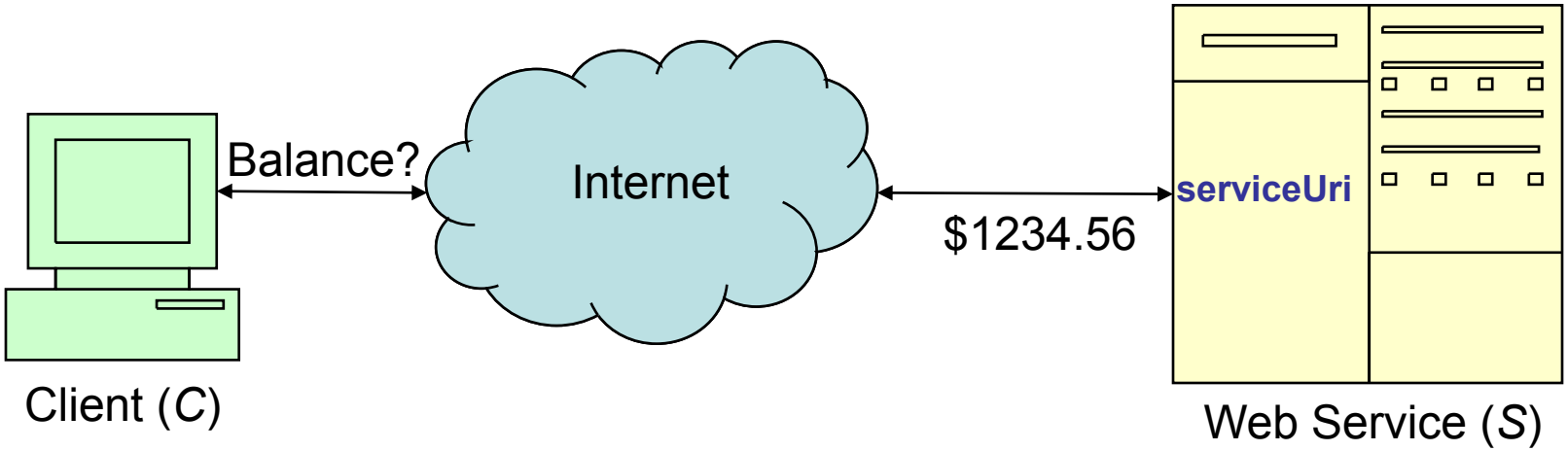
Verifying Web Services Security
Protocols, Standards, and Implementations

