. An authentication query requires that a message, its sender, or the whole session be authentic. All queries are verified assuming that the attacker controls some corrupted principals, and thereby has access to their keys and passwords. Finally, the table gives the total running time for ProVerif to verify all queries for the protocol.

Table 3 lists the sizes (in lines of F# code) of the modules in the protocol implementation, classified as trusted library code, verified web services code, and protocol code. The concrete implementations of the trusted library modules rely on CLR libraries, such as System.Cryptography for cryptographic functions; so, their size cannot be precisely determined.

5 Implementing the Verified WS-Security Library

Programming a security protocol based on WS-Security is an exercise in modularity. The messages of the protocol include elements, such as timestamps, addresses, encrypted keys, and signatures, that are defined by different specifications. Many of these elements eventually rely on low-level cryptographic computations. To assemble the complete SOAP message, each element must be encoded in some XML format.

To support this kind of programming, we structure our WS-Security library as follows. For each specification, we define an F# module Spec.fs and an interface Spec.fsi. Within a module, each high-level message component is defined as a datatype T. Operations to generate and check elements of type T (typically using cryptographic functions) are written as functions mkT and isT. Finally, for each datatype T, the module defines functions genT and parseT to translate elements of T to and from XML items. In this way, users of the library can ignore the XML representation and instead program with the more abstract representation T and its corresponding functions.

For instance, the soap.fs module partially implements the SOAP standard [16]. It has the following interface:

```
type envelope = { header: item list; body: item }
val parseEnvelope: item → envelope
val genEnvelope: envelope → item
```

A SOAP envelope is abstractly represented as a record that contains a list of headers and a body. The functions parseEnvelope and genEnvelope translate such records to and from XML items. Since there is no cryptography involved in constructing an envelope, there are no other functions in the interface.

Similarly, the wsaddressing.fs module implements the headers defined in the WS-Addressing specification [10]; it has a record type that abstractly represents optional headers and it has functions to translate records to and from SOAP header elements.

The full WS-Security library consists of five F# modules, including soap.fs and wsaddressing.fs, with a total of 1648 lines of code. We believe that these modules are usable not only by programmers aiming to write verifiable web services security protocols, but also by protocol designers looking for precise executable specifications for the web services standards. In the rest of this section, we look in more detail at the modules that implement the security mechanisms of WS-Security.

XML Signature. The XML Signature standard "specifies XML syntax and processing rules for creating and representing digital signatures." [13] An XML signature, as defined in the standard, cryptographically attests to the integrity and authenticity of a set of XML items. An example is the <Signature> element in the protocol messages in the appendix. It includes metadata describing the computation of the signature value: each signed element is first transformed using the specified canonicalization method (xml−exc−c14n), then hashed using the specified digest method (SHA1); the digests and metadata are finally signed using the specified signature method (RSA−SHA1). The recipient of such a signature recomputes the digests and checks the received signature value before accepting the signed elements as authentic.

In our library, the xmldsig.fs module implements XML signatures. The datatype for an XML signature is a record dsig that includes the relevant contents of the <Signature> element as well as additional values needed for computing and checking the signature:

```
type dsig = {
    siginfo: item;
    sigval: bytes;
    keyinfo: item;
    signkey: keybytes option;
    verifkey: keybytes option;
    targets: item list }
```

The field siginfo corresponds to the <SignedInfo> element containing the metadata and all the digests; sigval contains the signature value; keyinfo identifies the signing key.

The module contains auxiliary functions for generating siginfo from the list of signed elements (targets). To compute the sigval, we use a signing key (signkey); to check a received sigval, we use the corresponding verification key (verifkey).

The module provides functions for constructing and checking signatures using both symmetric and asymmetric signing algorithms, such as HMAC−SHA1 and RSA−SHA1:

```
val mkSignature: item list → item → keybytes → str → dsig
val isSignature: item list → keybytes → dsig → bytes
```

The function call, mkSignature targets keyinfo signkey alg, constructs a dsig element for the elements listed in targets, using signature key signkey and signing algorithm alg. Conversely, isSignature targets verifkey dsig uses verifkey to check that dsig is a valid XML signature computed from targets. The full module consists of 307 lines of code.

