Verifying Web Services Security Protocols, Standards, and Implementations

Karthik Bhargavan
Microsoft Research, Cambridge
Microsoft Research - INRIA Joint Centre, Paris

Joint work with C. Fournet and A.D. Gordon
+ R. Puccella, S. Tse, N. Swamy,
+ R. Corin, E. Zalinescu, J.J. Leifer, P-M. Deniérou

Papers & Tools available at http://Securing.WS
A Simple Banking Service

Client (C) \[\rightarrow\] Internet \[\rightarrow\] Web Service (S)

C \(\rightarrow\) S: account\(_C\)
S \(\rightarrow\) C: balance\(_C\)

Balance? $1234.56

serviceUri
A Sample SOAP Request

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
        <BalanceRequest xmlns="http://stockservice.contoso.com">
            <AccountNumber>account_\text{C}</AccountNumber>
        </BalanceRequest>
    </soap:Body>
</soap:Envelope>

- Says: “get me the balance for account_\text{C}”
- XML not meant to be read by humans, so we’ll omit namespace info, trailing brackets, and quote strings...
A Sample SOAP Request

```xml
<Envelope>
  <Body>
    <BalanceRequest>
      <AccountNumber> account_C </>
    </BalanceRequest>
  </Body>
</Envelope>
```

- Says: “get me the balance for $account_C$”
- XML not meant to be read by humans, so we’ll omit namespace info, trailing brackets, and quote strings...that’s better
Threat analysis for our Banking Service

Client (C) \[\xrightarrow{\text{Balance?}}\] Internet \[\xrightarrow{$1234.56\text{ serviceUri}$}\] Web Service (S)

\[C \rightarrow S: \text{account}_C\]
\[S \rightarrow C: \text{balance}_C\]
A Network-based Opponent

1. $O$ may intercept the account number and balance
2. $O$ may send incorrect information to $C$ and $S$
3. O may pretend to be C and get $balance_C$
4. O may pretend to be S and send C incorrect information
Security Goals for our Banking Service

1. Secrecy
   Keep \( account_C \) and \( balance_C \) secret (from everyone except \( C \) and \( S \))

2. Client and Request Authentication
   At \( S \), only accept a request \( account_C \) if sent by customer \( C \)

3. Service and Response Authentication (and Correlation)
   At \( C \), only accept a response \( balance_C \) if it was sent by bank \( S \) in response to the last request sent by \( C \)
Password-based Authentication

\[ C \rightarrow S: account_c, \text{HMAC-SHA1}(pwd_c, account_c) \]
\[ S \rightarrow C: balance_c \]

- Assume \( C \) has a username “C” & password \( pwd_c \) at \( S \)
- Request Authentication
  At \( S \), only accepts an \( account_c \) after checking \( pwd_c \)
- \( C \) MACs \( account_c \) using the shared password
- \( S \) checks the MAC on symbol before responding

(\text{HMAC-SHA1} = \text{Keyed Hash, Message Authentication Code})
A Sample SOAP Request

Says: “get me the balance for $account_C$”

XML not meant to be read by humans, so we’ll omit namespace info, trailing brackets, and quote strings...that’s better
WS-Security: Password-based Auth

<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=1>
        <Username>"C"
        <Nonce>"mTbzQM84RkFqza+Ies/xw=="
        <Created>"2004-09-01T13:31:50Z"
      </UsernameToken>
      <Signature>
        <SignedInfo>
          <SignatureMethod Algorithm=hmac-sha1>
            <Reference URI="#2">
              <DigestValue>"U9sBHidIkVvKA4vZo0gGKxMhA1g="
              <SignatureValue>"8/ohMBZ5JwzYyu+POU/v879R01s="
            </Reference>
          </Reference>
          <KeyInfo>
            <SecurityTokenReference>
              <Reference URI="#1 ValueT"}
A More Typical Secure RPC Protocol

C \rightarrow S: \text{RSA-Encrypt}(pk_S, k),
     \text{AES-Encrypt}(k, account_C),
     \text{AES-Encrypt}(k, \text{HMAC-SHA1}(pwd_C, account_C))

S \rightarrow C: \text{AES-Encrypt}(k, balance_C),
     \text{AES-Encrypt}(k, \text{RSA-SHA1}(sk_S, balance_C))

- Assume C has a username “C” & password $pwd_C$ at S
- Assume S has a public key pair $pk_S, sk_S$

($\text{HMAC-SHA1} = \text{Keyed Hash / Message Authentication Code}$)
($\text{RSA-SHA1} = \text{Asymmetric Signature}$)
($\text{RSA-Encrypt} = \text{Asymmetric Encryption}$)
($\text{AES-Encrypt} = \text{Symmetric Encryption}$)
Verification Goal

• Implementations of Web Services security protocols are difficult to get right
  - Underlying crypto protocols are complex
  - Message formats are verbose and flexible
  - Protocols may be composed in varied ways

• We want to verify web services protocol code for authentication and secrecy goals
Outline

1. Verifying crypto protocol implementations
   - A formal method applied to executable code
   - A simple example protocol
2. Web Services security specifications
   - XML message formats
   - Simple SOAP security protocols
3. Verifying a federated identity protocol
   - A large case study using web services security
4. Secure implementations of session types
Part I
Verifying Cryptographic Protocol Implementations
Cryptographic protocols go wrong

- Historically, one keeps finding simple attacks against protocols
  - even carefully-written, widely-deployed protocols, even a long time after their design & deployment
  - simple = no need to break cryptographic primitives

- Why is it so difficult?
  - concurrency + distribution + cryptography
    - Little control on the runtime environment
  - active attackers
    - Hard to test
  - implicit assumptions and goals
    - Authenticity, secrecy
Hi Bob, love Alice

We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)

1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties of NS threat model using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman, Elgamal invent SSL; holes in v1, v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs

circa 2000: Several FMs for “D&Y problem”: tradeoff between accuracy and approximation

circa 2009: Many FMs now developed; several deliver both accuracy and automation
2009: Job Done?

- Many automated tools for verifying protocol models
  - eg Athena, TAPS, ProVerif, FDR, AVISPA, etc
- Applied to large industrial cryptographic protocols
  - e.g. SSL, IPSEC, Kerberos, Web Services, Cardspace
- Emerging tools and proof techniques for computational crypto models, as used by cryptographers
  - e.g. CryptoVerif
But what of implementation bugs?

- What You Verify is not What You Run
  - Formal models are short, abstract, hand-written
    - They ignore large functional parts of implementations
    - Their formulation is driven by verification techniques
    - It is easy to write models that are safe but dysfunctional (testing & debugging is difficult)
  
- Models and implementations drift apart...
  - Even informal synchronization involves painful code reviews
  - How does one keep track of implementation changes?

- Implementation code is the closest we get to a formal description of most protocols

There is a need to build tools to analyze code
Approach 1: From Model to Code

- Compile protocol models to code
  - CAPSL: Muller and Millen (2001)

- Compile high-level specifications, such as multi-party sessions to provably secure cryptographic protocols
  - s2ml: Bhargavan, Corin, Deniérou, Fournet, Leifer (2009)
Approach 2: From Code to Model

- Extract protocol models from implementations
  - Csur: Goubault-Larrecq and Parrennes (VMCAI’05)
  - fs2pv: Bhargavan, Fournet, Gordon, Tse (CSF’06)

- We extract verifiable pi calculus models from reference protocol implementations in ML
  - fs2pv: a compiler from ML to the applied pi calculus

- We express the attacker model and the intended security properties in terms of ML

- We justify the tool by proving that, for any attack on the ML, there is an attack on the pi calculus model

- We use the ProVerif theorem prover to verify the pi calculus model
One source, three tasks

My protocol

My code

Authz

Other Libraries

Application

Source code with modules and strong interfaces

fs2pv

Symbolic Crypto

Concrete Crypto

ProVerif

Platform (CLR)

Crypto Net

Symbolic verification

Symbolic testing & debugging

Some other implementation

Interoperability (via TCP)
1. Symbolic testing and debugging

Coded in F#, C#...

My code

My protocol

Application

Authz

Other Libraries

We use idealized cryptographic primitives

Symbolic Crypto

Attacker (test)

We model attackers as arbitrary code with access to selected libraries

Platform (CLR)

Crypto Net

We can code any given potential attack as a program
2. Formal verification

We only support a subset of F#.

My code

Authz

Other Libraries

Application

My protocol

We model attackers as arbitrary code with access to selected libraries.

We translate to pi calculus using fs2pv.

Symbolic Crypto

Formal verification considers ALL such attackers.

Attacker (unknown)

Pass: Ok for all attackers, or No + potential attack trace.
3. Concrete testing & interop

We test that our code produces and consumes the same messages as another implementation.

We only change our implementation of cryptographic primitives.

We can also run attacks to test other implementations.

Coded in C#, F#...

My code

Authz

Other Libraries

My protocol

Application

Concrete Crypto

Crypto Net

Platform (CLR)

Some Other Implementation

Interoperability (via TCP)
F# is a dialect of ML running on the CLR developed by Don Syme at MSR Cambridge
http://research.microsoft.com/fsharp
“Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET.”