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. Deniélou

Papers & Tools available at http://Securing.WS

A Simple Banking Service



$C \rightarrow S$: $account_C$
$S \rightarrow C$: $balance_C$

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_C </AccountNumber>
    </BalanceRequest>
  </soap:Body>
</soap:Envelope>
```

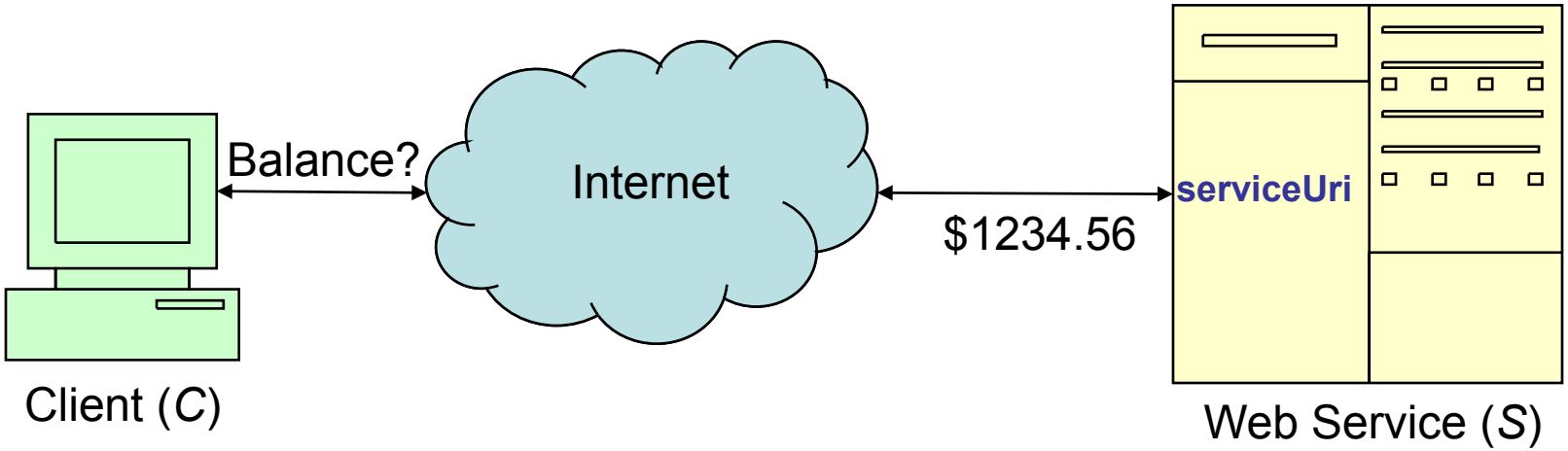
- Says: "get me the balance for $account_C$"
- XML not meant to be read by humans, so we'll omit namespace info, trailing brackets, and quote strings…

A Sample SOAP Request

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

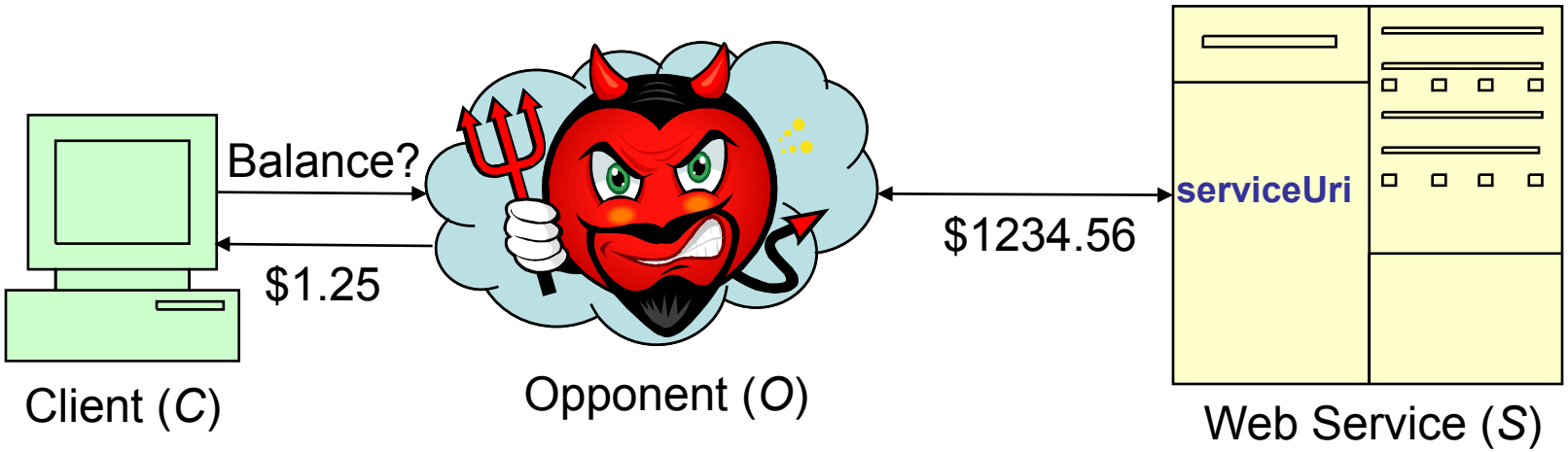
- Says: "get me the balance for $account_C$"
- XML not meant to be read by humans, so we'll omit namespace info, trailing brackets, and quote strings…that's better

Threat analysis for our Banking Service



$C \rightarrow S$: $account_C$
$S \rightarrow C$: $balance_C$

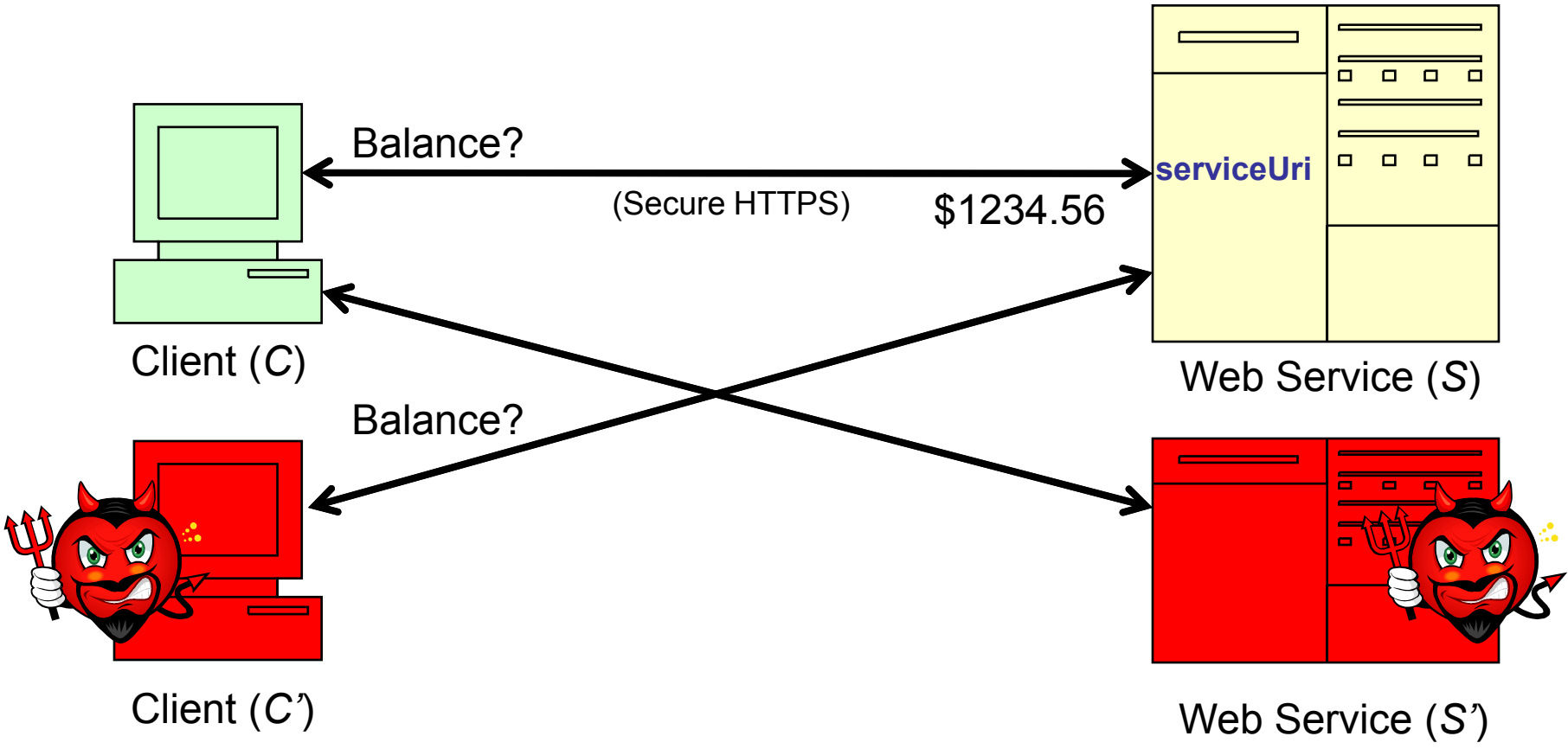
A Network-based Opponent



$C \rightarrow O(S)$: $account_C$
$S \rightarrow O(C)$: $balance_C$
$O(S) \rightarrow C$: $1.25

1. *O* may intercept the account number and balance
2. *O* may send incorrect information to *C* and *S*

Corrupted Clients and Services



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

$C \rightarrow S$: $account_C$
$S \rightarrow C$: $balance_C$

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

$C \rightarrow S$: $account_C$, 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

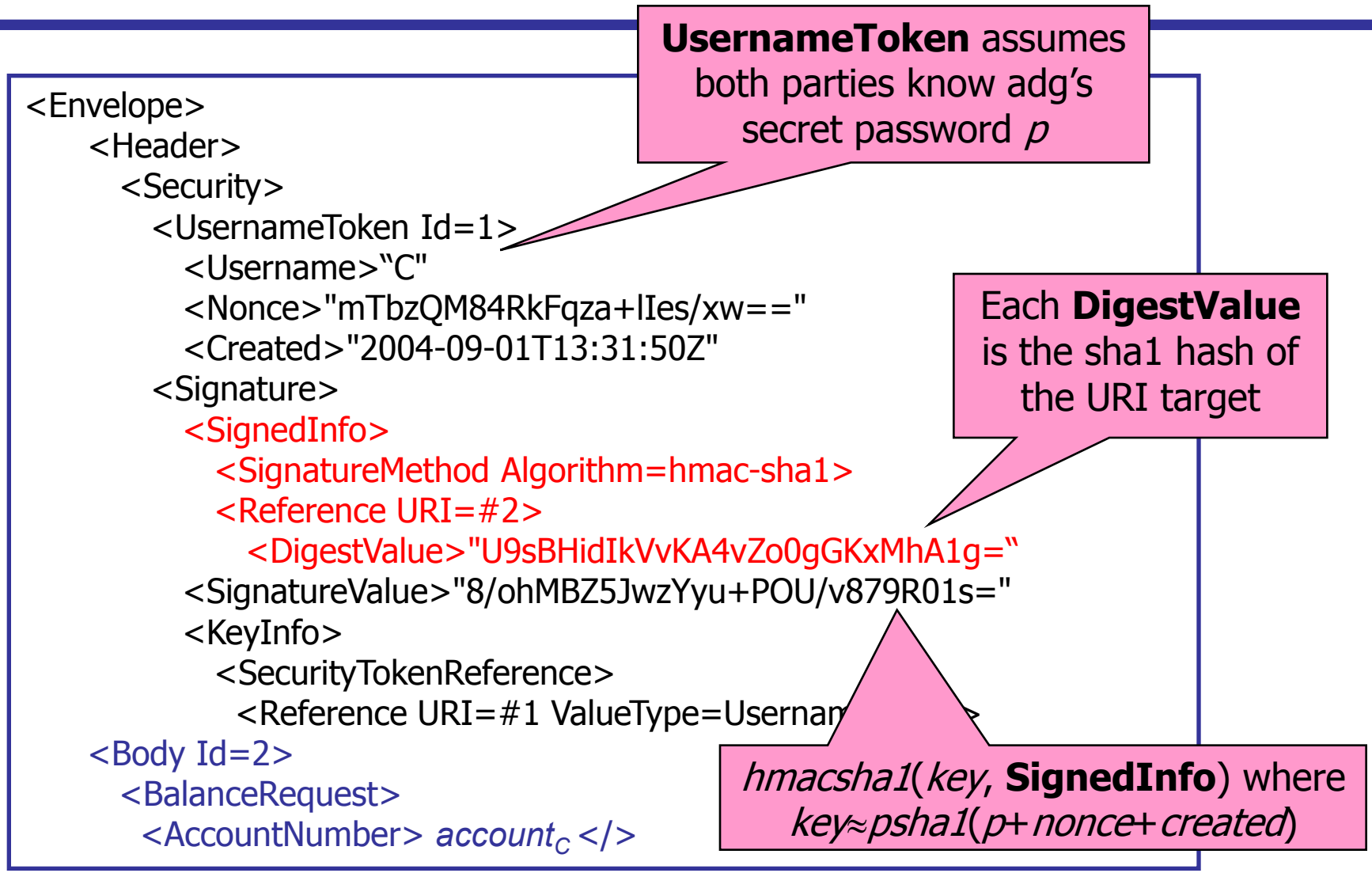
(HMAC-SHA1 = Keyed Hash, Message Authentication Code)

A Sample SOAP Request

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

- 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



A More Typical Secure RPC Protocol

$C \rightarrow S$: RSA-Encrypt(pk_S, k),
AES-Encrypt(k, $account_C$),
AES-Encrypt(k, HMAC-SHA1(pwd_C,$account_C$))
$S \rightarrow C$: AES-Encrypt(k, $balance_C$),
AES-Encrypt(k, 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

(HMAC-SHA1 = Keyed Hash / Message Authentication Code)
(RSA-SHA1 = Asymmetric Signature)
(RSA-Encrypt = Asymmetric Encryption)
(AES-Encrypt = 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

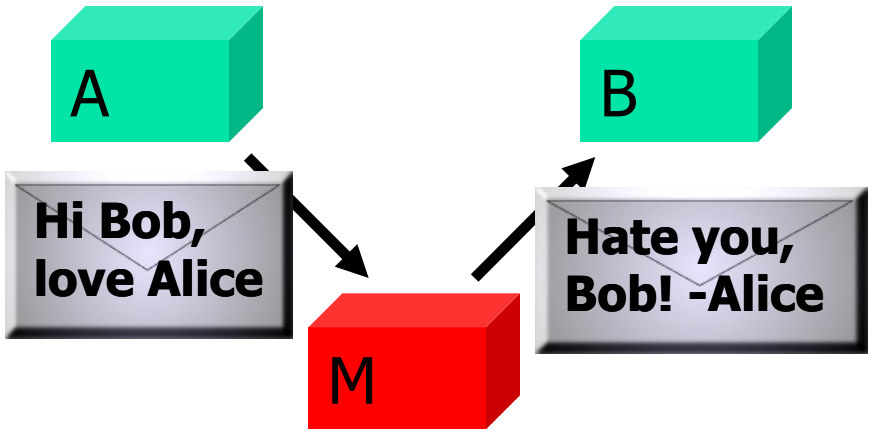
1. Verifying crypto protocol implementations
   - A formal method applied to executable code
   - A simple example protocol
2. Web Services security specifications
   - XML message formats
   - Simple SOAP security protocols
3. Verifying a federated identity protocol
   - A large case study using web 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

Formal Methods for Crypto (1978—2009)



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: Abadi, Anderson, Needham, et al propose various informal "robustness principles"
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

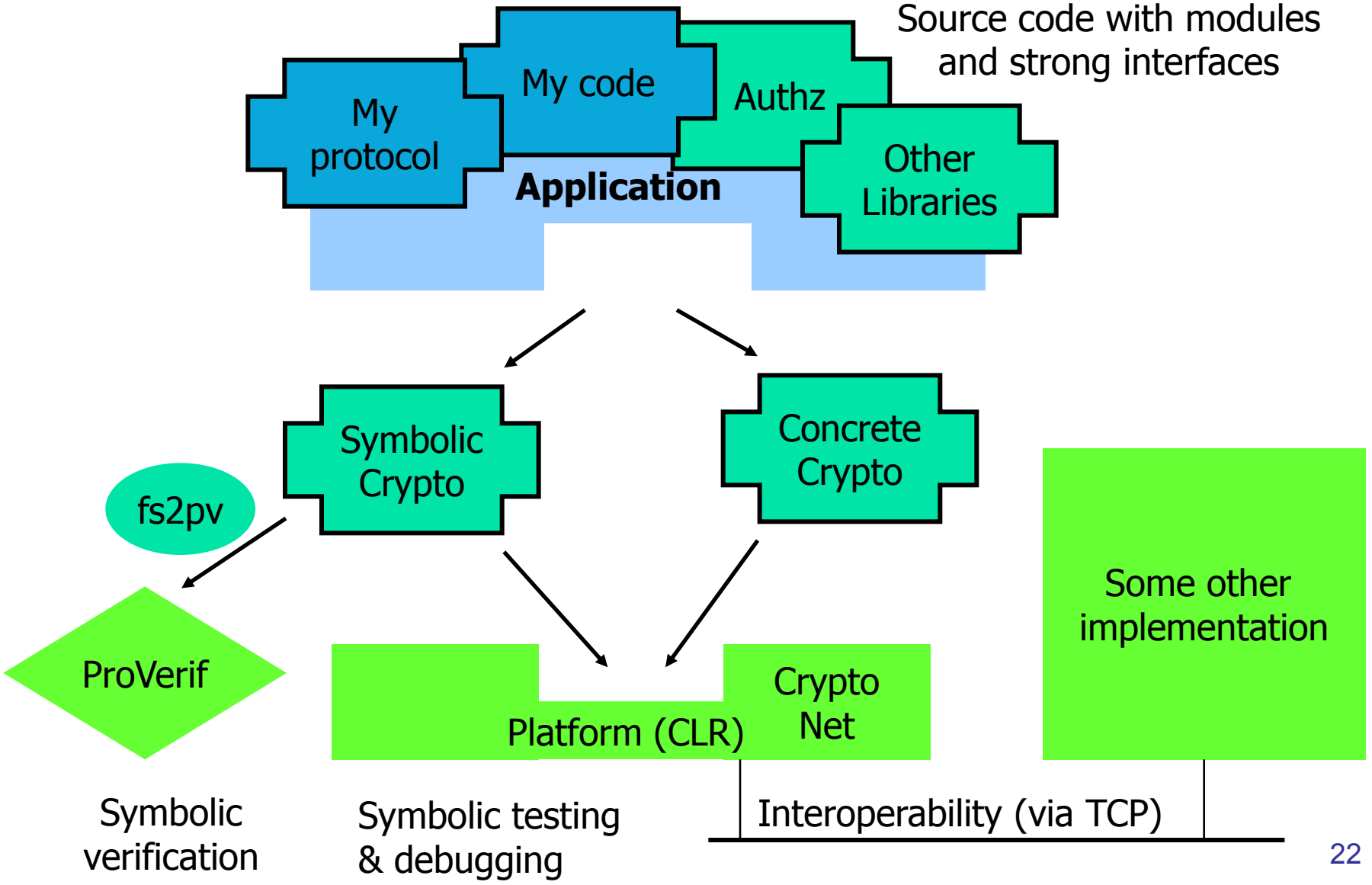
- Compile protocol models to code
  - Strand spaces: Perrig, Song, Phan (2001), Lukell (2003)
  - CAPSL: Muller and Millen (2001)
  - Spi calculus: Lashari (2002), Pozza, Sista, Durante (2004)

- Compile high-level specifications, such as multi-party sessions to provably secure cryptographic protocols
  - s2ml: Bhargavan, Corin, Deniélou, Fournet, Leifer (2009)

Approach 2: From Code to Model

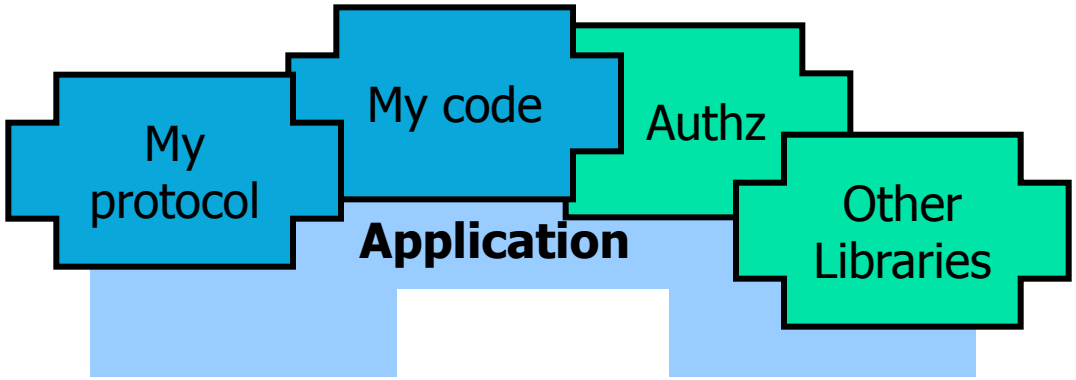
- Extract protocol models from implementations
  - Csur: Goubault-Larrecq and Parrennes (VMCAI'05)
  - fs2pv: Bhargavan, Fournet, Gordon, Tse (CSF'06)
- We extract verifiable pi calculus models from reference protocol implementations in ML
  - fs2pv: a compiler from ML to the applied pi calculus
- We 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

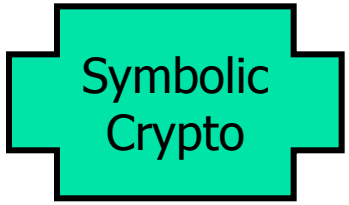


1. Symbolic testing and debugging

Coded in F#, C#...

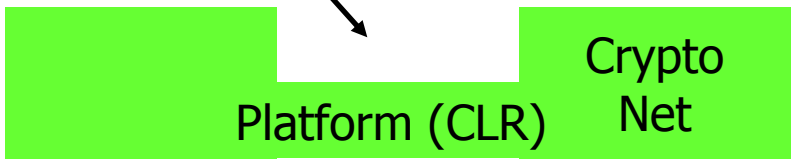


We use idealized cryptographic primitives



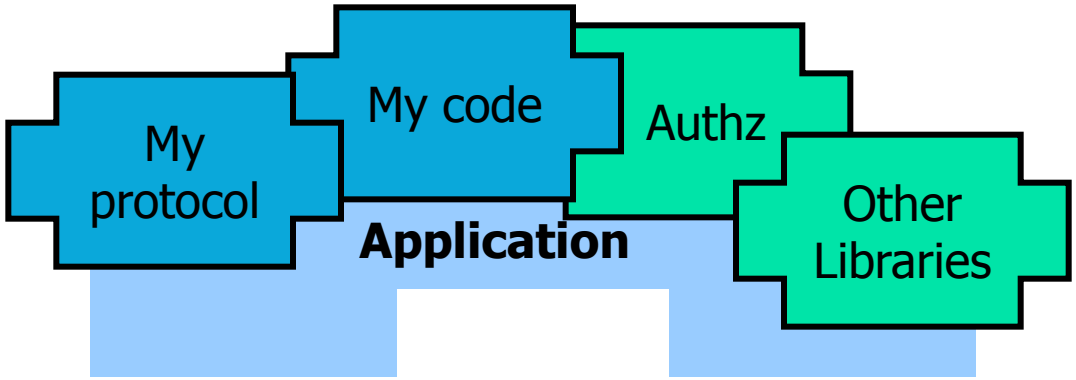
We model attackers as arbitrary code with access to selected libraries

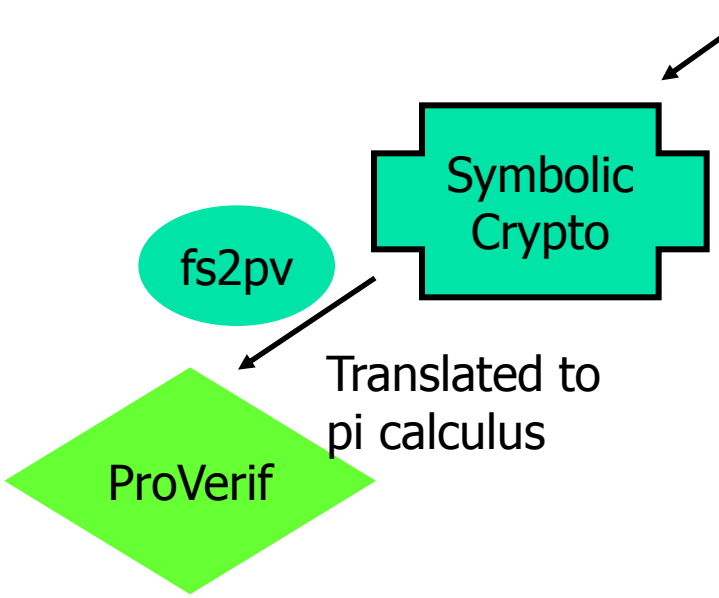
We can code any given potential attack as a program



2. Formal verification

We only support a subset of F#





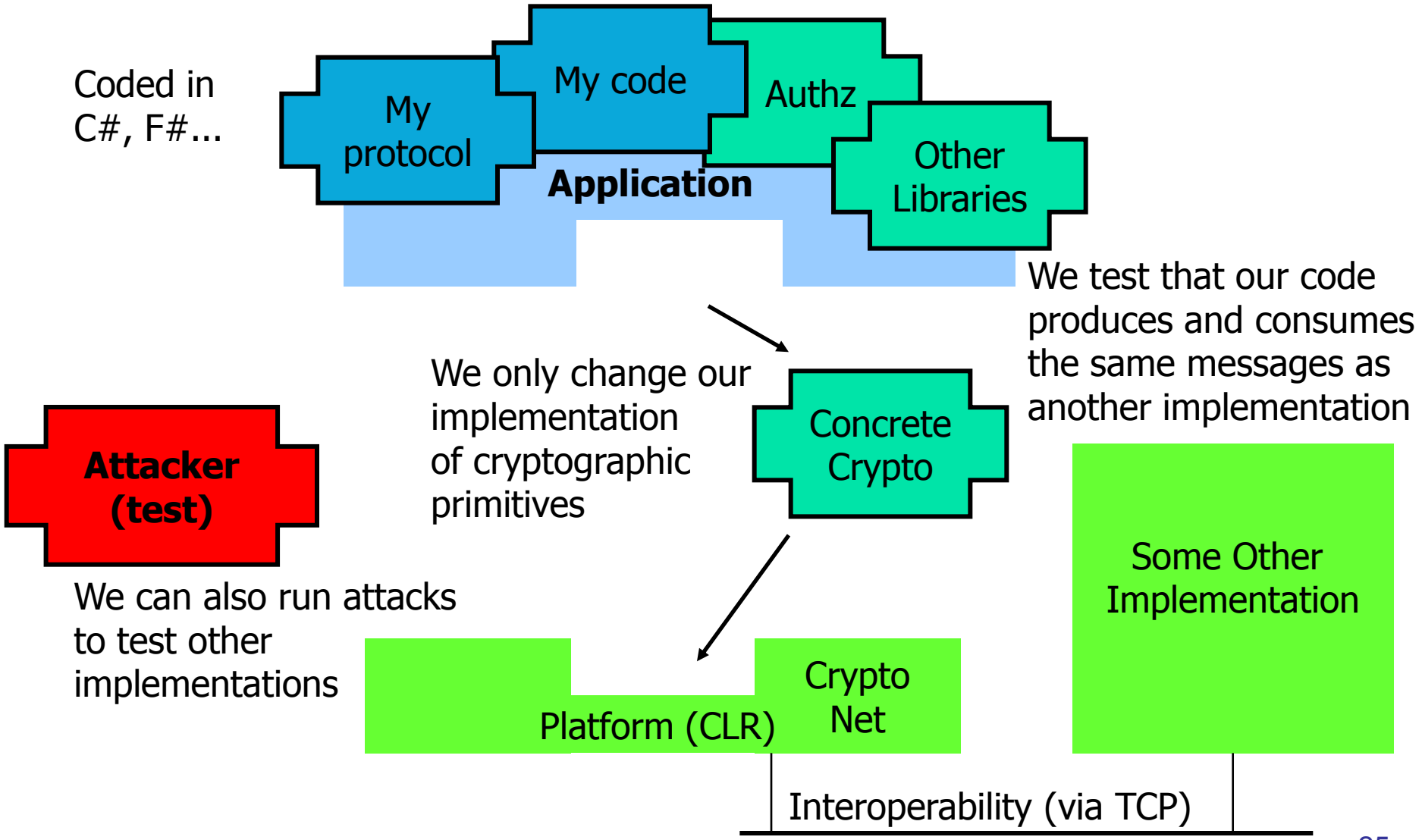
Translated to pi calculus

We model attackers as arbitrary code with access to selected libraries

Formal verification considers ALL such attackers

Pass: Ok for all attackers, or
No + potential attack trace

3. Concrete testing & interop



Source Language: F#

- 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

Target Pi Calculus Verifier: ProVerif

- 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

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In **14th IEEE Computer Security Foundations Workshop (CSFW 2001).** IEEE Computer Society, 2001.

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 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 P 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

$$A \rightarrow B: \quad \begin{array}{l} \mathrm{HMACSHA1}(nonce, pwd_A | text), \\ \mathrm{RSAEncrypt}(pk_B, nonce), \\ text \end{array}$$

- 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

$$A \rightarrow B: \quad \begin{array}{l} \mathrm{HMACSHA1}(nonce, pwd_A | text), \\ \mathrm{RSAEncrypt}(pk_B, nonce), \\ text \end{array}$$

Assuming library functions:
Crypto.concat
Crypto.utf8
Crypto.hmacsha1
Crypto.mkNonce
Crypto.rsa_encrypt
Crypto.rsa_decrypt