There are several challenges in implementing XML Signature. First, our functions must correctly implement the low-level details of the signature. This includes not only the details of the XML format such as namespaces and attributes, but also the use of the canonicalization, digest, and signature algorithms. In xmldsig.fs, the functions parseSignature and genSignature translate records of type dsig to and from XML. We test these functions by inspecting the message traces as well as by extensive interoperability testing with other implementations. Our datatype and functions hide these details from the programmer, so all programs using these functions are guaranteed to generate standards-conformant XML signatures.

Second, the standard offers several options for each step of signature computation and an implementation is expected to support a subset. In our implementaion, we choose one canonicalization and one digest algorithm, but allow two signature algorithms and several ways of referring to signing keys. These choices do not affect the module interface: the types and functions remain the same. Hence, we can easily add implementations for additional algorithms as the need arises and rely on the F# module and type system to integrate them.

XML Encryption. The XML Encryption standard "specifies a process for encrypting data and representing the result in XML" [12]. When parts of a message are to be encrypted using a symmetric key, the encrypted data mechanism can be used; when only an asymmetric key is available for encryption, one first generates a fresh symmetric key, uses it to encrypt data, and then protects the symmetric key using the encrypted key mechanism. Both these mechanisms are depicted in the protocol messages in the appendix; the <EncryptedData> element contains a cipher value computed by applying a symmetric encryption algorithm (AES−128) to the message body using a key encrypted within an <EncryptedKey> element using an asymmetric algorithm (RSA−1.5).

The xmlenc.fs module implements XML encryption, in a similar style to xmldsig.fs. It defines two record types encdata and encrkey representing encrypted data and encrypted keys. It provides functions to construct (encrypt) and decrypt records of these types and functions to translate them to and from XML. It also provides functions to combine common encryption tasks; for instance, the function call, mkEncDatakey ek str plain, generates a fresh symmetric key, uses it to encrypt the plain-text plain as an encrypted data block, uses the public-key ek to in turn encrypt the symmetric key, and returns both the encrypted data and the encrypted key.

The module xmlenc.fs is implemented in 419 lines of code. It implements two symmetric algorithms for encrypting data, AES−128 and AES−256, and two asymmetric algorithms for encrypting keys, RSA−1.5 and RSA−OAEP. Our choices are motivated by the default settings in WSE and WCF; WSE supports AES−128 and RSA−1.5, while WCF uses AES−256 and RSA−OAEP.

WS-Security. The wssecurity.fs module implements the content of the security header, as specified in the WS-Security standard [24]. The security header contains several optional elements, such as a message timestamp, tokens identifying principals, XML signatures, and encrypted keys. The record representing this header is as follows:

```
type security = {
    timestamp: ts;
    utoks: utok list;
    xtoks: xtok list;
    ekeys: encrkey list;
    dsigs: dsig list }
```

It consists of a timestamp (ts), generated using the mkTimeStamp function, username tokens (utoks) identifying users and passwords, X.509 tokens (xtoks) containing public-key certificates, encrypted keys (ekeys), and XML signatures (dsigs).

The module offers functions for constructing different kinds of tokens and for generating signatures and encrypted blocks using them. For instance, the function call, mkX509Signature prin targets, generates an X.509 token corresponding to principal prin and uses its private key to compute an XML signature for the element list targets. The module also provides functions for translating security headers to and from XML. For instance, the function genX509SecurityHeader takes a certificate, an encrypted key, a timestamp, and a signature and generates the corresponding XML security header; parseX509SecurityHeader does the reverse.

The wssecurity.fs module consists of 538 lines of F# code. It does not yet support several token types defined in WS-Security, such as Kerberos and SAML tokens.

6 Conclusions

This paper demonstrates a new programming method for developing verified WS-Security protocol implementations. Our implementations rely on a reusable library that implements a significant subset of the web services security specifications. We demonstrate the effectiveness of our method on a detailed example of a WS-Security mutual authentication protocol. We verify a series of security properties, and discover some vulnerabilities. Verification depends on our custom optimizing compiler from a subset of F# into the pi calculus, and on ProVerif, a resolution-based prover for the pi calculus.