Suitable for protocol programming, and model extraction
- Simple formal semantics
- Modular programming based on typed interfaces
- Algebraic data types with pattern matching are useful for symbolic crypto and XML processing

Future versions of our tools may target C# or C
ProVerif is an automated cryptographic protocol verifier developed by Bruno Blanchet (ENS) http://www.di.ens.fr/~blanchet/crypto-eng.html

What it can prove:
- Secrecy, authenticity (correspondence properties)
- Equivalences (e.g. protection of weak secrets)
- These properties are undecidable: ProVerif may diverge or fail

How it works:
- Internal representation based on Horn clauses
- Resolution-based algorithm, with clever selection rules
- Attack reconstruction

Automatic, but models may have to be tuned for efficient verification

What do we prove?

- Let $L$ be a set of modules representing the symbolic libraries for cryptography, networking
- Let $P$ be the protocol implementation
- Let $I$ be the interface exported by $L$ and $P$
  - We write $L \cdot P :: I$
- Let $q$ be a desired security property
  - Authentication or secrecy property written as a correspondence between events in a trace
- Then, for all opponent programs $O$ that respect $I$, every trace of $L \cdot P \cdot O$ satisfies $q$
  - Hence, using only the values and functions in $I$, no opponent can break the property $q$ of our implementation
Example
Password-based Authentication
Password-based authentication

\[ A \rightarrow B : \text{HMACSHA1}(nonce, pwd_A | text), \]
\[ \text{RSAEncrypt}(pk_B, nonce), \]
\[ text \]

- A simple, one-message authentication protocol (simplified from a WS-Security sample protocol)
- Two roles
  - client (A) sends some text, along with a MAC
  - server (B) checks authenticity of the text
  - the MAC is keyed using a nonce and a shared password
  - the password is protected from dictionary attacks by encrypting the nonce with the server’s public key
Making and verifying messages

\[ A \rightarrow B : \text{HMACSHA1}(nonce, pwd_A | text),\]
\[ \text{RSAEncrypt}(pk_B, nonce),\]
\[ \text{text} \]

\[
\text{let } \text{mac } n \ \text{pwd } \text{text } = \text{hmacsha1 } n \ (\text{concat } (\text{utf8 } \text{pwd}) \ (\text{utf8 } \text{text}))
\]

\[
\text{let } \text{make text } \text{pke pwd } = \\
\text{let } \text{nonce } = \text{mkNonce}() \ \text{in} \\
(\text{mac nonce pwd text, rsa_encrypt pke nonce, text})
\]

\[
\text{let } \text{verify } (m, en, text) \ \text{skd pwd } = \\
\text{let } \text{nonce } = \text{rsa_decrypt skd en} \ \text{in} \\
\text{if not } (m = \text{mac nonce pwd text}) \ \text{then failwith "bad MAC"}
\]
Coding client and server roles

```ocaml
let address = S "http://server.com/pwmac"
let pwdA = Prins.getPassword(S "A")
let pkB = Prins.getPublicKey(S "B")

type Ev = Send of str | Accept of str

let client text =
  log(Send(text));
  Net.send address (marshall (make text pkB pwdA))

let skB = Prins.getPrivateKey("B")
let server () =
  let m,en,text = unmarshall (Net.accept address) in
  verify (m,en,text) skB pwdA; log(Accept(text))
```

Assuming library functions:
- Prins.getPassword
- Prins.getPublicKey
- Prins.getPrivateKey
- Net.send
- Net.accept

Exporting:
- pkB
- rsa_key
- client
- str
- unit
- server
- unit
Authentication, example theorem

For instance, let system $S$ be our example application code plus symbolic libraries.

Let $I_{pub}$ be the interface

\begin{itemize}
  \item Net.send: \texttt{fun} 2, Net.accept: \texttt{fun} 1,
  \item Crypto.S: \texttt{fun} 1, Crypto.iS: \texttt{fun} 1,
  \item Crypto.base64: \texttt{fun} 1, Crypto.ibase64: \texttt{fun} 1,
  \item Crypto.utf8: \texttt{fun} 1, Crypto.iutf8: \texttt{fun} 1,
  \item Crypto.concat: \texttt{fun} 1, Crypto.iconcat: \texttt{fun} 1,
  \item Crypto.concat3: \texttt{fun} 1, Crypto.iconcat3: \texttt{fun} 1,
  \item Crypto.mkNonce: \texttt{fun} 1, Crypto.mkPassword: \texttt{fun} 1,
  \item Crypto.rsa_keygen: \texttt{fun} 1, Crypto.rsa_pub: \texttt{fun} 1,
  \item Crypto.rsa_encrypt: \texttt{fun} 2, Crypto.rsa_decrypt: \texttt{fun} 2,
  \item Crypto.hmacsha1: \texttt{fun} 2,
  \item pkB: \texttt{val}, client: \texttt{fun} 1, server: \texttt{fun} 1
\end{itemize}

We can verify that $S :: I_{pub}$ is robustly safe for $\texttt{ev:Accept}(x) \Rightarrow \texttt{ev:Send}(x)$
One source, three tasks

- Using the concrete libraries, our client and server run using TCP
  Sending FADC1zZhW3XmgUABgRJj1KjnWyDvEoAAe...

- Using symbolic libraries, we can see through cryptography
  Sending HMACSHA1{nonce3}[’pwd1’ | ’Hi’]
  | RSAEncrypt{PK(rsa_secret2)}[nonce3] | ’Hi’

- Using symbolic libraries, fs2pv generates a ProVerif model
  RESULT Accept(x) ==> Send(x) is true.
The Source Language: a subset of F#

How to justify using ProVerif for proving F# properties?
A First-order Concurrent Core of F#

\[ M, N ::= \]
\[ x \]
\[ a \]
\[ f(M_1, \ldots, M_n) \]

\[ e ::= \]
\[ M \]
\[ \ell \ M_1 \ldots \ M_n \]
\[ \text{fork}(\text{fun}() \to e) \]
\[ \text{match } M \text{ with } (| M_i \to e_i)_{i \in 1..n} \]
\[ \text{let } x = e_1 \text{ in } e_2 \]

\[ d ::= \]
\[ \text{type } s = (| f_i \text{ of } s_{i1} \cdots s_{im_i})_{i \in 1..n} \]
\[ \text{let } x = e \]
\[ \text{let } \ell \ x_1 \ldots x_n = e \quad n > 0 \]

\[ S ::= d_1 \cdots d_n \]
Primitive Types, Functions

- Type: `string`, Constructor: `strcat (^)`
- Type: `bool`, Constructors: `True, False`
- Type: `'a option`, Constructors: `Some, None`
- Type: `'a list`, Constructors: `cons (::), nil ([])`
- Type: `exn`, Constructor: `Fail`
- Functions: `raise, failwith`
- Functions: `equal (=), check, Printf.printf`

Syntactic sugar:

- Types: `tuples, records`
- Language constructs: `do, when`
Primitive Functions: Pi

- **Pi.chan**
  - fresh channel, secret by default
- **Pi.send, Pi.recv**
  - send, receive messages
- **Pi.name**
  - fresh name, secret by default
  - used to model keys, nonces, etc.
- **Pi.log**
  - log an event in an event trace
  - used to specify security properties
  - The opponent is not allowed to use this function
Core Libraries: Net

- Net.send url message
- Net.accept url
- Net.respond message

```scala
module Net
val send: Crypto.str → Crypto.str → unit
val accept: Crypto.str → Crypto.str
val respond: Crypto.str → unit
```
Core Libraries: Crypto

- **Symbolic Types**
  - `str`: strings
  - `bytes`: bitsrings
  - `rsa_key`: asym keys

- **Conversions**
  - `base64`, `utf8`, `concat`

- **Fresh values**
  - `mkPassword`
  - `mkKey`
  - `rsa_keygen`

- **Crypto primitives**
  - `hmacsha1`
  - `rsa_encrypt`
Two implementations of Crypto

```fsharp
module Crypto // concrete code in F#
open System.Security.Cryptography

type bytes = byte[]
type rsa_key = RSA of RSAPrParameters
...
let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
    let x = Bytearray.make 16 in
    rng.GetBytes x; x
...
let hmacsha1 k x =
    new HMACSHA1(k).ComputeHash x
...
let rsa = new RSACryptoServiceProvider()
let rsa_keygen () = ...
let rsa_pub (RSA r) = ...
let rsa_encrypt (RSA r) (v:bytes) = ...
let rsa_decrypt (RSA r) (v:bytes) =
    rsa.ImportParameters(r);
    rsa.Decrypt(v,false)
```

```fsharp
module Crypto // symbolic code in F

type bytes =
    | Name of Pi.name
    | HmacSha1 of bytes * bytes
    | RsaKey of rsa_key
    | RsaEncrypt of rsa_key * bytes
...

and rsa_key = PK of bytes | SK of bytes
...

let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"
...
let hmacsha1 k x = HmacSha1(k,x)
...
let rsa_keygen () = SK (freshbytes "rsa")
let rsa_pub (SK(s)) = PK(s)
let rsa_encrypt s t = RsaEncrypt(s,t)
...
let rsa_decrypt (SK(s)) e = match e with
    | RsaEncrypt(pke,t) when pke = PK(s) -> t
    | _ -> failwith "rsa_decrypt failed"
```
Core Libraries: Prins