```
let mac n pwd text = hmacsha1 n (concat (utf8 pwd) (utf8 text))

let make text pke pwd =
  let nonce = mkNonce() in
  (mac nonce pwd text, rsa_encrypt pke nonce, text)

let verify (m,en,text) skd pwd =
  let nonce = rsa_decrypt skd en in
  if not (m = mac nonce pwd text) then failwith "bad MAC"
```

Coding client and server roles

```
let address = S "http://server.com/pwdmac"
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    unit
```

Authentication, example theorem

For instance, let system S be our example application code plus 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, Crypto.ibase64: **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 can verify that $S :: I_{pub}$ is robustly safe for **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x)

One source, three tasks

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

  Sending FADCIzZhW3XmgUABgRJj1KjnWyDvEoAAe...

- 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#

$M, N ::=$	value
$\quad x$	$\quad$ variable
$\quad a$	$\quad$ name
$\quad f(M_1, \ldots, M_n)$	$\quad$ constructor application
$e ::=$	expression
$\quad M$	$\quad$ value
$\quad \ell\ M_1\ \ldots\ M_n$	$\quad$ function application
$\quad$ fork(**fun**()$\rightarrow e$)	$\quad$ fork a parallel thread
$\quad$ **match** M **with** $(\mid M_i \rightarrow e_i)^{i \in 1..n}$	$\quad$ match: M_i patterns, $n \geq 0$
$\quad$ **let** $x = e_1$ **in** e_2	$\quad$ sequential evaluation
$d ::=$	declaration
$\quad$ **type** $s\ =\ (\mid f_i$ **of** $s_{i1} * \cdots * s_{im_i})^{i \in 1..n}$	$\quad$ datatype declaration
$\quad$ **let** $x\ = e$	$\quad$ value declaration
$\quad$ **let** $\ell\ x_1 \ldots x_n = e \qquad n > 0$	$\quad$ function declaration
$S ::= d_1 \cdots d_n$	system: list of declarations

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

```
type name
val name: string → name

type 'a chan
val chan: unit → 'a chan
val send: 'a chan → 'a → unit
val recv: 'a chan → 'a

type 'a trace
val trace: unit → 'a trace
val log: 'a trace → 'a → unit
```

Core Libraries: Net

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

```
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

```
module Crypto
type str // strings
type bytes // byte arrays
type rsa_key // RSA keys
val S: string → str
val iS: str → string
val base64: bytes → str
val ibase64: str → bytes
val utf8: str → bytes
val iutf8: bytes → str
val concat: bytes → bytes → bytes
val concat3: bytes → bytes → bytes → bytes
val iconcat: bytes → bytes * bytes
val iconcat3: bytes → bytes * bytes * bytes

val mkNonce: unit → bytes
val mkPassword: unit → str
val mkKey: unit → bytes
val rsa_keygen: unit → rsa_key

val hmacsha1: bytes → bytes → bytes
val rsa_pub: rsa_key → rsa_key
val rsa_encrypt: rsa_key → bytes → bytes
val rsa_decrypt: rsa_key → bytes → bytes
val aes_encrypt: bytes → bytes → bytes
val aes_decrypt: bytes → bytes → bytes
```

Two implementations of Crypto

```
module Crypto // concrete code in F#
open System.Security.Cryptography
type bytes = byte[]
type rsa_key = RSA of RSAParameters
...
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)
```

```
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

```
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
val getX509Cert: 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, Formally

Authentication queries are of the form **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n.

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

The system S is safe for q if and only if, whenever $S \rightarrow_{\equiv}^{*} C$, 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 $\mathsf{Prim} \vdash S : I_{pub}, I_{priv}$ for some I_{priv}.

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

$S :: I_{pub}$ 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, Crypto.ibase64: **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

ev:ServerRecv(U,S,req) $\Rightarrow$ **ev**:ClientSend(U,_,req)||**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 π calculus:

$P, Q, R ::=$	process
out $M(N)$	asynchronous output of N on channel M
in $M(x); P$	input of x from channel M (x has scope P)
!in $M(x); P$	replicated input
new $x; P$	fresh generation of name x (x has scope P)
$P \mid Q$	parallel composition of P and Q
0	inactivity
let $\vec{x} = D$ **in** P **else** Q	bind results of D to $\vec{x}$ in P, or else run Q
begin L	begin-event labelled L
end L	end-event labelled L

Functions as Processes

Milner's Call-By-Value Continuation-Passing Translation from λ to π:

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

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

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

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

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

We can translate it as a process

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

We actually translate mac into a ProVerif reduction rule:

```
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 concat/2.
fun hmacsha1/2.
fun pk/1.
fun rsaencrypt/2.
reduc rsadecrypt(k,rsaencrypt(pk(k),m)) = m.


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


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

let bob =
    in (netChan, x);
    let concat(mac,concat(enc,text)) = x in
    let nonce = rsadecrypt(skB,enc) in
    if mac = hmacsha1(nonce,concat(pwdA,text)) then
       event BobAccepts(text)
    else 0.

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}.

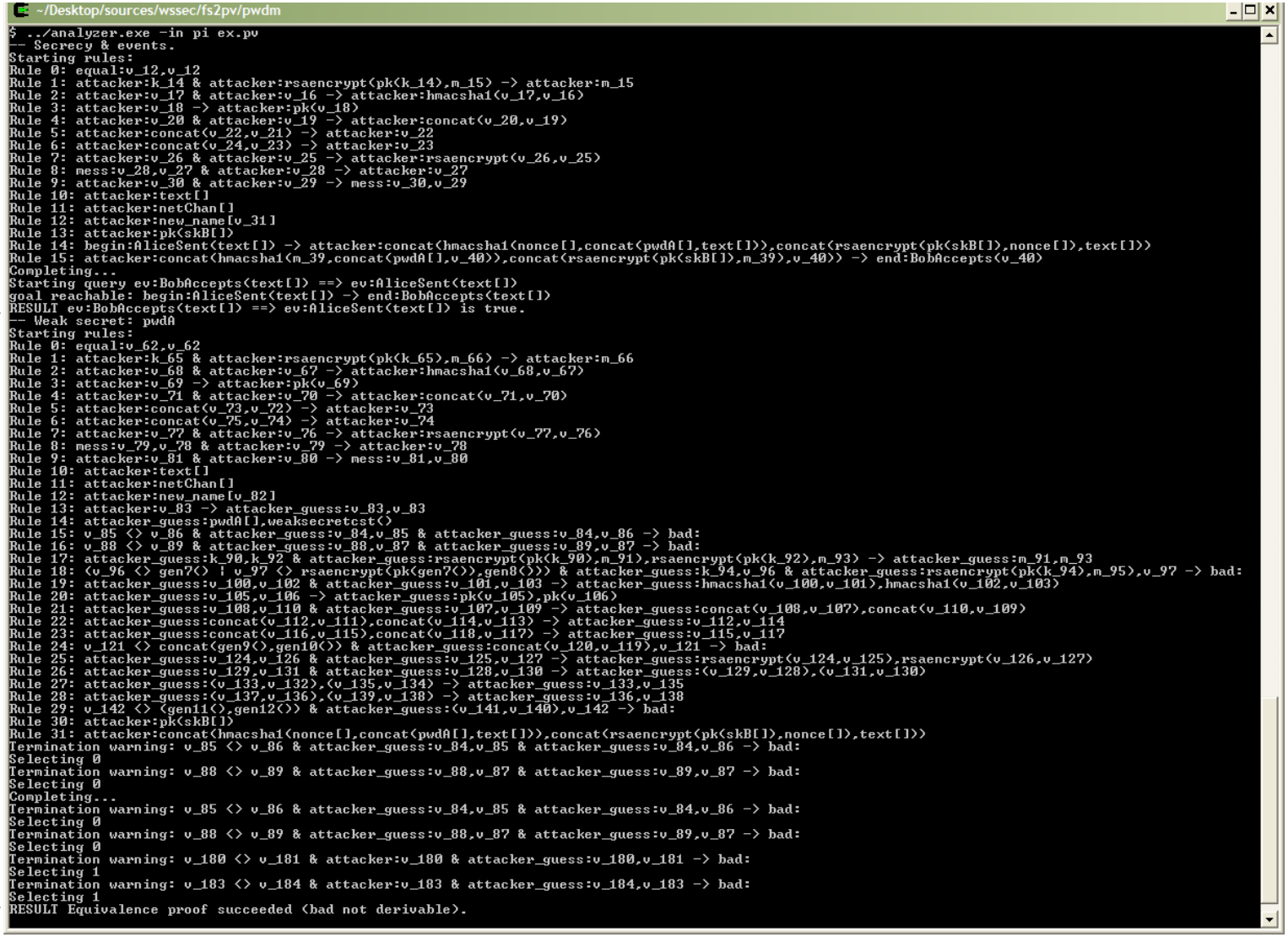
To verify that S_0 is robustly safe for q and I_{pub},
1. we run ProVerif on $[\![S_0 :: I_{pub}]\!]$ with query q;
2. if ProVerif completes successfully, we apply Theorem 1.

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

Protocol Verification with ProVerif

Message Authentication →

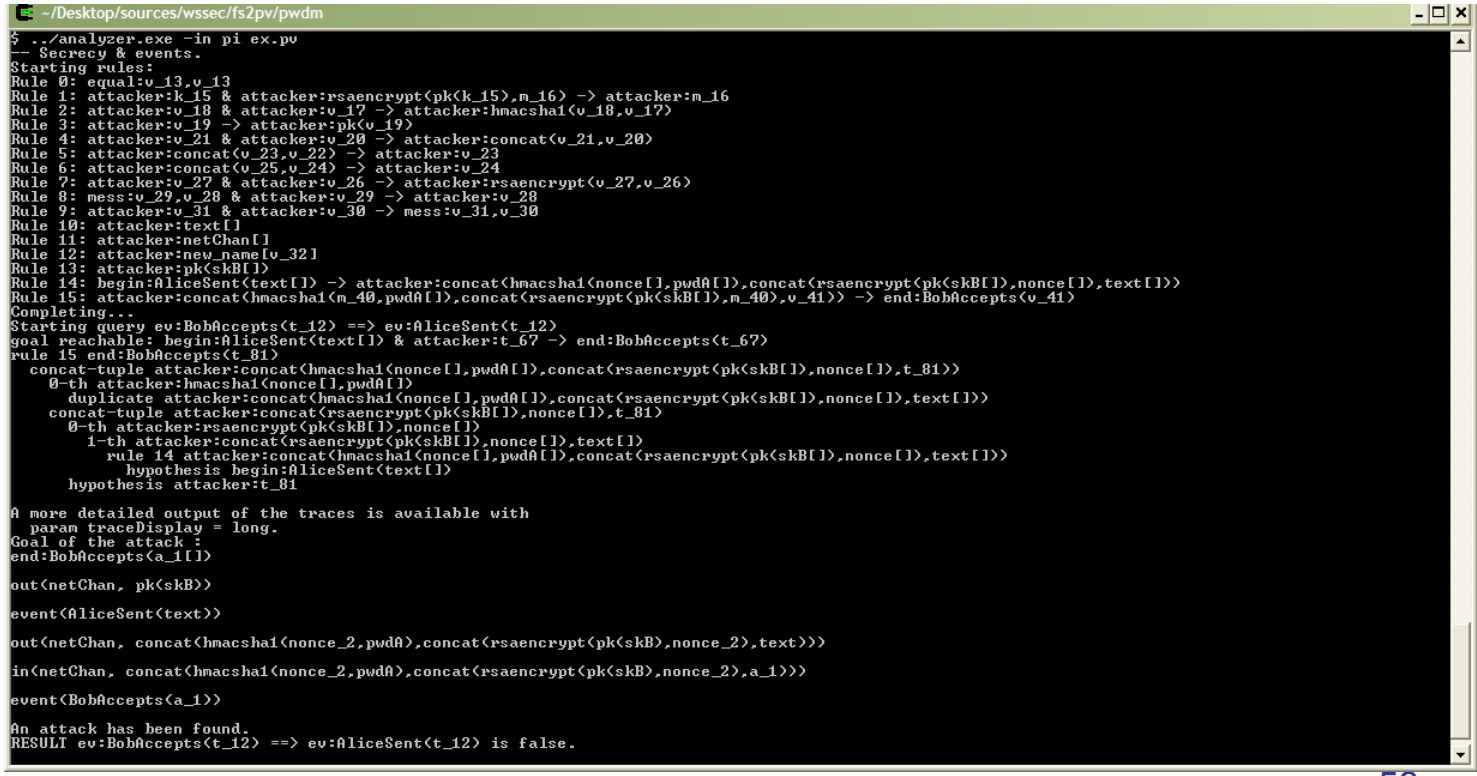
Password Secrecy →



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



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

Web Services Security
Specifications and Implementations

- 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

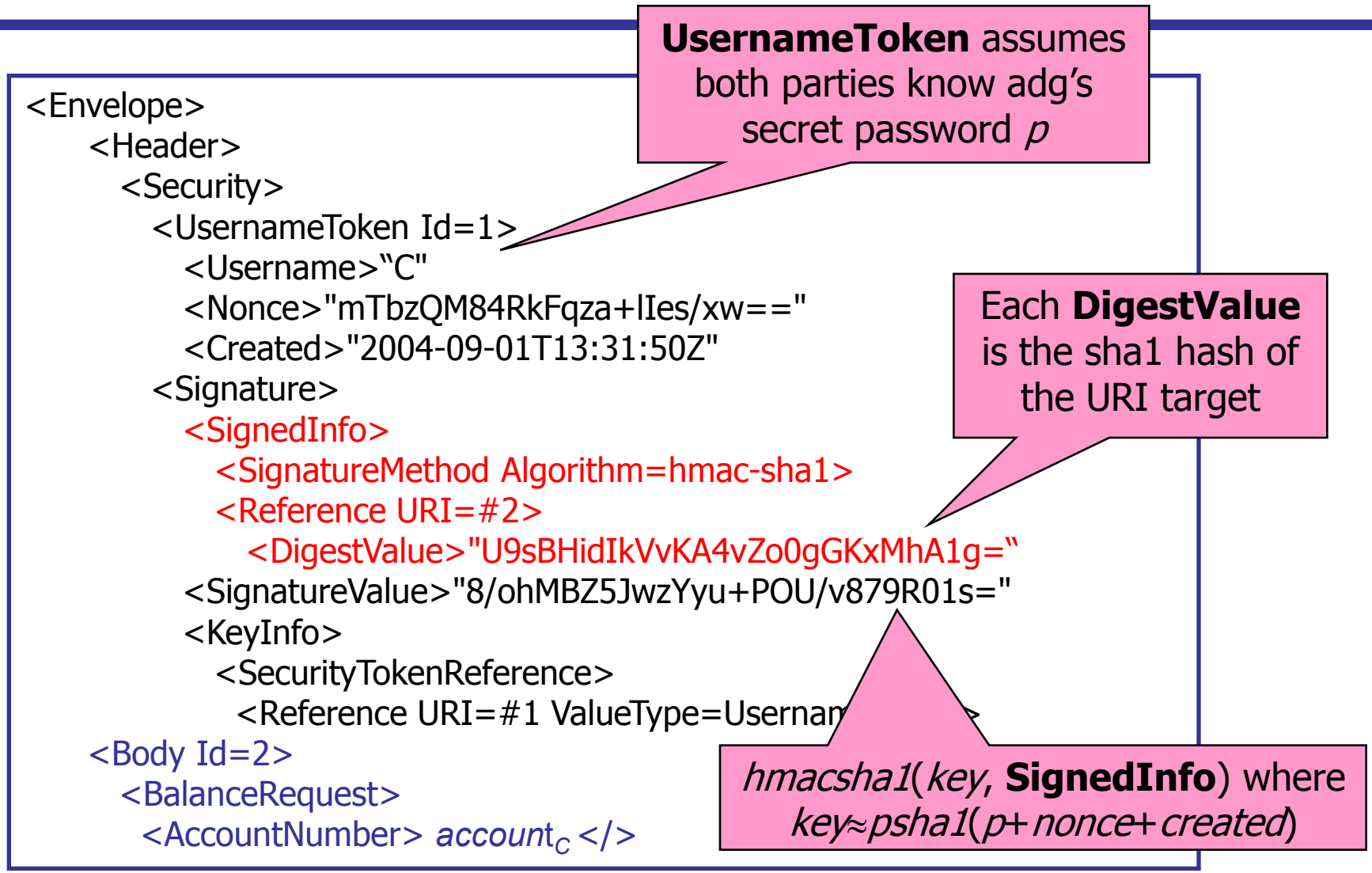
$C \rightarrow S$: $account_C$, HMAC-SHA1(pwd_C,$account_C$)
$S \rightarrow C$: $balance_C$

- 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

(HMAC-SHA1 = Keyed Hash, Message Authentication Code)

Example 1: Password-based Auth



Example 2: Signing Multiple Elements

```
<Envelope>
    <Header>
      <Action Id=1> "http://stockservice.contoso.com/wse/samples/2003/06/StockQuoteRequest"
      <MessageID Id=2> "uuid:abc4946b-112f-4a26-b923-4ffc948c15ef"
      <ReplyTo Id=3> <Address> "http://schemas.xmlsoap.org/ws/2004/03/addressing/role/
      <To Id=4> "http://localhost/UsernameSignCodeService/UsernameSigningService.asmx"
      <Security mustUnderstand="1">
        <Timestamp Id=5>
          <Created> "2004-09-01T13:31:50Z"
          <Expires> "2004-09-01T13:32:50Z"
        <UsernameToken Id=7>
          <Username> "adg"
          <Nonce> "mTbzQM84RkFqza+lIes/xw=="
          <Created> "2004-09-01T13:31:50Z"
        <Signature>
          <SignedInfo>
            <CanonicalizationMethod Algorithm=exc-c14n>
            <SignatureMethod Algorithm=hmac-sha1>
            <Reference URI=#1> ...
            <Reference URI=#2> ...
            <Reference URI=#3> ...
            <Reference URI=#4> ...
            <Reference URI=#5> ...
            <Reference URI=#6> ...
          <SignatureValue>
            "8/ohMBZ5JwzYyu+POU/v879R01s="
          <KeyInfo>
            <SecurityTokenReference>
              <Reference URI=#7 ValueType=UsernameToken>
    <Body Id=6>
      <BalanceRequest>
        <AccountNumber> account_C </>
```

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

$$
\begin{aligned}
C \rightarrow S : \; & TS \mid \\
& \text{RSA-SHA1}\{sk_C\}[symbol \mid TS] \mid \\
& \text{RSA-Encrypt}\{pk_S\}[symkey_1] \mid \\
& \text{AES-Encrypt}\{symkey_1\}[symbol] \\
S \rightarrow C : \; & \text{RSA-SHA1}\{sk_S\}[quote \mid \text{RSA-SHA1}\{sk_C\}[symbol \mid TS]] \mid \\
& \text{RSA-Encrypt}\{pk_C\}[symkey_2] \mid \\
& \text{AES-Encrypt}\{symkey_2\}[quote]
\end{aligned}
$$

Two-step encryption

Response signature includes request signature

- 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</>
        <Expires>PlusOneMinute</></>
      <BinarySecurityToken EncodingType='Base64Binary' ValueType='X509v3'
                           Id='X509Token-client.com'>
        X509(Root,C,sha1RSA,pk_C)</>
      <EncryptedKey Id='Encrkey'>
        <EncryptionMethod Algorithm='rsa-1_5' />
        <KeyInfo><SecurityTokenReference>...</>
        <CipherData>
          <CipherValue>RSA−Encrypt{pk_S}[key]</></>
        <ReferenceList>
          <DataReference URI='guid6' /></></></>
      <Signature>...</>
  <Body Id='Body'>
    <EncryptedData Id='guid6' Type='Content'>
      <EncryptionMethod Algorithm='aes128-cbc' />
      <CipherData>
        <CipherValue>AES−Encrypt{key}[
req =       <Symbol>"MSFT"</>]</></></></></></></>
```