Although the bulk of our code is verified, we assume the correctness of a few core libraries, such as those implementing cryptographic algorithms and networking. The combination of our compiler and ProVerif is effective, but in case of failure the user does need to interpret rather low-level error messages in source language terms.

In future, we aim to improve the usability of our tools, and to extend our work to more complicated protocols and protocol compositions.

Acknowledgements Stephen Tse, co-author of our previous paper [7], participated in the design of fs2pv, and completed its original implementation, during his internship at Microsoft Research.

Appendix

This appendix presents and describes the protocol messages for the X.509 mutual authentication protocol of Section 3.

Symbolic Messages The listing X509MutualAuthMsg1.xml shows the first message as printed out by a symbolic run of the protocol; X509MutualAuthMsg2.xml shows the second message.

In X509MutualAuthMsg1.xml, ts1 is the symbolic timestamp, and req is the serialized request. The message has a security header that contains ts1, an encrypted symmetric key key1, and an XML digital signature for req and ts1. The key key1 is encrypted using the public key certificate for the server; in this message the certificate is issued by Root and has a serial number guid4 and public key PK(rsa_secret3). The XML signature value sv1 is computed as the RSA−SHA1 signature of the element si, which in turn contains the SHA1 hashes of req and ts1. Finally, the body of the message is the request req encrypted under the symmetric key key5.

The second message can be read similarly; the main difference is that the signature includes a new <SignatureConfirmation> element containing the signature value sv1 from the first message.

Concrete Messages The XML messages printed our in concrete runs of the protocol are instances of the symbolic messages, where each symbol expression is replaced by a concrete, string-encoded value.

For instance, the timestamp ts1 is now the concrete XML element

```
<Timestamp Id="Timestamp" xmlns="http://...wss−wssecurity−utility−1.0.xsd">
  <Created>2006−04−27T09:12:17Z</Created>
  <Expires>2006−04−27T09:13:17Z</Expires>
</Timestamp>
```

and the signature value sv1 is now the 172-character base64-encoded string

```
4Bpd7K+2n6eW+brpEwYO9hdwHrcNPOAoK+Bqn4........KCstFrZQ24=
```

References

1. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.
2. 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.
3. Apache Software Foundation. *Apache WSS4J*, 2006. At http://ws.apache.org/wss4j/.

X509MutualAuthMsg1.xml

```
      <Envelope>
        <Header>
          <Security>
ts1  =      <Timestamp Id='Timestamp'>
              <Created>Now1</>
              <Expires>PlusOneMinute</></>
            <BinarySecurityToken EncodingType='Base64Binary' ValueType='X509v3'
                                 Id='X509Token-client.com'>
             X509(Root,client . com,sha1RSA,PK(rsa_secret1))</>
            <EncryptedKey Id='Encrkey'>
              <EncryptionMethod Algorithm='rsa-1_5' />
              <KeyInfo>
                <SecurityTokenReference>
                  <X509Data>
                    <X509IssuerSerial>
                      <X509IssuerName>Root</>
                      <X509SerialNumber>guid4</></></></></></>
              <CipherData>
                <CipherValue>RSA−Enc{PK(rsa_secret3)}[key5]</></>
              <ReferenceList>
                <DataReference URI='guid6' /></></>
            <Signature>
si1  =        <SignedInfo>
                <CanonicalizationMethod Algorithm='xml-exc-c14n#' />
                <SignatureMethod Algorithm='rsa-sha1' />
                <Reference URI='Body'>
                  <Transforms>
                    <Transform Algorithm='xml-exc-c14n#' /></>
                  <DigestMethod Algorithm='sha1' />
                  <DigestValue>SHA1(
                    <Body Id='Body'>req</>)</></>
                <Reference URI='Timestamp'>
                  <Transforms>
                    <Transform Algorithm='xml-exc-c14n#' /></>
                  <DigestMethod Algorithm='sha1' />
                  <DigestValue>SHA1(ts)</></></>
              <SignatureValue>
sv1  =           RSA−SHA1{rsa_secret1}[si]
              </>
              <KeyInfo>
                <SecurityTokenReference>
                  <Reference URI='X509Token-client.com' ValueType='X509v3' />
                </></></></></>
        <Body Id='Body'>
          <EncryptedData Id='guid6' Type='Content'>
            <EncryptionMethod Algorithm='aes128-cbc' />
            <CipherData>
              <CipherValue>AES−Enc{key5}[
req  =            <Add>
                    <n1>100</>
                    <n2>15.99</></></>]</></></></></>
```