- **Types of Principals**
  - principalU
    - username and password
  - principalX
    - X.509 public key certificates

- **Principals database**
  - genX509, genUserPassword
  - Adds new principal entries

- **Retrieving principal data**
  - getPublicKey
  - getPassword, getPrivateKey

- **Compromising principals**
  - leakPassword, leakPrivateKey
  - Triggers Leak event

```plaintext
module Prins
open Crypto

type principalU =
  {user: str;
   password: str;}

type principalX =
  {subject: str;
   cert: bytes;
   pubkey: rsa_key;
   privkey: rsa_key;}

val genUserPassword: str → unit
val genX509: str → unit
valgetX509Cert: str → bytes
val getPublicKey: str → rsa_key

// hidden from opponent:
val getPassword: str → str
val getPrivateKey: str → rsa_key

// exposed to opponent:
// models password and key compromise
val leakPassword: str → str
val leakPrivateKey: str → rsa_key
```
Authentication queries are of the form $\text{ev}:E \Rightarrow \text{ev}:B_1 \lor \cdots \lor \text{ev}:B_n$.

$$C \models \textbf{query } \text{ev}:E \Rightarrow \text{ev}:B_1 \lor \cdots \lor \text{ev}:B_n \text{ if and only if whenever } C \equiv \textbf{event } E \sigma \mid C',$$
there is $C''$ and $i \in 1..n$ such that $C' \equiv \textbf{event } B_i \sigma \mid C''$.

The system $S$ is safe for $q$ if and only if, whenever $S \rightarrow^{\ast} C$, we have $C \models q$.

We write $I \vdash S : I'$ to mean that system $S$
assumes an implementation of interface $I$,
and exports the interface $I'$.

We write $S :: I_{pub}$ to mean that $\text{Prim} \vdash S : I_{pub},I_{priv}$ for some $I_{priv}$.

An opponent $O$ for $S :: I_{pub}$ is any system with $\text{Prim} \setminus \log,I_{pub} \vdash O$.

$S :: I_{pub}$ is robustly safe for $q$ when $S :: I_{pub}$ and $S O$ is safe for $q$ for all opponents $O$. 
Password-based Authentication

Let $S$ be the system consisting of application code and symbolic libraries. Let $I_{pub}$ be the interface

Net.send: **fun** 2, Net.accept: **fun** 1,
Crypto.S: **fun** 1, Crypto.iS: **fun** 1,
Crypto.base64: **fun** 1, Cryptoibase64: **fun** 1,
Crypto.utf8: **fun** 1, Crypto.iutf8: **fun** 1,
Crypto.concat: **fun** 1, Crypto.iconcat: **fun** 1,
Crypto.concat3: **fun** 1, Crypto.iconcat3: **fun** 1,
Crypto.mkNonce: **fun** 1, Crypto.mkPassword: **fun** 1,
Crypto.rsa.keygen: **fun** 1, Crypto.rsa.pub: **fun** 1,
Crypto.rsa.encrypt: **fun** 2, Crypto.rsa.decrypt: **fun** 2,
Crypto.hmacsha1: **fun** 2,
pkB: **val**, client: **fun** 1, server: **fun** 1

We verify that $S :: I_{pub}$ is robustly safe for

\[
\text{ev:ServerRecv}(U,S,\text{req}) \Rightarrow \text{ev:ClientSend}(U,\_\text{,req})||\text{ev:Leak}(U)
\]
Proof Technique
Compiling F# to the Pi Calculus
An Applied Pi Calculus

Source language of the ProVerif theorem prover

Processes of the $\pi$ calculus:

\[
P, Q, R ::= \\
\begin{array}{ll}
\textbf{out } & M(N) \\
\textbf{in } & M(x); P \\
\textbf{!in } & M(x); P \\
\textbf{new } & x; P \\
& P \mid Q \\
& 0 \\
& \textbf{let } \vec{x} = D \textbf{ in } P \textbf{ else } Q \\
& \textbf{begin } L \\
& \textbf{end } L \\
\end{array}
\]

process:
- asynchronous output of $N$ on channel $M$
- input of $x$ from channel $M$ ($x$ has scope $P$)
- replicated input
- fresh generation of name $x$ ($x$ has scope $P$)
- parallel composition of $P$ and $Q$
- inactivity
- bind results of $D$ to $\vec{x}$ in $P$, or else run $Q$
- begin-event labelled $L$
- end-event labelled $L$
Functions as Processes

Milner’s Call-By-Value Continuation-Passing Translation from $\lambda$ to $\pi$:

$$[[x]]k \triangleq \textbf{out } k(x)$$

$$[[\lambda x.e]]k \triangleq \textbf{new } f; \ (\textbf{out } k(f) \mid \textbf{in } f(\langle x, k' \rangle); [[e]]k')$$

$$[[e_1 \ e_2]]k \triangleq \textbf{new } k_1; \ ([[[e_1]]k_1 \mid \textbf{in } k_1(f); \textbf{new } k_2; ([[[e_2]]k_2 \mid \textbf{in } k_2(x); \textbf{out } f(\langle x, k \rangle))])$$

This is the core of model extraction, but additionally we perform transformations to speed up verification.
How to compile a function?

- Our tool specifically targets symbolic verification, with many optimization to help ProVerif converge
  - Complete inlining (anticipating resolution)
    + Dead Code Elimination

- We select a translation for each function
  - Pure non-recursive functions are compiled to term reductions (as supported by ProVerif)
  - Pure recursive functions are compiled to predicate declarations (logic programming)
  - Functions with side-effects are compiled to Pi Processes

- The generated reduction, predicate, or process is declared public or private depending on whether it is in the interface
Compiling a Function

Consider the F# function

```fsharp
let mac nonce pwd text =
    Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

We can translate it as a process

```fsharp
!in(mac, (nonce,pwd,text,k));
    out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

We actually translate `mac` into a ProVerif reduction rule:

```fsharp
reduc mac(nonce,pwd,text) =
    HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```
Protocol Model in ProVerif

Crypto Model

Secrets and Channels

Pi Calculus Process for A

Pi Calculus Process for B

Full Protocol

---

Data

*data concat/2.*
*fun hmacsha1/2.*
*fun pk/1.*
*fun rsaencrypt/2.*
*reduce rsa decrypt (k, rsa encrypt (pk (k), m)) = m.*

Private

*private free pwdA.*
*private free skB.*
*free netChan.*
*free text.*

Let Alice

*let alice = new nonce;*
*event AliceSent (text);*
*out (netChan, concat (hmacsha1 (nonce, concat (pwdA, text)), concat (rsa encrypt (pk (skB), nonce), text))).*

Let Bob

*let in (netChan, x);*
*let concat (mac, concat (enc, text)) = x in*
*let nonce = rsa decrypt (skB, enc) in*
*if mac = hmacsha1 (nonce, concat (pwdA, text)) then*
*event Bob Accepts (text)*
*else 0.*

Process

*process out (netChan, pk (skB)); (alice | bob)*

---

ProVerif’s applied pi-calculus syntax
Soundness of our translation

Theorem 1 (Reflection of Robust Safety)

If $S_0 :: I_{pub}$ and $[S_0 :: I_{pub}]$ is robustly safe for $q$ (in the pi calculus) then $S_0$ is robustly safe for $q$ and $I_{pub}$ (in F#)

- $S_0$ is the series of modules that define our system;
- $I_{pub}$ is the list of values and functions of $S_0$ available to the attacker;
- $q$ is our target security query; and
- $[S_0 :: I_{pub}]$ is the ProVerif script compiled from $S_0$ and $I_{pub}$.

To verify that $S_0$ is robustly safe for $q$ and $I_{pub}$,
1. we run ProVerif on $[S_0 :: I_{pub}]$ with query $q$;
2. if ProVerif completes successfully, we apply Theorem 1.

The proof relies on an operational correspondence between reductions on F configurations and reductions in the pi calculus.
Protocol Verification with ProVerif
Protocol Verification with ProVerif

- If a property is false, ProVerif exhibits the counter-example as an attack
  - E.g. Suppose A does not include *text* in the HMACSHA1

---

**Attack Trace**