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

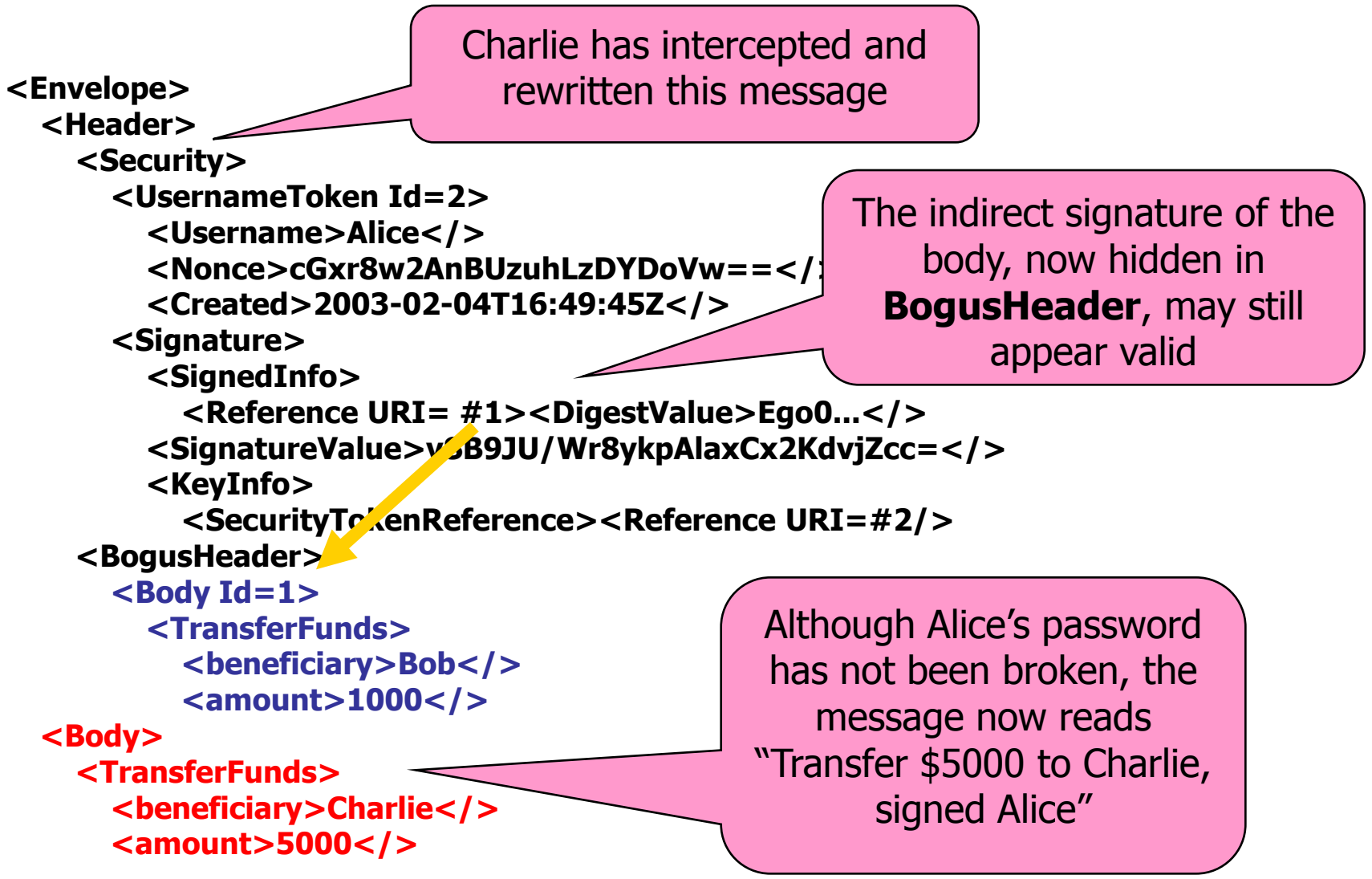
(An attack that uses the XML format)
A Signed SOAP Message Before...

```
<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=2>
        <Username>Alice</>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
        <Created>2003-02-04T16:49:45Z</>
      <Signature>
        <SignedInfo>
          <Reference URI= #1><DigestValue>Ego0...</
        <SignatureValue>VSB9JU/Wr8ykpAlaxCx2KdvjZcc=</
        <KeyInfo>
          <SecurityTokenReference><Reference URI=#2/>
  <Body Id=1>
    <TransferFunds>
      <beneficiary>Bob</>
      <amount>1000</>
```

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

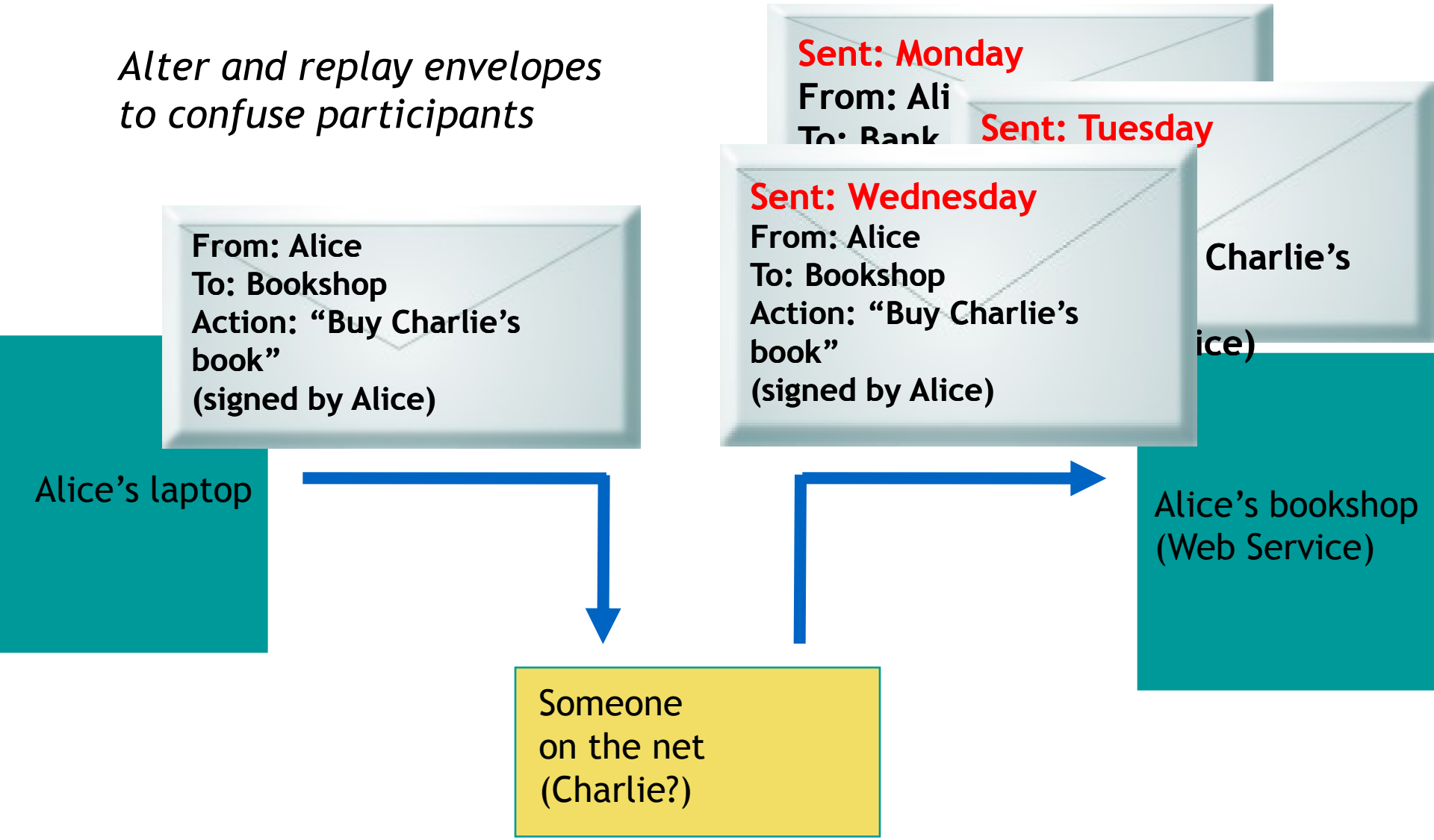


Why does the attack work?

- 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

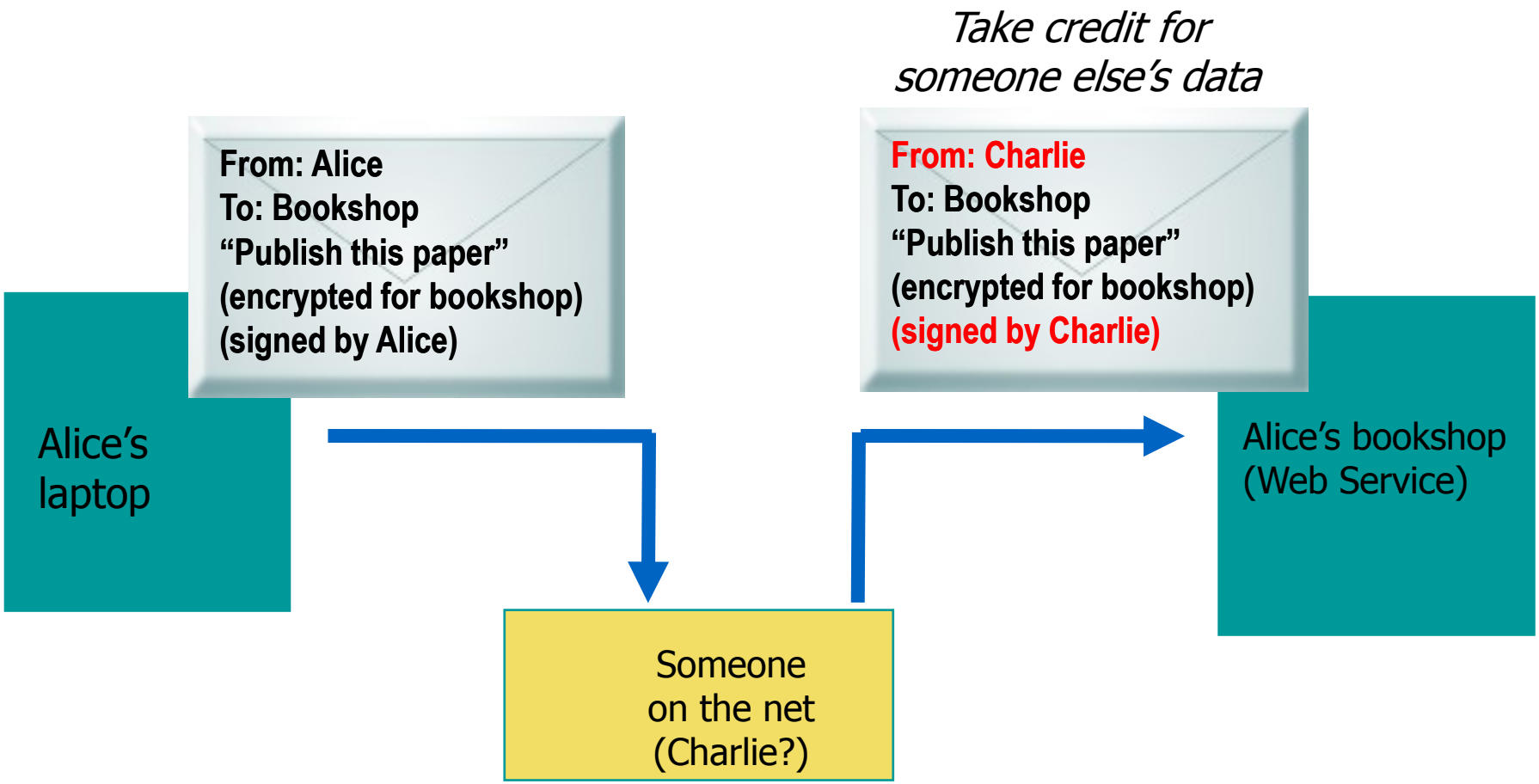
Unsigned Message Timestamps

Alter and replay envelopes to confuse participants

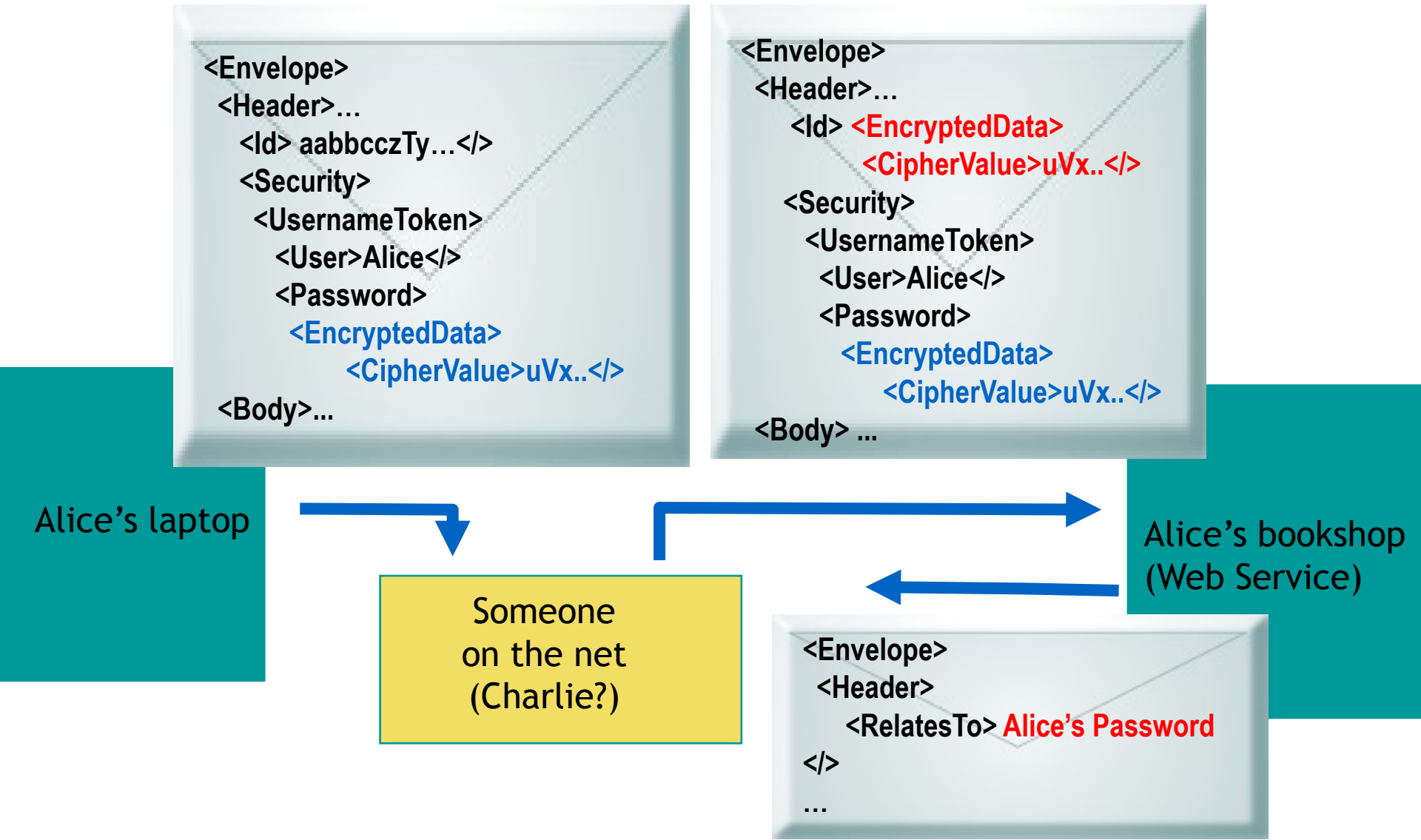


Encrypt or Sign First?