X509MutualAuthMsg2.xml

```
      <Envelope>
        <Header>
          <Security>
ts2  =      <Timestamp Id='Timestamp'>
              <Created>Now2</>
              <Expires>PlusOneMinute</></>
            <BinarySecurityToken EncodingType='Base64Binary' ValueType='X509v3'
                                 Id='X509Token-localhost'>
               X509(Root,localhost , sha1RSA,PK(rsa_secret3)) </>
            <EncryptedKey Id='Encrkey'>
              <EncryptionMethod Algorithm='rsa-1_5' />
              <KeyInfo>
                <SecurityTokenReference>
                  <X509Data>
                    <X509IssuerSerial>
                      <X509IssuerName>Root</>
                      <X509SerialNumber>guid2</></></></></>
              <CipherData>
                <CipherValue>RSA−Enc{PK(rsa_secret1)}[key7]</></>
              <ReferenceList>
                <DataReference URI='guid8' /></></>
            <Signature>
si2  =        <SignedInfo>
                <CanonicalizationMethod Algorithm='xml-exc-c14n#' />
                <SignatureMethod Algorithm='rsa-sha1' />
                <Reference URI='Body'>
                  <Transforms>
                    <Transform Algorithm='xml-exc-c14n#' /></>
                  <DigestMethod Algorithm='sha1' />
                  <DigestValue>SHA1(
                    <Body Id='Body'>resp</>)</></>
                <Reference URI='Timestamp'>
                  <Transforms>
                    <Transform Algorithm='xml-exc-c14n#' /></>
                  <DigestMethod Algorithm='sha1' />
                  <DigestValue>SHA1(ts)</></>
                <Reference URI='SigConf'>
                  <Transforms>
                    <Transform Algorithm='xml-exc-c14n#' /></>
                  <DigestMethod Algorithm='sha1' />
                  <DigestValue>SHA1(
                     <SignatureConfirmation Value='sv1' Id='SigConf' />
                     )</></></>
              <SignatureValue>
sv2  =           RSA−SHA1{rsa_secret3}[si2]
              </>
              <KeyInfo>
                <SecurityTokenReference>
                  <Reference URI='X509Token-localhost' ValueType='X509v3' />
                </></></></></>
        <Body Id='Body'>
          <EncryptedData Id='guid8' Type='Content'>
            <EncryptionMethod Algorithm='aes128-cbc' />
            <CipherData>
              <CipherValue>AES−Enc{key7}[
resp  =         <AddResponse>
                  <n>115.99</></></>]</></></></></>
```

4. K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *2004 ACM Workshop on Secure Web Services*, pages 11–22, October 2004.
5. K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Computer Science*, 340(1):102–153, June 2005.
6. K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004.
7. 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)*, 2006. To appear.
8. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.
9. 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.
10. D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, August 2004. W3C Member Submission.
11. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.
12. D. Eastlake, J. Reagle, et al. *XML Encryption Syntax and Processing*, 2002. W3C Recommendation.
13. D. Eastlake, J. Reagle, D. Solo, et al. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation.
14. 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.
15. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.
16. M. Gudgin et al. *SOAP Version 1.2*, 2003. W3C Recommendation.
17. IBM Corporation. *IBM WebSphere Application Server*, 2006. At http://www.ibm.com/software/websphere/.
18. 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.
19. 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.
20. Microsoft Corporation. *Web Services Enhancements (WSE) 2.0*, 2004. At http://msdn.microsoft.com/webservices/building/wse/default.aspx.
21. Microsoft Corporation. *Windows Communication Foundation (WCF)*, 2006. At http://windowscommunication.net.
22. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
23. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.
24. 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.
25. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
26. D. Syme. *F#*, 2005. At http://research.microsoft.com/fsharp/fsharp.aspx.
27. 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.