```
A more detailed output of the traces is available with
page tracesDisplay = long.
Goal of the attack:
end(BohAccepts, c_12)

out(GetChan, pk[chB])

event(AliceSent(c_11))

out(GetChan, concat(MacHash nonce_2, pk[chB], concat(crypt(pk[chB], nonce_2), text)))
in(GetChan, concat(MacHash nonce_2, pk[chB], concat(crypt(pk[chB], nonce_2), text)))
```

An attack has been found.
```
```
Limitations of fs2pv and ProVerif

- We cannot write higher-order functions
  - Upcoming version will allow higher-order functions as long as they do not appear in interfaces

- We cannot use some builtin types such as refs, arrays
  - We use channels instead to store/retrieve data
  - Upcoming version will allow limited references

- To use additional libraries, such as System.SQL or List, the user must code up symbolic implementations
  - We provide implementations for Crypto, Net, Prins, XML

- We avoid recursive and stateful functions
  - Recursive functions often lead to non-terminating analyses
  - Stateful functions often take a lot of memory to verify
Summary

• It is possible to write realistic, executable code and verify its security goals

• We presented a formal method for verifying crypto protocol implementations written in ML

• We obtain strong security theorems against a powerful opponent model
  - Opponent controls network + some participants
  - Unlimited number of sessions, message size

• For large examples, verification may not scale
  - Verification problem is undecideable, in general
  - Extracted pi calculus model represents whole program
Part II
Web Services Security Specifications and Protocols
A series of new specifications seeks to standardize cryptographic mechanisms for use with web services:

- **WS-Security**: message signatures, encryption
- **WS-Trust**: token issuance, key establishment
- **WS-SecureConversation**: secure sessions
- **WS-SecurityPolicy**: specifying policies

Web services developers can combine these mechanisms to build a custom security protocol:

- in Java: Apache WSS4J, IBM Websphere
- in C#: Microsoft Windows Communication Foundation
  Web Services Enhancements
SOAP Message Security

WS-Security using
XML-Signature & XML-Encryption
WS-Security

- SOAP Envelope/Header/Security header includes:
  - Timestamp
    - To help prevent replay attacks
  - Tokens identifying principals and keys
    - Username token: name and password
    - X509: name and public-key
    - Others including Kerberos tickets, and session keys
  - Signatures
    - Syntax given by XML-DSIG standard
    - Bind together list of message elements, with key derived from a security token
  - Encrypted Keys
    - Syntax given by XML-ENC standard
  - Various message elements may be encrypted
Password-based Authentication

\[ C \rightarrow S: \text{account}_C, \text{HMAC-SHA1}(\text{pwd}_C, \text{account}_C) \]
\[ S \rightarrow C: \text{balance}_C \]

- Assume \( C \) has a username “C” & password \( \text{pwd}_C \) at \( S \)
- Request Authentication
  At \( S \), only accepts an \( \text{account}_C \) after checking \( \text{pwd}_C \)
- \( C \) MACs \( \text{account}_C \) using the shared password
- \( S \) checks the MAC on \( \text{symbol} \) before responding

(HMAC-SHA1 = Keyed Hash, Message Authentication Code)
Example 1: Password-based Auth

UsernameToken assumes both parties know adg’s secret password $p$

Each DigestValue is the sha1 hash of the URI target

$hmacsha1(key, SignedInfo)$ where $key=\text{psha1}(p+nonce+created)$
Example 2: Signing Multiple Elements

To prevent redirections, need to sign To and Action

To prevent replays, need to sign Timestamp and MessageId

Actually, to prevent various XML rewriting attacks, it’s necessary to co-sign other message parts with the body.
Example 3: X.509 Mutual Auth

\[ C \rightarrow S : TS |\]

\[
\text{RSA-SHA1}\{sk_C\}[symbol \mid TS] | \\
\text{RSA-Encrypt}\{pk_S\}[\text{symkey}_1] | \\
\text{AES-Encrypt}\{\text{symkey}_1\}[symbol] \\
\]

\[ S \rightarrow C : \text{RSA-SHA1}\{sk_S\}[quote \mid \text{RSA-SHA1}\{sk_C\}[symbol \mid TS]] | \\
\text{RSA-Encrypt}\{pk_C\}[\text{symkey}_2] | \\
\text{AES-Encrypt}\{\text{symkey}_2\}[quote] \]

- Assume C and S have key-pairs: \((sk_C, pk_C)\) and \((sk_S, pk_S)\) (Assume they have exchanged X.509 public-key certificates)
- Secrecy of messages
  - freshly encrypted under \(pk_S\) and \(pk_C\)
- Request and Response Authentication
- Request-Response Correlation
  - signature of request counter-signed in response
Example 3: Request Message
(symbolic)