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.

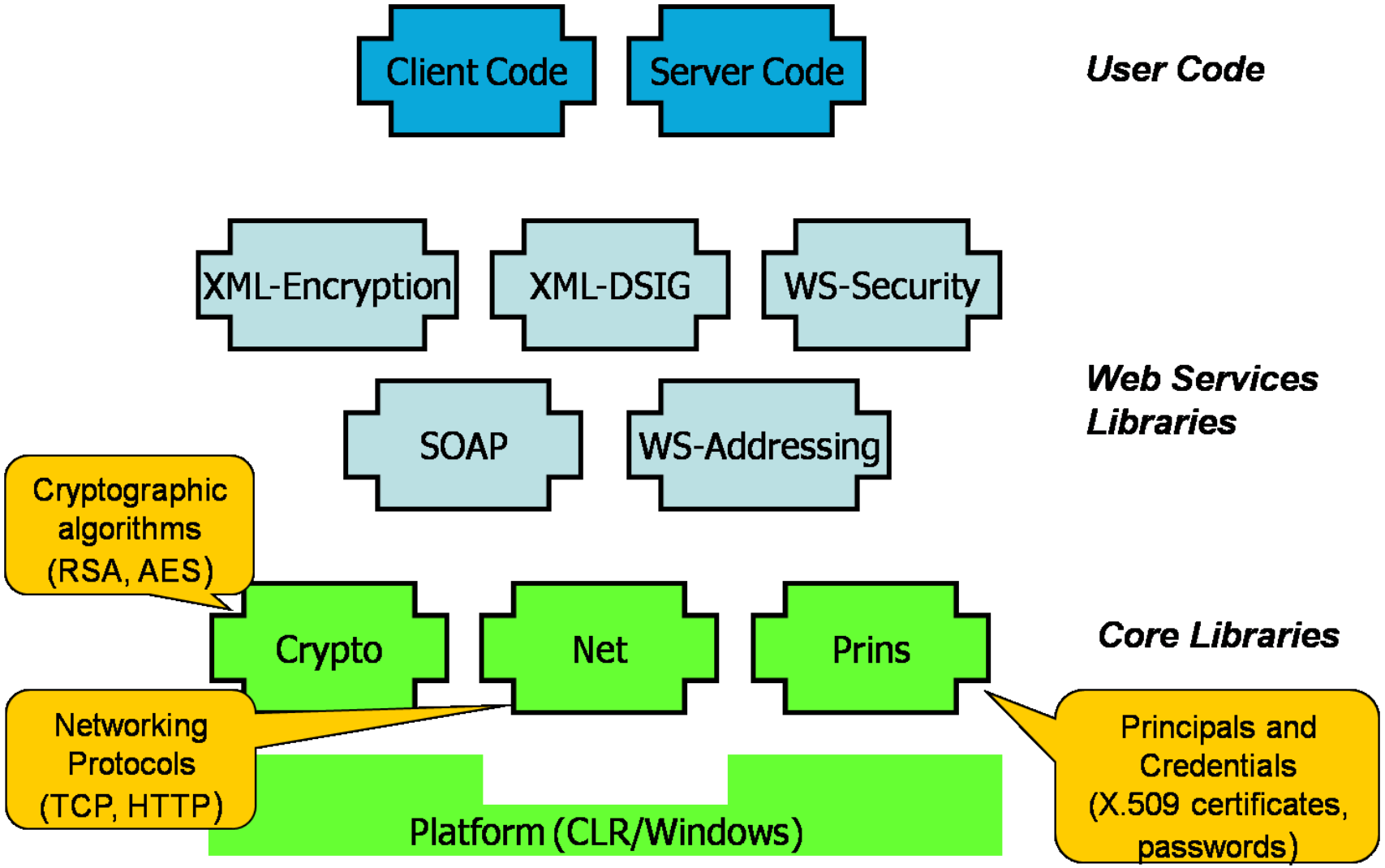


A Password Decryption Attack

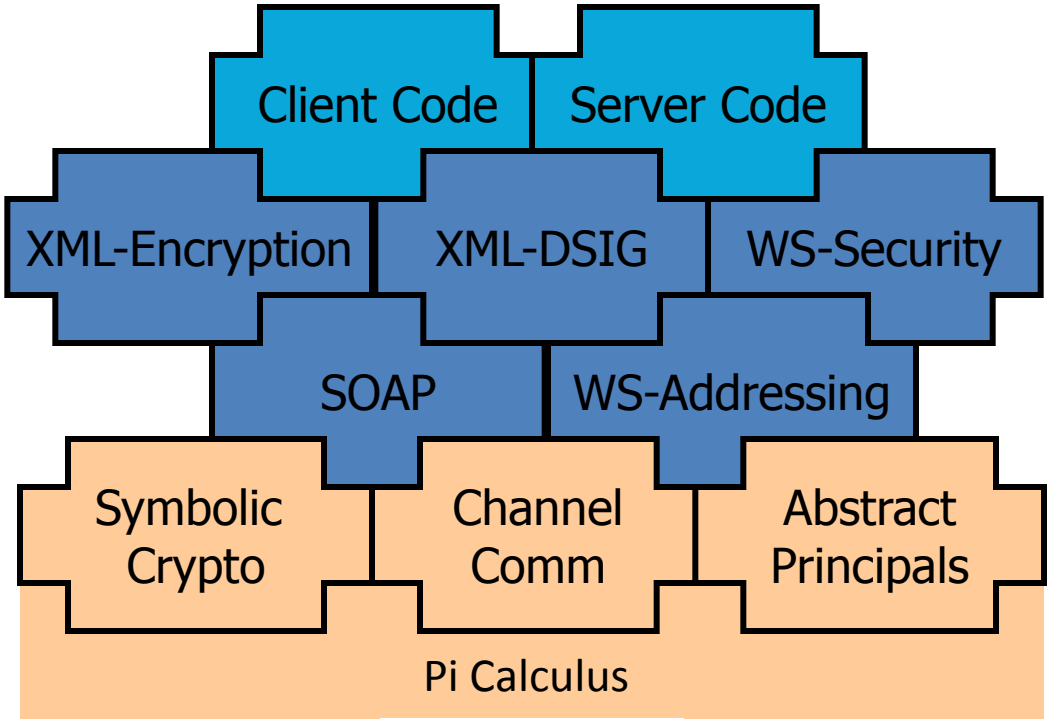


Verifying WS-Security Protocol Implementations

A Verified WS-Security Library

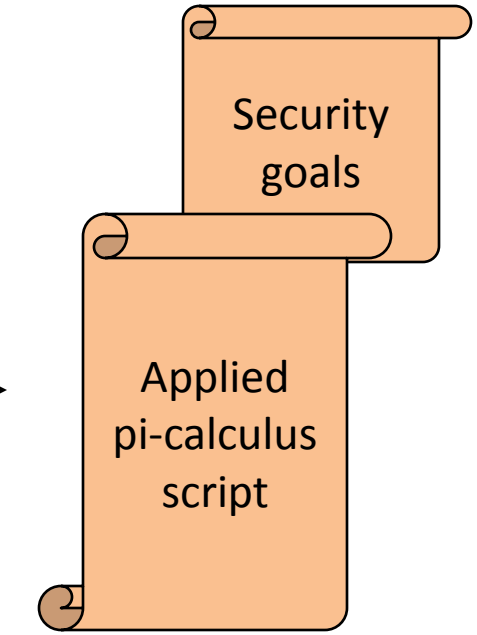


Verifying WS-Security Code

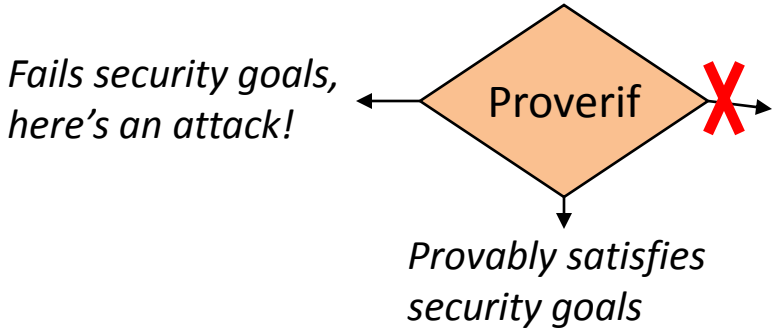


1. Replace core libraries and platform with modules implementing a symbolic Dolev-Yao Abstraction

2. **fs2pv**



3. Verify



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

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

Protocol	**Security Goals**				**Verification**	
	queries	secrecy	authenticity	insiders	clauses	time
password-based MAC	4	weak pwd	msg	no	69	0.8s
password-based MAC variant	5	pwd	msg, sender	yes	213	2.2s
Otway-Rees	16	key	msg, sender	yes	155	1m50s
WS password signing	5	no	msg, sender	yes	456	5.3 s
WS X.509 signing	5	no	msg, sender	yes	460	2.6 s
WS password-based MAC	3	weak pwd	msg, sender	no	436	10.9s
WS request-response	15	no	session	yes	503	44m45s

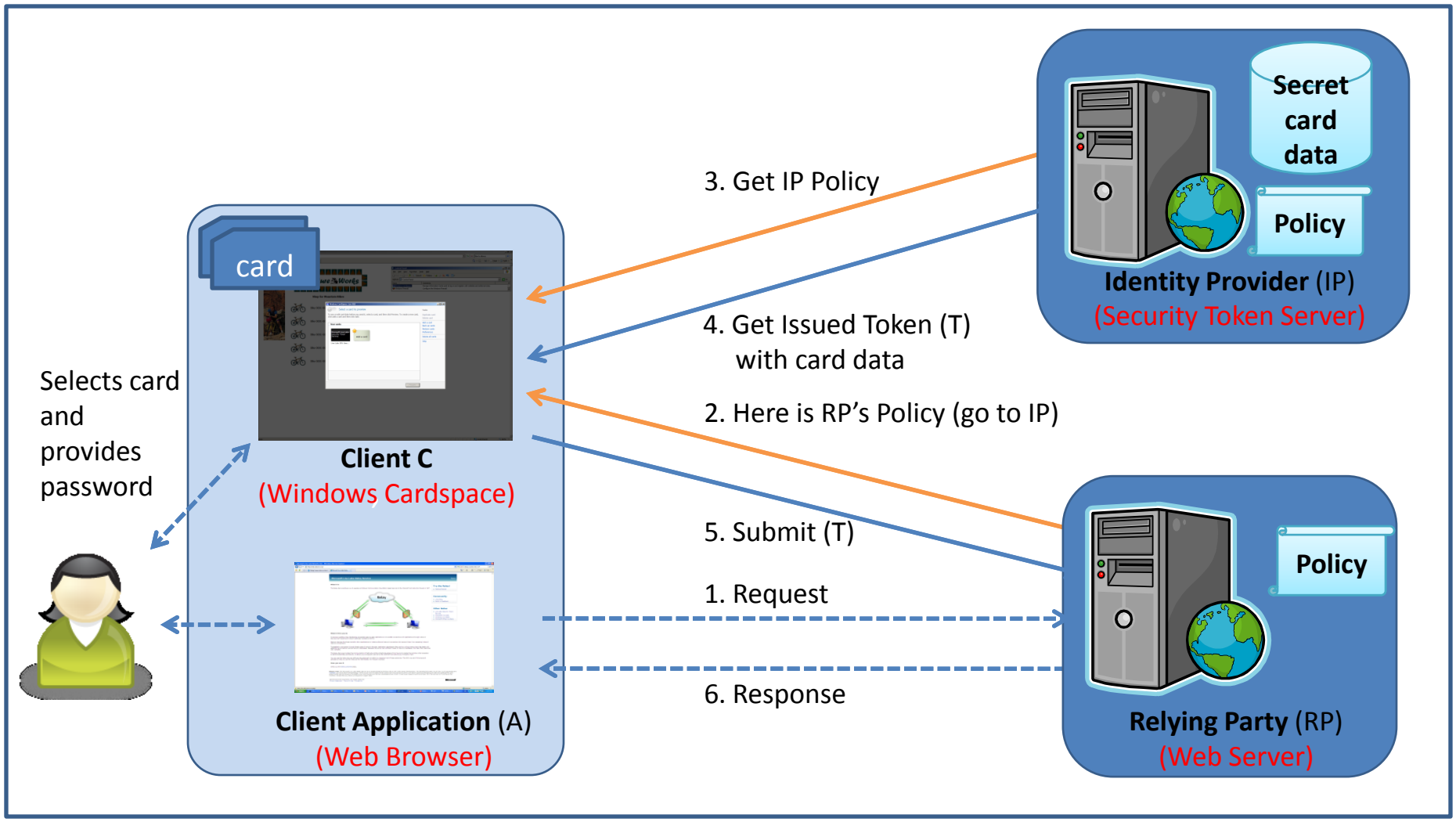
Part III
Federated Identity for Web Applications

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
  - Pfitzmann et al. 2005, Hansen et al. 2006, Bodei et al. 2003
- Can we verify their implementations?