```
<Envelope>
  <Header>
    <Security>
      ts1 = <Timestamp Id='Timestamp'>
        <Created>Now1</Created>
        <Expires>PlusOneMinute</Expires>
      </Timestamp>
      <BinarySecurityToken EncodingType='Base64Binary' ValueType='X.509v3'>
        X.509(Root,C,sha1RSA,pkC)
      </BinarySecurityToken>
      <EncryptedKey Id='Encrkey'>
        <EncryptionMethod Algorithm='rsa-1_5' />
        <KeyInfo>
          <SecurityTokenReference>...</SecurityTokenReference>
        </KeyInfo>
        <CipherData>
          <CipherValue>RSA-Encrypt{pk5}[key]</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='guid6' />
        </ReferenceList>
        <Signature>...</Signature>
      </EncryptedKey>
    </Security>
  </Header>
  <Body Id='Body'>
    <EncryptedData Id='guid6' Type='Content'>
      <EncryptionMethod Algorithm='aes128-cbc' />
      <CipherData>
        <CipherValue>AES-Encrypt[key][req = <Symbol>"MSFT"]</CipherValue>
      </CipherData>
    </EncryptedData>
  </Body>
</Envelope>
```

A symbolic representation of an X.509 cert issued by “Root” to “C”

Encrypting fresh symmetric “key”

Encrypting symbol under “key”
Attacks on WS-Security Protocols
Attacks on SOAP security

- Web services vulnerable to same sorts of attacks as websites
  - Buffer overruns, denial of service, SQL injection, etc

- New concerns: flexible, XML-based protocols
  - Web services developers can design and deploy their own application-specific security protocols
  - XML message format open to rewriting attacks
    - Much like classic active attackers (Needham-Schroeder’78)
    - Attacker can redirect, replay, modify, impersonate
    - New: message processing is driven by a flexible, semi-structured message format

- This flexibility is bad news for security
  - We found a range of problems in specs & code, thus motivating research on theory and tools
An attack that uses the XML format

A Signed SOAP Message Before...

<Envelope>
  <Header>
    <Security>
      <UsernameToken Id="2">
        <Username>Alice</Username>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</Nonce>
        <Created>2003-02-04T16:49:45Z</Created>
      </UsernameToken>
      <Signature>
        <SignedInfo>
          <Reference URI="#1"><DigestValue>Ego0...</DigestValue></Reference>
          <SignatureValue>YB9JU/Wr8ykpAlaxCx2KdvjZcc=</SignatureValue>
        </SignedInfo>
        <KeyInfo>
          <SecurityTokenReference><Reference URI="#2"/></SecurityTokenReference>
        </KeyInfo>
      </Signature>
    </Security>
  </Header>
  <Body Id="1">
    <TransferFunds>
      <beneficiary>Bob</beneficiary>
      <amount>1000</amount>
    </TransferFunds>
  </Body>
</Envelope>

Message to bank’s web service says: “Transfer $1000 to Bob, signed Alice”

Bank can verify the signature has been computed using key derived from Alice’s secret password.
and After an XML Rewriting Attack

Charlie has intercepted and rewritten this message

The indirect signature of the body, now hidden in **BogusHeader**, may still appear valid

Although Alice’s password has not been broken, the message now reads “Transfer $5000 to Charlie, signed Alice”
Why does the attack work?

- The SOAP message format is flexible, with optional headers
- A valid XML-Signature is not necessarily a secure WS-Security message signature
  - More checks are needed in the WS-Security implementation

- Implementing standards is tricky
  - An implementation must be willing to accept messages it will never send, for interoperability
  - It must implement a range of algorithms, one of which is dynamically chosen based on the incoming message
  - It must carefully correlate checks in different modules
Unsigned Message Timestamps

Alter and replay envelopes to confuse participants

From: Alice
To: Bookshop
(signed by Alice)

Sent: Monday
From: Alice
To: Bank
Action: “Pay Charlie $20”
(signed by Alice)

Sent: Tuesday
From: Alice
To: Bank
(signed by Alice)

Sent: Wednesday
From: Alice
To: Bookshop
(signed by Alice)

Someone on the net
(Charlie?)
Encrypt or Sign First?

Should the client sign before encrypting or encrypt before signing? Both are allowed by the specifications. Both can be incorrect depending on the rest of the protocol.

Take credit for someone else’s data
A Password Decryption Attack

<Envelope>
  <Header>...
  <Id> aabbcczTy...</Id>
  <Security>
    <UsernameToken>
      <User>Alice</User>
      <Password>
        <EncryptedData>
          <CipherValue>uVx..</CipherValue>
        </EncryptedData>
      </Password>
    </UsernameToken>
  </Security>
  <Body>...</Body>
</Envelope>

<Envelope>
  <Header>...
  <Id> aabbcczTy...</Id>
  <Security>
    <UsernameToken>
      <User>Alice</User>
      <Password>
        <EncryptedData>
          <CipherValue>uVx..</CipherValue>
        </EncryptedData>
      </Password>
    </UsernameToken>
  </Security>
  <Body>...</Body>
</Envelope>

Alice’s laptop

Someone on the net (Charlie?)

Alice’s bookshop (Web Service)
Verifying WS-Security Protocol Implementations
A Verified WS-Security Library

Client Code  Server Code

XML-Encryption  XML-DSIG  WS-Security

SOAP  WS-Addressing

User Code

Web Services Libraries

Crypto  Net  Prins

Core Libraries

Cryptographic algorithms (RSA, AES)

Networking Protocols (TCP, HTTP)

Platform (CLR/Windows)

 Principals and Credentials (X.509 certificates, passwords)
Verifying WS-Security Code

1. Replace core libraries and platform with modules implementing a symbolic Dolev-Yao Abstraction

2. fs2pv

3. Verify

Fails security goals, here’s an attack!

Proverif

Provably satisfies security goals
Experimental results

• We coded and verified a series of protocols and libraries
  - An implementation of Otway-Rees
  - Libraries for principals + realistic attacker models
  - Libraries for Web Services Security standards
  - A series of Web Services sample protocols

• We tested interoperability with other implementations of web services protocols (WSE, WCF)
  - We can use our command-line client
    + client application code in C#
    + an IIS/WSE web server
  - We can register an IIS/F# SOAP filter for our server
    + client application code in C# using WSE
# Experimental results

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Implementation</th>
<th>LOCs</th>
<th>messages</th>
<th>bytes</th>
<th>symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>password-based MAC</td>
<td></td>
<td>38</td>
<td>1</td>
<td>208</td>
<td>16</td>
</tr>
<tr>
<td>password-based MAC variant</td>
<td></td>
<td>75</td>
<td>1</td>
<td>238</td>
<td>21</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td></td>
<td>148</td>
<td>4</td>
<td>74; 140; 134; 68</td>
<td>24; 40; 20; 11</td>
</tr>
<tr>
<td>WS password signing</td>
<td></td>
<td>85</td>
<td>1</td>
<td>3835</td>
<td>394</td>
</tr>
<tr>
<td>WS X.509 signing</td>
<td></td>
<td>85</td>
<td>1</td>
<td>4650</td>
<td>389</td>
</tr>
<tr>
<td>WS password-based MAC</td>
<td></td>
<td>85</td>
<td>1</td>
<td>6206</td>
<td>486</td>
</tr>
<tr>
<td>WS request-response</td>
<td></td>
<td>149</td>
<td>2</td>
<td>6206; 3187</td>
<td>486; 542</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Security Goals</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>queries</td>
<td>secrecy</td>
</tr>
<tr>
<td>password-based MAC</td>
<td>4</td>
<td>weak pwd</td>
</tr>
<tr>
<td>password-based MAC variant</td>
<td>5</td>
<td>pwd</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>16</td>
<td>key</td>
</tr>
<tr>
<td>WS password signing</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>WS X.509 signing</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>WS password-based MAC</td>
<td>3</td>
<td>weak pwd</td>
</tr>
<tr>
<td>WS request-response</td>
<td>15</td>
<td>no</td>
</tr>
</tbody>
</table>
Part III
Federated Identity for Web Applications
Windows Cardspace using WS-Trust & SAML
Federated Identity-Management

- Many (new) identity-management protocols for the web
  - (Single Sign-on) Passport, Liberty, Shibboleth, SAML, OpenID, InfoCard
  - Implementations becoming widely available
    Windows Cardspace (Vista)
- Potentially, these will soon be the crypto protocols most actively used by web users, from blogs to banks.
- Some models and verification results
- Can we verify their implementations?
InfoCard: Information Card Card Profile v1.0

Principal identities and protocol configured by policies and card database
Sample Configurations

- **Self-issued card (created by user)**
  - No IP, token is issued by client (Cardspace)
  - Token associated with asymmetric key-pair
  - All messages mac-ed and encrypted (WS-Security)

- **Managed card (issued by IP to user)**
  - User authenticates with username-password
  - IP issues token associated with symmetric key (encrypted for RP) and a user pseudonym
  - All messages mac-ed and encrypted (WS-Security)
    - Or: C-IP exchanges over TLS, C-RP over WS-Security

- **Many more configurations possible**
Authentication Goal [A1]

- **IP authenticates U before issuing token**

- If IP issues a token that contains the secret card data of U and is meant for use at RP
  - then U must have selected this card and IP, and approved its use at RP.

- **Protocol Design**: IP requires that all token requests be authenticated using U’s credential
  - TLS: Request contains U’s username and password
  - WS-Security: Request is XML-signed using a key generated from U’s password, or using U’s private key
Authentication Goal [A2]

• **RP authenticates U’s request (through IP)**

• If RP accepts a message with a token issued by IP that contains the secret card data of U
  – then U must have selected the card and IP, and approved its use at RP,
  – and IP must have issued the token for the card,
  – and U must have approved the token,
  – and C must have sent the message to RP.

• **Protocol Design:** RP requires that the token is authenticated by IP and that the message is authenticated using the token
  – Token is XML-signed using IP’s private key
  – Message is XML-signed using a fresh symmetric key, and signature is counter-signed using issued token key
Authentication Goal [A3]

• *C authenticates RP’s response*

• If C accepts a message from RP
  – then RP must have sent this message in response to C’s request message.

• Protocol Design: C requires that the response message is authenticated by RP and correlated with the request
  – Message is XML-signed using the same symmetric key as request
  – Optionally, the signature value of the request is echoed and signed in the response
Secrecy Goal [S1]

- **U’s data is released only to RPs chosen by U**

- If the attacker obtains the secret card data of U
  - then U must have selected the card and IP, and approved its use at a **compromised RP**,  
  - and IP must have issued a token for the card,  
  - and U must have approved the token.

- **Protocol Design**: IP authenticates U and then encrypts the token for RP  
  - If token is not specialized to one RP, then the token is sent in the clear
Secrecy Goal [S2]

• **RPs cannot reconstruct U’s browsing history**

• Two colluding RP’s cannot correlate use of a card
  – A protocol run where U presents the same card to RP and RP’ is observationally equivalent to one where U presents different cards to them, even if RP and RP’ are compromised.

• **Protocol Design:** IP computes a pseudonym for U and inserts it into each issued token
  – the pseudonym is specialized to the receiving RP
  – two RPs get tokens with different pseudonyms.
Secrecy Goal [S3]

- *IP only knows U’s browsing history if U tells it*

- The IP cannot discover which RP the user U is interacting with, unless U requests a token with limited scope

- *Protocol Design*: The token request contains no information about RP if the token scope is unlimited
  - C computes the pseudonym in this case and sends it to IP
### Federated Identity using SAML

#### WS-Trust

**Token Issuance**

- **Issuer issues SAML token for C to use at S**
- **Token contains key encrypted for S**
- **Separately, issuer provides key encrypted for C**
- **Issued token can be used for further WS-Trust exchanges or directly for message security**

#### SAML Token Issuance

<table>
<thead>
<tr>
<th>Action</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>C → I</td>
<td>$TS \mid \text{RSA-SHA1}{sk_C}[\text{RST}(S) \mid TS]$</td>
</tr>
<tr>
<td></td>
<td>$\text{token} = \text{SAML}(C, S, \text{RSA-Encrypt}{pk_S}[key])$</td>
</tr>
<tr>
<td>S → C</td>
<td>$\text{RSA-SHA1}{sk_S}[\text{RSTR}(\text{token}) \mid$</td>
</tr>
<tr>
<td></td>
<td>$\text{RSA-SHA1}{sk_C}[\text{RST}(S) \mid TS]] \mid$</td>
</tr>
<tr>
<td></td>
<td>$\text{RSA-Encrypt}{pk_C}[key]$</td>
</tr>
</tbody>
</table>

#### Token Usage

<table>
<thead>
<tr>
<th>Action</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>C → S</td>
<td>$\text{token, AES-Encrypt}{key}[msg_1]$</td>
</tr>
<tr>
<td>S → C</td>
<td>$\text{token, AES-Encrypt}{key}[msg_2]$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Protocol Narration (Self Issued Card)

Initially | C has: $\text{Card(cardId, claims}_U, PK(k_{RP})$ | RP has: $k_{RP}$
---|---|---
C: Request ($RP, M_{req}$) | C receives an application request
U: Select InfoCard (cardId, C, RP, claim-ty_RP) | User selects card
C(IP): Issue Token ($U, cardId, claims_U, RP, display-tok$) | C generates a self-issued token
U: $U: \text{Approve Token (display-tok)}$ | User approves token

C: generate fresh $k, \eta_1, \eta_2, (k_{proof}, PK(k_{proof}))$
C $\rightarrow$ RP: let $M_{ek} = \text{RSAEnc}(PK(k_{RP}), k)$ in
let $k_{sig} = \text{PSHA1}(k, \eta_1)$ in
let $k_{enc} = \text{PSHA1}(k, \eta_2)$ in
let $ppid_{cardId,RP} = H_4(\text{cardId, RP})$ in
let $k_{cardId,RP} = K(\text{cardId, RP})$ in
let $M_{tok} = \text{Assertion(Self, PK(k_{proof}), claims}_U, RP, ppid_{cardId,RP})$ in
let $M_{toksig} = \text{RSASHA1}(k_{cardId,RP}, M_{tok})$ in
let $M_{saml} = \text{SAML}(M_tok, M_{toksig})$ in
let $M_{mac} = \text{HMACSHA1}(k_{sig}, M_{req})$ in
let $M_{proof} = \text{RSASHA1}(k_{proof}, M_{mac})$ in
Service Request ($M_{ek}, \eta_1, \eta_2, PK(k_{cardId,RP}),$
$\text{AESEnc}(k_{enc}, M_{saml}), \text{AESEnc}(k_{enc}, M_{mac}),$
$\text{AESEnc}(k_{enc}, M_{proof}), \text{AESEnc}(k_{enc}, M_{req})$)

<table>
<thead>
<tr>
<th>RP:</th>
<th>Accept Request ($C, claims_U, M_{req}, M_{resp}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP:</td>
<td>generate fresh $\eta_3, \eta_4$</td>
</tr>
</tbody>
</table>
| RP $\rightarrow$ C: | let $k_{sig} = \text{PSHA1}(k, \eta_3)$ in
| | let $k_{enc} = \text{PSHA1}(k, \eta_4)$ in
| | let $M_{mac} = \text{HMACSHA1}(k_{sig}, M_{resp})$ in
| | Service Response ($\eta_3, \eta_4, \text{AESEnc}(k_{enc}, M_{mac}), \text{AESEnc}(k_{enc}, M_{resp})$)

C: Response ($M_{resp}$)

- **Default Protocol of Windows Cardspace**
- **Fresh session key, two nonces, and asymmetric key-pair**
- **Encrypt session key for RP**
- **Derive message signing key**
- **Derive message encryption key**
- **Compute PPID using card identifier, RP’s identity**
- **Compute token signing key using card, RP’s identity**
- **SAML assertion with public key, claims, and PPID**
- **Self-signed SAML assertion**
- **Issued token**
- **Message signature**
- **Endorsing signature proving possession of k_proof**
- **Request, with encrypted token, signatures and body**
- **RP accepts request and authorizes a response**
- **Fresh nonces**
- **Derive message signing key**
- **Derive message encryption key**
- **Message Signature**
- **Service Response, with encrypted signatures and body**
- **C accepts response and sends it to application**
### Protocol Narration for Managed Card

<table>
<thead>
<tr>
<th>Role</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong></td>
<td>Request (RP, M\textsubscript{req})</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>Select InfoCard (cardId, C, RP, \textsubscript{pwd\textsubscript{1,IP}}, claim-ty\textsubscript{IP})</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>Generate fresh k\textsubscript{1}, \eta\textsubscript{1,2,3}</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } M\textsubscript{ek} = RSAEnc(\text{PK}(k\textsubscript{RP}), k\textsubscript{1}) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } k\textsubscript{sig} = PSHA1(k\textsubscript{1}, \eta\textsubscript{1}) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } k\textsubscript{enc} = PSHA1(k\textsubscript{1}, \eta\textsubscript{2}) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } M\textsubscript{req} = RST(cardId, claim-ty\textsubscript{IP}, RP, \eta\textsubscript{ce}) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } M\textsubscript{user} = (U, \text{pwd\textsubscript{1,IP}}) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{let } M\textsubscript{ac} = HMACSHA1(k\textsubscript{sig}, (M\textsubscript{req}, M\textsubscript{user})) in</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>\text{Request Token (M\textsubscript{ek}, \eta\textsubscript{1,2,3},}</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>AESEnc(k\textsubscript{enc}, M\textsubscript{mac}), AESEnc(k\textsubscript{enc}, M\textsubscript{user}),</td>
</tr>
<tr>
<td><strong>C → IP</strong></td>
<td>AESEnc(k\textsubscript{enc}, M\textsubscript{ac}))</td>
</tr>
<tr>
<td><strong>IP</strong></td>
<td>Issue Token (U, cardId, claimsU, RP, display-tok)</td>
</tr>
<tr>
<td><strong>IP</strong></td>
<td>Generate fresh \eta\textsubscript{3,4,5,6,7}</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } k\textsubscript{sig} = PSHA1(k\textsubscript{1}, \eta\textsubscript{3}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } k\textsubscript{enc} = PSHA1(k\textsubscript{2}, \eta\textsubscript{4}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{tokkey} = RSAEnc(\text{PK}(k\textsubscript{RP}), PSHA1(\eta\textsubscript{ce}, \eta\textsubscript{se})) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } ppid\textsubscript{cardId,RP} = H\textsubscript{1}(cardId, RP) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{tok} = \text{assertion}(IP, M\textsubscript{tokkey}, claimsU, RP, ppid\textsubscript{cardId,RP}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{toksig} = RSAEnc(M\textsubscript{tokkey}, k\textsubscript{ آخر}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{ek} = RSAEnc(\text{PK}(k\textsubscript{RP}), k\textsubscript{2}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{encok} = (M\textsubscript{ek}, AESEnc(k\textsubscript{2}, SAML(M\textsubscript{tok}, M\textsubscript{toksig}))) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{r} = RSTB(M\textsubscript{encok}, \eta\textsubscript{se}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{let } M\textsubscript{mac} = HMACSHA1(k\textsubscript{sig}, M\textsubscript{r}) in</td>
</tr>
<tr>
<td><strong>IP → C</strong></td>
<td>\text{Token Response (\eta\textsubscript{3,4,5,6,7}, AESEnc(k\textsubscript{enc}, M\textsubscript{mac}), AESEnc(k\textsubscript{enc}, M\textsubscript{r}))}</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td>Approve Token (display-tok)</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>Generate fresh k\textsubscript{2,3,4,5,6,7}</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } M\textsubscript{ek} = RSAEnc(\text{PK}(k\textsubscript{RP}), k\textsubscript{2}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } k\textsubscript{sig} = PSHA1(k\textsubscript{2}, \eta\textsubscript{5}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } k\textsubscript{enc} = PSHA1(k\textsubscript{2}, \eta\textsubscript{6}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } k\textsubscript{prooff} = PSHA1(\eta\textsubscript{ce}, \eta\textsubscript{se}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } M\textsubscript{mac} = HMACSHA1(k\textsubscript{sig}, M\textsubscript{req}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } k\textsubscript{endorse} = PSHA1(k\textsubscript{prooff}, \eta\textsubscript{7}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{let } M\textsubscript{prooff} = HMACSHA1(k\textsubscript{endorse}, M\textsubscript{mac}) in</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>\text{Service Request (M\textsubscript{ek}, \eta\textsubscript{5,6,7,8}, M\textsubscript{encok},}</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>AESEnc(k\textsubscript{enc}, M\textsubscript{mac}), AESEnc(k\textsubscript{enc}, M\textsubscript{prooff}),</td>
</tr>
<tr>
<td><strong>C → RP</strong></td>
<td>AESEnc(k\textsubscript{enc}, M\textsubscript{req}))</td>
</tr>
<tr>
<td><strong>RP</strong></td>
<td>Accept Request (IP, claimsU, M\textsubscript{req}, M\textsubscript{resp})</td>
</tr>
<tr>
<td><strong>RP</strong></td>
<td>Generate fresh \eta\textsubscript{8,9}</td>
</tr>
<tr>
<td><strong>RP → C</strong></td>
<td>\text{let } k\textsubscript{sig} = PSHA1(k\textsubscript{2}, \eta\textsubscript{8}) in</td>
</tr>
<tr>
<td><strong>RP → C</strong></td>
<td>\text{let } k\textsubscript{enc} = PSHA1(k\textsubscript{2}, \eta\textsubscript{9}) in</td>
</tr>
<tr>
<td><strong>RP → C</strong></td>
<td>\text{let } M\textsubscript{mac} = HMACSHA1(k\textsubscript{sig}, M\textsubscript{resp}) in</td>
</tr>
<tr>
<td><strong>RP → C</strong></td>
<td>\text{Service Response (\eta\textsubscript{8,9}, AESEnc(k\textsubscript{enc}, M\textsubscript{mac}), AESEnc(k\textsubscript{enc}, M\textsubscript{resp}))}</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>\text{Response (M\textsubscript{resp})}</td>
</tr>
</tbody>
</table>

*User receives an application request*

*User selects card and provides password*

*Fresh session key, two nonces, and client entropy for token key*

*Encrypt session key for IP*

*Derive message signing key*

*Derive message encryption key*

*Token request message body*

*User authentication token*

*Message signature*

*Token Request, with encrypted signatures, token and body*

*IP issues token for U to use at RP*

*Fresh nonces, server entropy, token encryption key*

*Derive message signing key*

*Derive message encryption key*

*Compute token key from entropies, encrypt for RP*

*Compute PPID using card master key, RP’s identity*

*SAML assertion with token key, claims, and PPID*

*SAML assertion signed by issuer*

*Token encryption key, encrypted for RP*

*Encrypted issued token*

*Token response message body*

*Message Signature*

*Token Response, with encrypted signature and body*

*User approves token*

*Fresh session key, three nonces*

*Encrypt session key for RP*

*Derive message signing key*

*Derive message encryption key*

*Compute token key from entropies*

*Message signature*

*Derive a signing key from the issued token key*

*Endorsing signature proving possession of token key*

*Service Request, with issued token, encrypted signatures and body*

*RP accepts request and authorizes a response*

*Fresh nonces*

*Derive message signing key*

*Derive message encryption key*

*Message signature*

*Service Response, with encrypted signatures and body*

*C accepts response and sends it to application*
A Reference InfoCard Implementation

Symbolic Libraries

- C.fs
- RP.fs
- IP.fs

Symbolic Debugging

Security Verification

Concrete Libraries

.NET Platform Libraries (Crypto, Networking, Credentials)

- WSSecurity.fs
- XMLenc.fs
- XMLdsig.fs
- WSAddressing.fs
- SOAP.fs

Run

Yes

No

Attack!

Verify

Execution and Interop
**Wssec.fsi (interface)**