InfoCard: Information 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 SAML Token Issuance

$$
\begin{aligned}
& C \rightarrow I : TS \mid \text{RSA-SHA1}\{sk_C\}[\text{RST}(S) \mid TS] \\
& token = \text{SAML}(C, S, \text{RSA-Encrypt}\{pk_S\}[key]) \\
& S \rightarrow C : \text{RSA-SHA1}\{sk_S\}[\text{RSTR}(token) \mid \\
& \qquad \text{RSA-SHA1}\{sk_C\}[\text{RST}(S) \mid TS]] \mid \\
& \qquad \text{RSA-Encrypt}\{pk_C\}[key]
\end{aligned}
$$

Token Usage

$$
\begin{aligned}
& C \rightarrow S : token, \text{AES-Encrypt}\{key\}[msg_1] \\
& S \rightarrow C : token, \text{AES-Encrypt}\{key\}[msg_2] \\
& \ldots
\end{aligned}
$$

- 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

Protocol Narration (Self Issued Card)

Default Protocol of Windows Cardspace

Initially	**C has:** $Card(cardId, claims_U)$, $\mathtt{PK}(k_{RP})$	**RP has:** k_{RP}

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

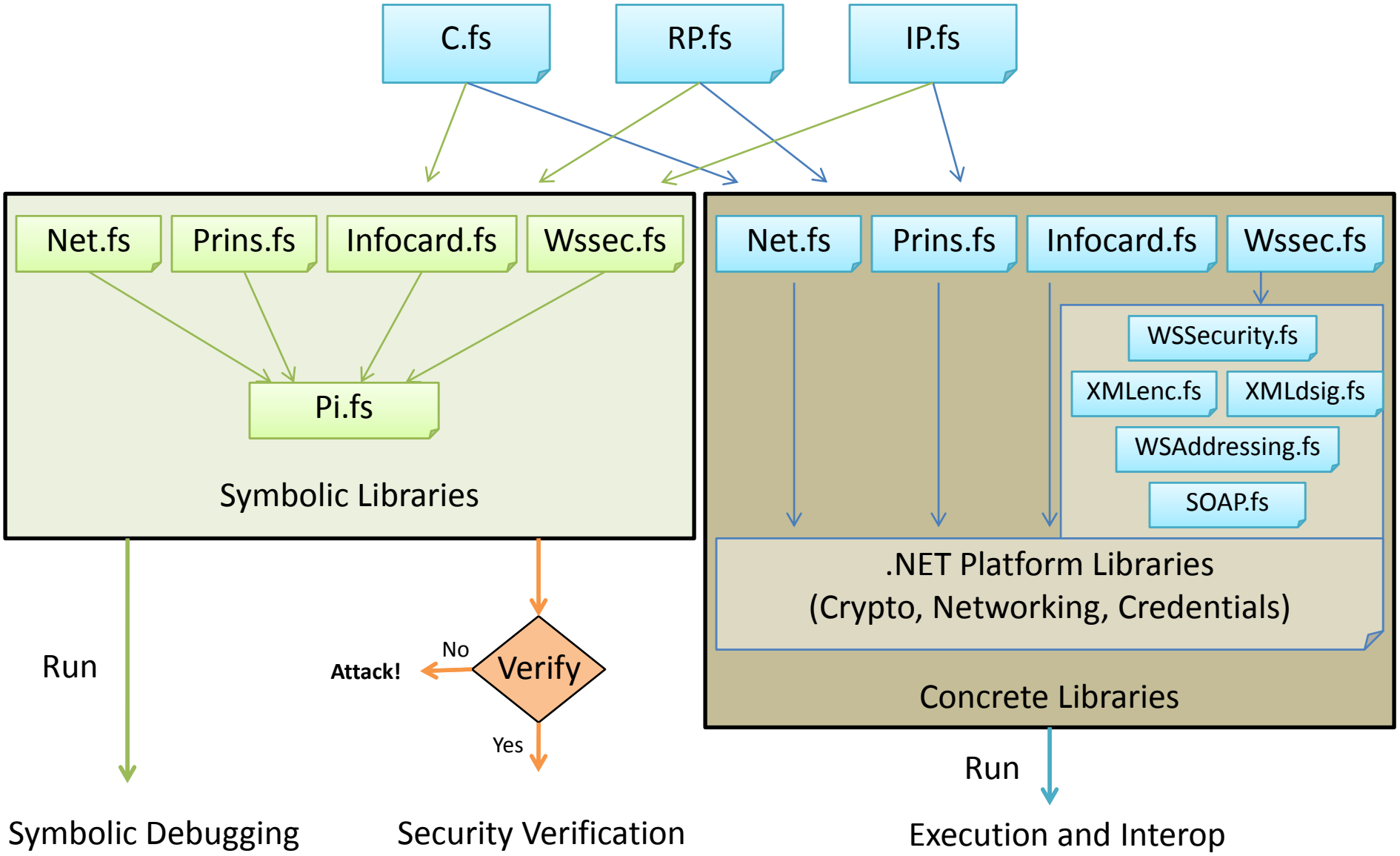
Protocol Narration for Managed Card

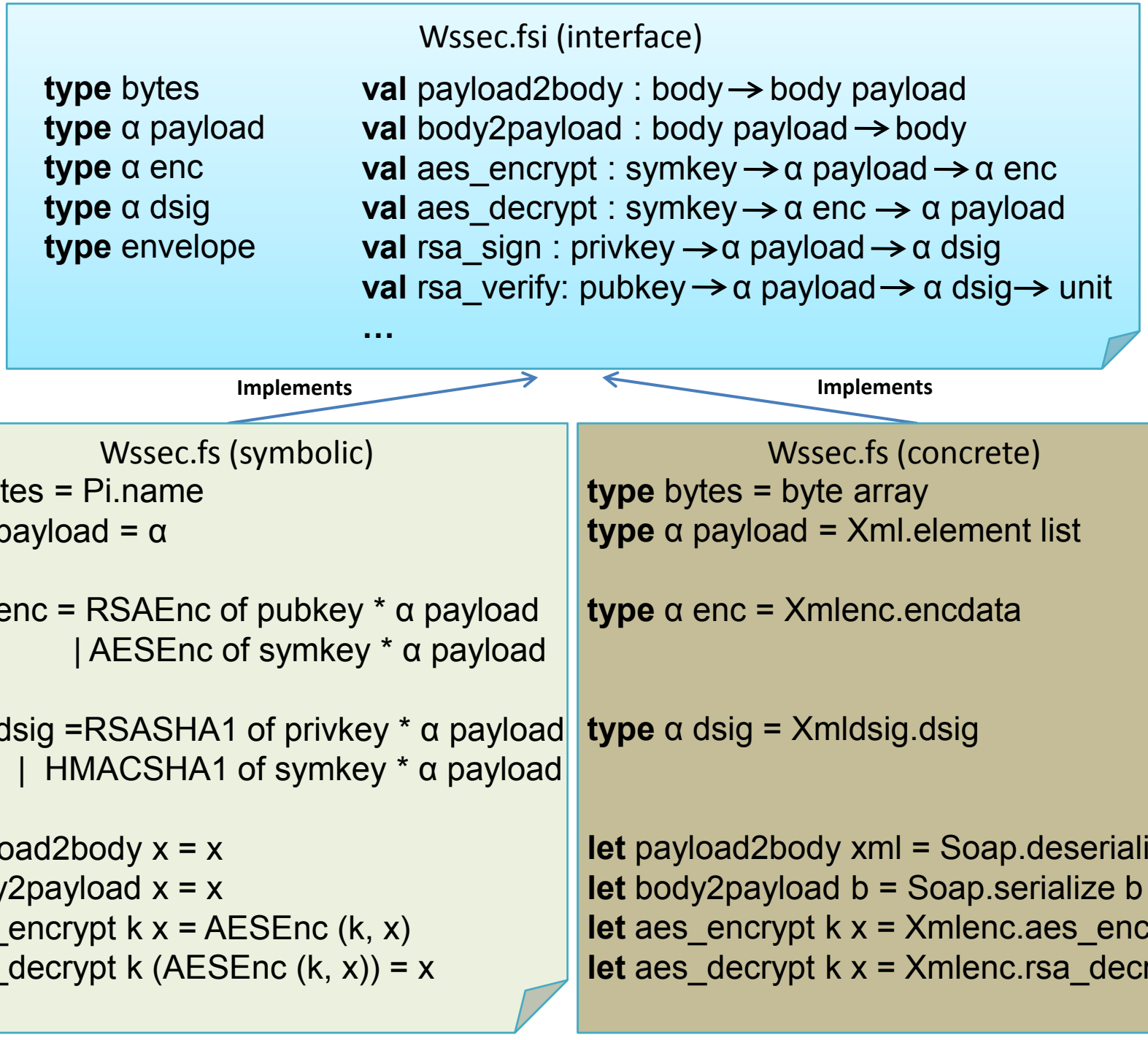
Protocol Implemented by MSN Live Labs

Initially	**C has:** $cardId$, $\mathtt{PK}(k_{IP})$, $\mathtt{PK}(k_{RP})$	**IP has:** k_{IP}, $\mathtt{PK}(k_{RP})$, $Card(cardId, claims_U, pwd_{U,IP}, k_{cardId})$	**RP has:** k_{RP}, $\mathtt{PK}(k_{IP})$

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

A Reference InfoCard Implementation




Wssec.fsi (interface)
type bytes
type α payload
type α enc
type α dsig
type envelope
val payload2body : body → body payload
val body2payload : body payload → body
val aes_encrypt : symkey → α payload → α enc
val aes_decrypt : symkey → α enc → α payload
val rsa_sign : privkey → α payload → α dsig
val rsa_verify: pubkey → α payload → α dsig → unit
…
Implements
Implements
Wssec.fs (symbolic)
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
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 (concrete)
type bytes = byte array
type α payload = Xml.element list
type α enc = Xmlenc.encdata
type α dsig = Xmldsig.dsig
let payload2body xml = Soap.deserialize xml
let body2payload b = Soap.serialize b
let aes_encrypt k x = Xmlenc.aes_encrypt k x
let aes_decrypt k x = Xmlenc.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

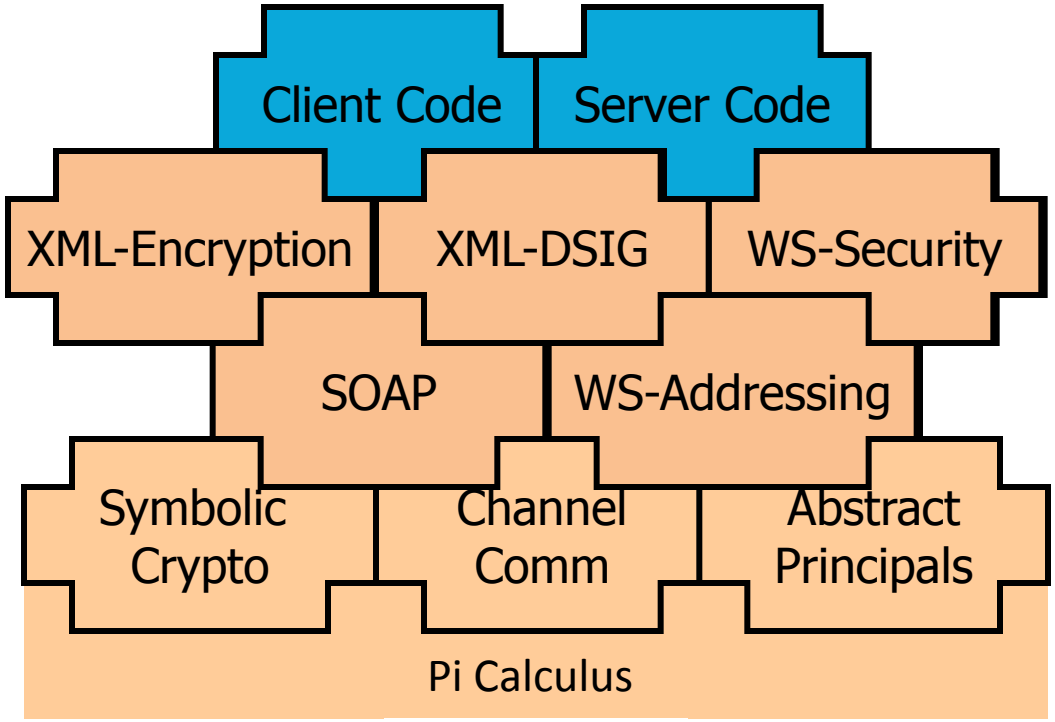
- If the token is self-issued then the pseudonym does not provide enough privacy, breaking [S2]
  - Pseudonym = Hash (cardId + RP's X.509 identifier)
  - Unless cardId is a cryptographically random secret, two colluding RP's can guess it, confirm its value by computing the hash, and correlate the use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

- *Fix:* Use a strong random cardId, keep it secret

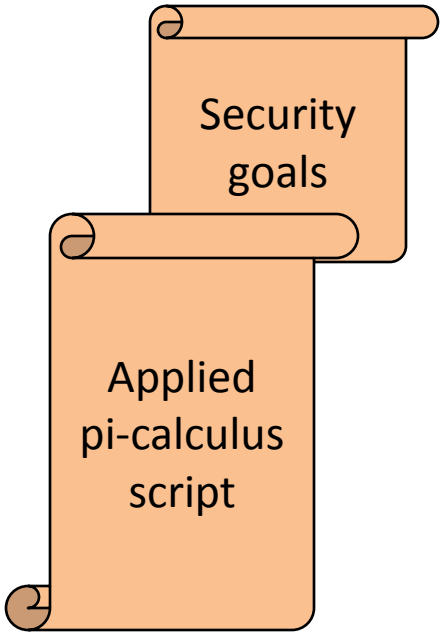
Safety Results

Name	LOC	Crypto Ops	Auth	Secrecy	Verif Time
SelfIssued-SOAP	1410 (80)	9,3	A1-A3	S1,S2	38s
UserPassword-TLS	1426(96)	0,5,17,6	A1-A3	S1,S2	24m40s
UserPassword-SOAP	1429(99)	9,11,17,6	A1-A3	S1,S2	20m53s
UserCertificate-SOAP	1429(99)	13,7,11,6	A1-A3	S1-S3	66m21s
UserCertificate-SOAP-v	1429(99)	7,5,7,4	**A3 Fails!**	S1-S3	10s

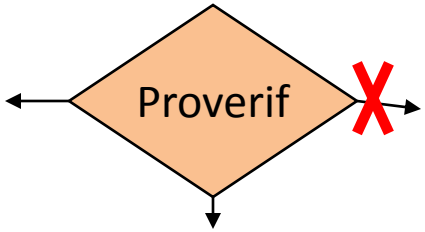
Levels of Abstraction



2. **fs2pv**



3. Verify



Fails security goals, here's an attack!