```plaintext
let payload2body x = x
let body2payload x = x
let aes_encrypt k x = AESEnc (k, x)
let aes_decrypt k (AESEnc (k, x)) = x
```

**Wssec.fs (symbolic)**

```plaintext
type bytes = Pi.name
type α payload = α

type α enc = RSAEnc of pubkey * α payload | AESEnc of symkey * α payload

type α dsig = RSASHA1 of privkey * α payload | HMACSHA1 of symkey * α payload
```

**Wssec.fs (concrete)**

```plaintext
type bytes = byte array

type α payload = Xml.element list

type α enc = Xml.encdata

type α dsig = Xmldsig.dsig
```

Let statements:

```plaintext
let payload2body xml = Soap.deserializer xml
let body2payload b = Soap.serializer b
let aes_encrypt k x = Xmldenc.aes_encrypt k x
let aes_decrypt k x = Xmldenc.rsa_decrypt k x
```
Vulnerabilities

- If RP’s policy does not require signatures to be encrypted, we find a man-in-the-middle attack that breaks [A3]
  - The attacker can replace U’s token with his own, and recompute the message signature
  - The protocol terminates with inconsistent states at C and RP
- A similar attack is found if IP does not require encrypted signatures
- Fix: Use strong policies requiring encrypted signatures and/or signature confirmation
Vulnerabilities

• If the token is self-issued then the pseudonym does not provide enough privacy, breaking [S2]
  – Pseudonym = Hash (cardId + RP’s X.509 identifier)
  – Unless cardId is a cryptographically random secret, two colluding RP’s can guess it, confirm its value by computing the hash, and correlate the use of the same card at different RPs

• Fix: Use a strong random cardId, keep it secret
# Safety Results

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Crypto Ops</th>
<th>Auth</th>
<th>Secrecry</th>
<th>Verif Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelfIssued-SOAP</td>
<td>1410</td>
<td>9,3</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>38s</td>
</tr>
<tr>
<td>UserPassword-TLS</td>
<td>1426</td>
<td>0,5,17,6</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>24m40s</td>
</tr>
<tr>
<td>UserPassword-SOAP</td>
<td>1429</td>
<td>9,11,17,6</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>20m53s</td>
</tr>
<tr>
<td>UserCertificate-SOAP</td>
<td>1429</td>
<td>13,7,11,6</td>
<td>A1-A3</td>
<td>S1-S3</td>
<td>66m21s</td>
</tr>
<tr>
<td>UserCertificate-SOAP-v</td>
<td>1429</td>
<td>7,5,7,4</td>
<td><strong>A3 Fails!</strong></td>
<td>S1-S3</td>
<td>10s</td>
</tr>
</tbody>
</table>
Levels of Abstraction

1. Replace core + WS-Security libraries + platform with modules implementing a symbolic Dolev-Yao Abstraction

2. fs2pv

3. Verify

Fails security goals, here's an attack!

Provably satisfies security goals

Security goals

Applied pi-calculus script

Proverif
Part IV
Cryptographic Protocol Synthesis for Multi-party Sessions
Multi-party Sessions

• Sessions specify message flows between roles
  – As a graph, with roles as nodes and labelled messages as edges
  – Example of a session with 3 roles:

```
{c,p,w,q} Request {c,p,w,q}
```

```
Forward {c,p,w,q}
```

```
{x} Reply {x}
```

```
Audit {c,p,w}
```

```
{d} Details {d}
```

```
{o} Retry {o}
```

```
Resume {q}
```

• Active area for distributed programming
  – A.k.a. protocols, or contracts, or workflows
  – Pi calculus settings, web services, operating systems
  – Common strategy: type systems enforce protocol compliance
    “If every site program is well-typed, sessions follow their spec”
Compiling Sessions to Crypto Protocols

• We design a small session language.
• From a session description, we generate a secure implementation that will shield our programs from any coalition of remote peers.

1. Well-typed programs play their role.
   - Functional result.
2. A role using our generated implementation can safely assume that remote peers play their role without having to trust their code.
   - Security theorem.
Architecture

Annotated interface

Networking & Cryptography Libraries

Symbolic Annotations (Refinement Types)

Concrete OpenSSL
Concrete .Net
Symbolic

Session Declaration

Annotated interface

Generated ML module

ML Application Code

Formal verification by typing

s2ml
Session Compiler & Protocol Synthesizer

Running Program

Ocaml/F# Compiler

Proof of correctness

ψ χ Φ
Example: Web Service Negotiation

Session Proxy =

role c =
send Request {c,p,w,q};
recv Reply {x}

role p =
recv Request {c,p,w,q} ->
send ( Forward
+ Audit;
  loop:
  recv Details {d} ->
send ( Retry {o}; loop
    + Resume ))

role w =
[...]
Expressiveness

- Directed graphs with loops, branching, value passing
- A role can commit to a value and later reveal it

![Diagram showing a directed graph with nodes labeled c, p, and w, and edges labeled {x} Commit, Ack, Reveal{r}, {x} Write {x}, {x} Rebind {x}, No {x}, {c,p} Init {c,p}, {w} Choice {w}, Contact {c,p,w}]

- A variable can be rebound (on some branches)

- Session participants can be dynamically selected
Local Compliance by ML Typing

Each role is compiled to a role function that expects continuations to drive the session (CPS style).

The continuations are constrained by the generated types.

Session Proxy =

(...)
role p =
  recv Request {c,p,w,q} -> send
  ( Forward + Audit;
    loop:
      recv Details {d} -> send ( Retry {o}; loop + Resume ))
(...)

type var_c = C of principal
type var_p = P of principal
type var_w = W of principal
type var_q = Q of string
type var_x = X of string
type var_d = D of string
type var_o = O of string

(* Proxy function for role p *)
type result_p = string
type msg3 = {
  hRequest : (var_c * var_p * var_w * var_q -> msg4)
}
and msg4 =
  | Forward of (result_p)
  | Audit of (msg6)
and msg6 = { hDetails : (var_d -> msg7)}
and msg7 =
  | Retry of (var_o -> msg6)
  | Resume of (result_p)

val p : principal -> msg3 -> result_p

Source file Proxy.session
Generated file Proxy.mli
Programming With Sessions

- Principal registration
  - Give crypto and network information (public/private keys, IP, ...)
- CPS programming

```ocaml
let rec handler_details n = 
  {hDetails = function D d ->
   if n<2
     then Retry (O "Objections", handler_details (n+1))
   else Resume "Proxy done"
  }
in
Proxy.p "Bob"

val p : principal → msg3 → result_p
```

Sample of a user code file

Generated file `Proxy.mli`
Implementability Conditions

- We want global session integrity, not just local compliance.
- Some sessions are always vulnerable:
  - They can be turned into safe sessions with extra messages.

We detect them and rule them out:
- We can turn them into safe sessions with extra messages.
Ensuring global session integrity

- Minimal number of MACs to check to get session integrity
  - For a given role it corresponds to a MAC from each of the roles involved since its own last involvement
  - Some variables need to be MACed as well.

```
{x} Write {x}
```

```
{x} Rebind {x}
```

```
No {x}
```

```
Need to check a MAC from c that includes x
```

```
Need to check a MAC from p that includes x
```

```
Need to check a MAC from p that includes x
```

```
Need to check a MAC from c that includes x
```
Evidence forwarding

• When a value is secret, only its hash is forwarded
  – All participants still check that the hashes are consistent
Generated Protocol Example (1)

- Different cryptographic protocol for each (abstract) path in session graph
- Protocol for upper path of Proxy example:

```
<table>
<thead>
<tr>
<th>Initially, c, p, w share symmetric keys for mac and enc</th>
</tr>
</thead>
<tbody>
<tr>
<td>c : Assign c, p, w, q</td>
</tr>
<tr>
<td>c : Fresh s</td>
</tr>
<tr>
<td>(Request) c → p : let h₀ = Request(s, 0) in</td>
</tr>
<tr>
<td>let m₀ = h₀</td>
</tr>
<tr>
<td>let a₀ = h₀</td>
</tr>
<tr>
<td>m₀</td>
</tr>
<tr>
<td>(Forward) p → w : let h₁ = Forward(s, 1) in</td>
</tr>
<tr>
<td>let m₁ = h₁</td>
</tr>
<tr>
<td>let a₁ = h₁</td>
</tr>
<tr>
<td>m₁</td>
</tr>
<tr>
<td>w : Assign x</td>
</tr>
<tr>
<td>(Reply) w → c : let h₂ = Reply(s, 2) in</td>
</tr>
<tr>
<td>let a₂ = h₂</td>
</tr>
<tr>
<td>h₂</td>
</tr>
</tbody>
</table>
```
Protocol for lower path of Proxy example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>c, p, w share symmetric keys for mac and enc</td>
</tr>
</tbody>
</table>
| c → p | (Request) Application at c assigns c, p, w, q  
Fresh session identifier s  
Request header  
 Plaintext message from c to p  
Authenticated message from c to p, w  
Formatted Request message |
| p → w | (Audit) Application at w assigns d  
Audit header  
 Plaintext message from p to w  
Authenticated message from p to w  
Formatted Audit message |
| w → p | (Details) Application at w re-assigns d  
Details header  
Authenticated message from w to p  
Formatted Details message |
| p → w | (Retry) Application at p assigns o  
Retry header  
Authenticated message from p to w  
Formatted Retry message |
| w → p | (Details) Application at w assigns p  
Details header  
Authenticated message from w to p  
Formatted Details message |
| p → w | (Resume) Application at p assigns c  
Resume header  
Authenticated message from p to w  
Authenticated message from p to c  
Formatted Resume message |
| w → c | (Reply) Application at w assigns x  
Reply header  
Hashes for MAC verification  
Authenticated message from w to c  
Formatted Reply message |
Formalizing Session Integrity

• A run of a program is a sequence of session events

• Session events:
  – Send\_f (p,s,x) for each label f
    • Records sender p, session s, vars x
  – Recv\_f (p,s,y) for each label f
    • Records recipient p, session s, vars y
  – Leak (p)
    • Records principal p

• A principal p is *compromised* in a given run if Leak(p) is recorded; otherwise it is *compliant*

• The *compliant events* in a run are the events logged by uncompromised principals
Security Theorem

• Let $R$ be a run of a program supporting sessions $\Sigma$
• Let $A$ be the subset of compliant principals in $R$
• Let $R_A$ consisting of the compliant events in $R$
• Then, for every session identifier $s$ in $R_A$, there exists an instance of a session in $\Sigma$ that matches exactly the events of $s$ in $R_A$
Proof technique

• Based on the F7 type checker
  – F# is extended with logical assertions
  – The type system is extended with refinement types
  – F7 type checks a program through 1\textsuperscript{st} order logic proof obligations given to the Z3 SMT solver

• All generated protocols verified automatically
Summary and Conclusions
What is novel about Web Services Security?

- Specifications define mechanisms, not protocols
  - Focus on message formats for interoperability
- Semi-structured, verbose message formats
  - Flexibility in ordering, optional elements
- Protocols embed other protocols
  - WS-Security uses XML-Signature, XML-Encryption
  - WS-Trust embeds Kerberos, TLS, SAML
- Protocols can be composed
  - WS-Trust then WS-Trust then WS-SecureConv
  - Multiple XML signatures, counter-signatures
  - Multi-party sessions
Conclusions

- Designing and implementing web services security protocols is tricky and error-prone
  - Attacks on libraries and user code
- Modelling and verification tools help
  - Automatic model extraction is even better
  - Automatic code generation is sometimes applicable
- Cryptographic protocol verification tools can handle fairly sophisticated XML-based protocols
- A combination of manual and automated proof can yield powerful theorems for flexible multi-party protocols

- Limitations:
  - Our results hold only in our (Dolev-Yao) model
  - Automated proof is still infeasible for some complex protocols
Related Work (Web Services)

• Symbolic (Dolev-Yao) analyses of web services security protocols
  - Pi-calculus models using Proverif
    [Bhargavan, Fournet, Gordon, … POPL’04, CCS’04, SWS’04 &’05]
  - CSP models using FDR
    [Kleiner & Roscoe ARSPA’04, MFPS’05]
  - HLPSL models using AVISPA
    [Backes, Mödersheim, Pfitzmann, Viganò FOSSACS’06]

• Computational analyses and proofs
  - Using the BPW library
    [Backes, Mödersheim, Pfitzmann, Viganò FOSSACS’06]
<table>
<thead>
<tr>
<th>Language</th>
<th>Authors</th>
<th>Year</th>
<th>Tools</th>
<th>Frameworks</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Goubault-Larrecq, Parrennes</td>
<td>2005</td>
<td>Csur</td>
<td>SPASS</td>
<td>FM</td>
</tr>
<tr>
<td>Java</td>
<td>O'Shea</td>
<td>2006</td>
<td>LysaTool</td>
<td>FM</td>
<td>NSL, Otway-Rees (self-written)</td>
</tr>
<tr>
<td>F#</td>
<td>Bhargavan, Fournet, Gordon, Tse, Swamy</td>
<td>2006</td>
<td>FS2PV</td>
<td>PV</td>
<td>FM</td>
</tr>
<tr>
<td>Java</td>
<td>Poll, Schubert</td>
<td>2007</td>
<td>JML</td>
<td>FSA</td>
<td>MIDP-SSH (independent)</td>
</tr>
<tr>
<td>F#</td>
<td>Bhargavan, Corin, Fournet</td>
<td>2007</td>
<td>FS2CV</td>
<td>CV</td>
<td>CM</td>
</tr>
</tbody>
</table>

This table omits work on deriving code from models, and tools to check for insecure configurations of security protocols.
Building a Cryptographic Verification Kit

Reference Implementation

Typed Interface

Protocol

Application

Crypto Library

Network Library

Typed Interface

Computational Crypto

Polytime Adversary

Symbolic Crypto

Active Adversary

Compile

Concrete runs and interop tests over .NET Runtime

fs2cv

Computational crypto proof using CryptoVerif

fs2pv

Symbolic proof or attack trace using ProVerif

f7

Symbolic proof by typing using Z3

One Source Many Tasks
## Scalable Verification by Typing

<table>
<thead>
<tr>
<th>Category</th>
<th>Protocol</th>
<th>F# Code</th>
<th>F7 Interface</th>
<th>Typechecking time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small examples</strong></td>
<td>MAC-based Auth</td>
<td>40 lines</td>
<td>12 lines</td>
<td>2.8s</td>
</tr>
<tr>
<td></td>
<td>Flexible Signatures</td>
<td>167 lines</td>
<td>52 lines</td>
<td>14.6s</td>
</tr>
<tr>
<td><strong>Custom-generated multi-party sessions</strong></td>
<td>4-party Distributed Login</td>
<td>2053 lines</td>
<td>1542 lines</td>
<td>1m43.4s</td>
</tr>
<tr>
<td></td>
<td>3-party Web Service Negotiation</td>
<td>2181 lines</td>
<td>1939 lines</td>
<td>2m34.1s</td>
</tr>
<tr>
<td><strong>Web Services Security</strong></td>
<td>X.509-based XML Signatures</td>
<td>1731 lines</td>
<td>479 lines</td>
<td>2m49.3s</td>
</tr>
<tr>
<td></td>
<td>Password-X.509 Mutual Auth</td>
<td>1791 lines</td>
<td>489 lines</td>
<td>3m29.8s</td>
</tr>
<tr>
<td><strong>Windows Cardspace</strong></td>
<td>UserCertificate-SOAP</td>
<td>1429 lines</td>
<td>309 lines</td>
<td>6m3s</td>
</tr>
</tbody>
</table>

*fs2pv cannot verify*
Open Problems

• How do we relate low-level secrecy and authentication guarantees to high-level access control and authorization policies?

• When can we ignore XML and treat web services security protocols as standard cryptographic protocols?
  - Can we abstract from the message format verifiably?

• How do we verify arbitrary compositions of protocols automatically?
  - An unlimited number of sequential WS-Trust exchanges
  - An arbitrary farm of web services sharing secret keys
  - Multi-party sessions with delegation

• How do we verify third party code written in Java or C?
Reading List

• FS2PV

• Verifying Web Services Security

• Verifying Windows Cardspace

• Protocol Synthesis for Multi-party Sessions
Questions?

All tools and papers available from

http://securing.ws
http://research.microsoft.com/~karthb
http://www.msr-inria.inria.fr/projects/sec/