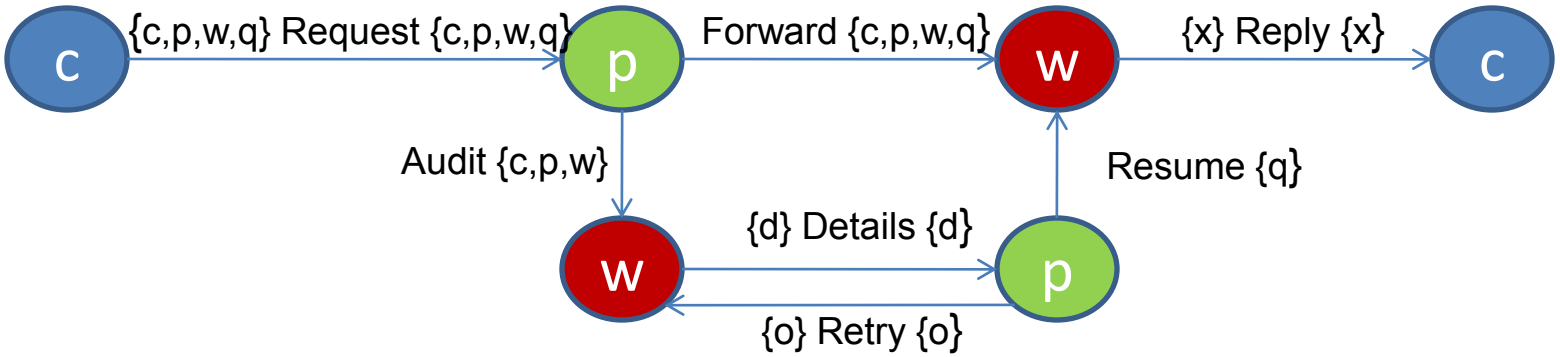
Provably satisfies security goals

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

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:



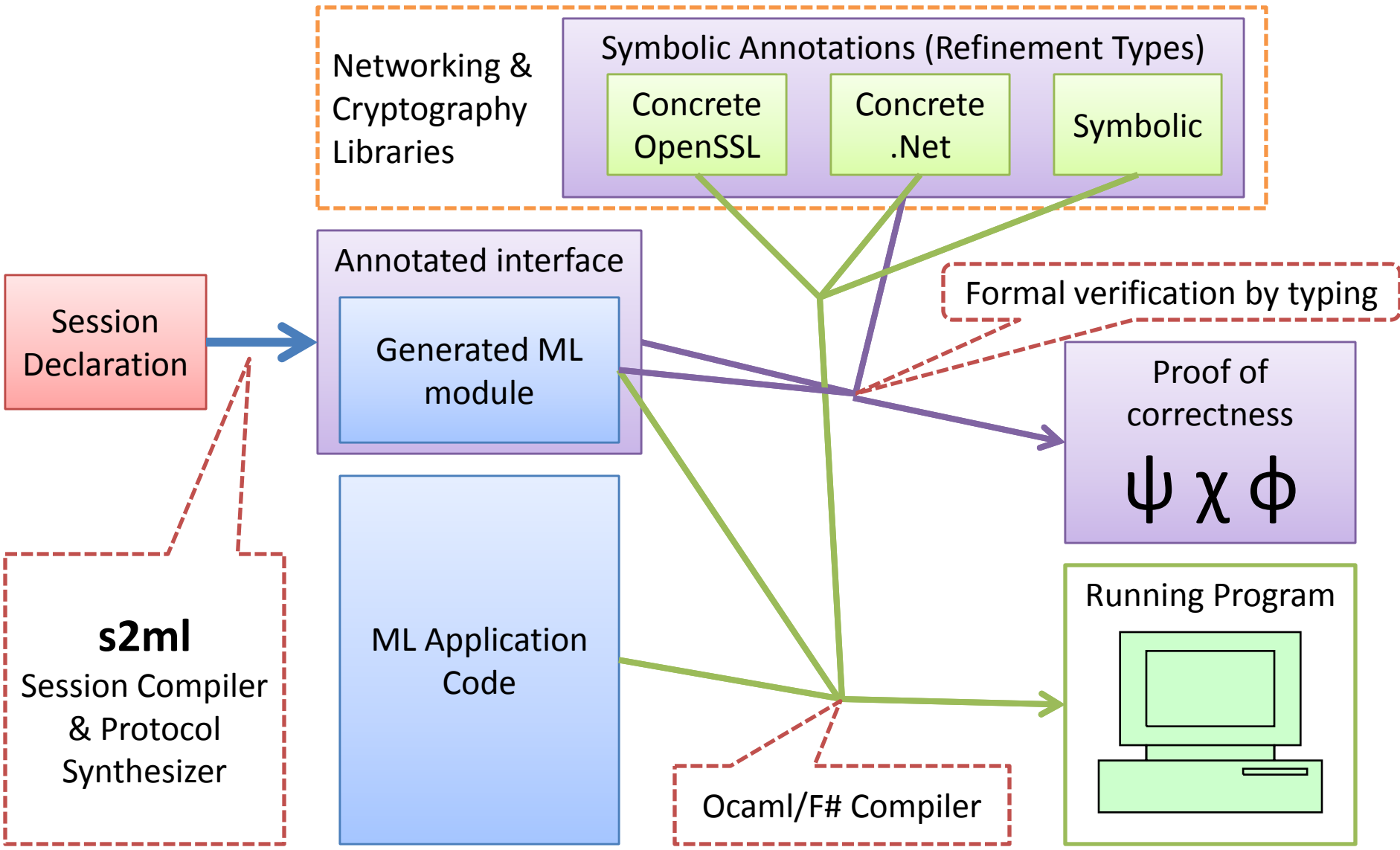
- Active area for distributed programming
  - A.k.a. protocols, or contracts, or workflows
  - Pi calculus settings, web services, operating systems
  - Common strategy: type systems enforce protocol compliance
    "If every site program is well-typed, sessions follow their spec"

Compiling Sessions to Crypto Protocols

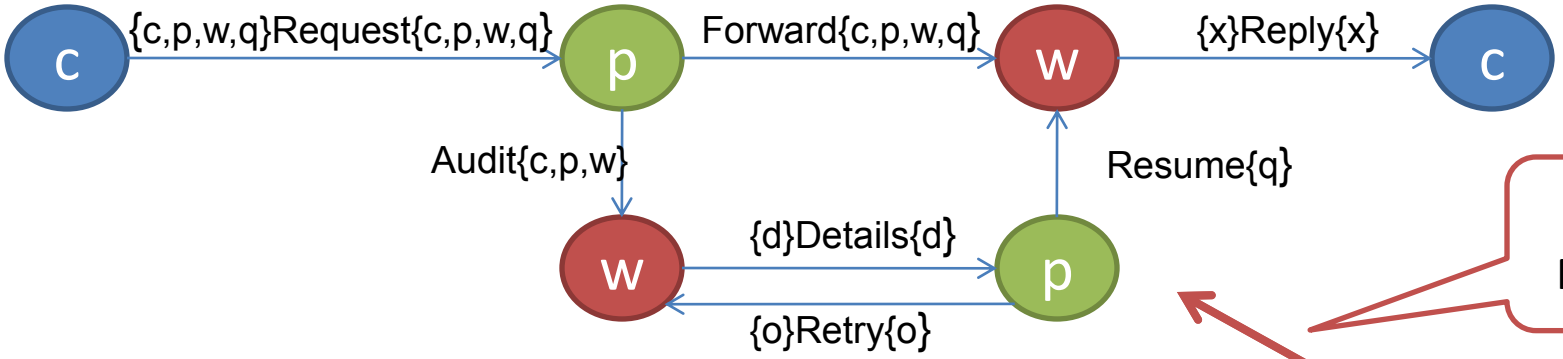
- We design a small session language.
- From a session description, we generate a secure implementation that will shield our programs from any coalition of remote peers.

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



Example: Web Service Negotiation



Role Projection

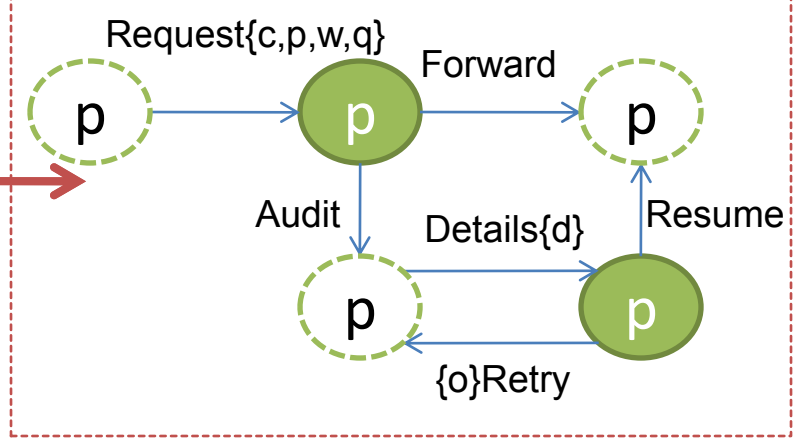
```
Session Proxy =

  var q: string  (* query *)
  var x: string  (* reply *)
  var d: string  (* details *)
  var o: string  (* objection *)

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 ))
```

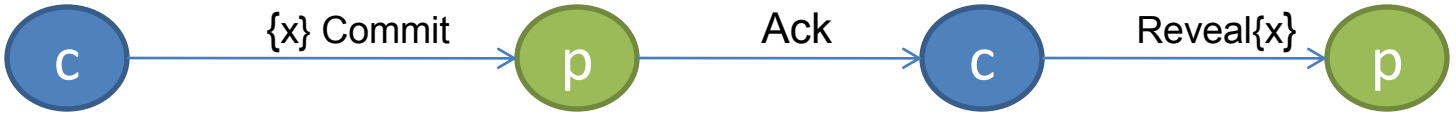
```
role w =
  […]
```



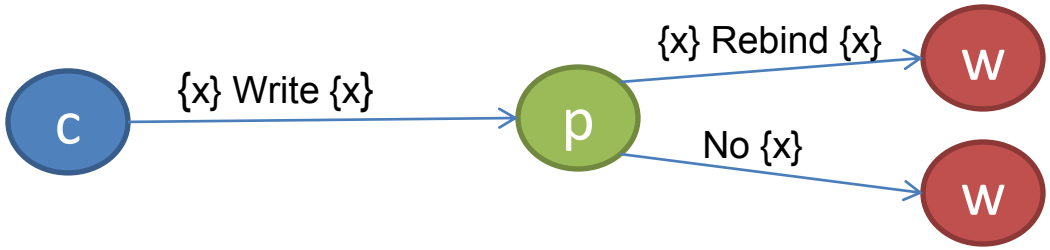
Source file `Proxy.session`

Expressiveness

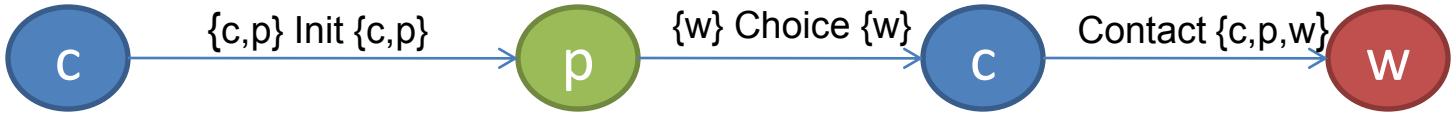
- Directed graphs with loops, branching, value passing
- A role can commit to a value and later reveal it



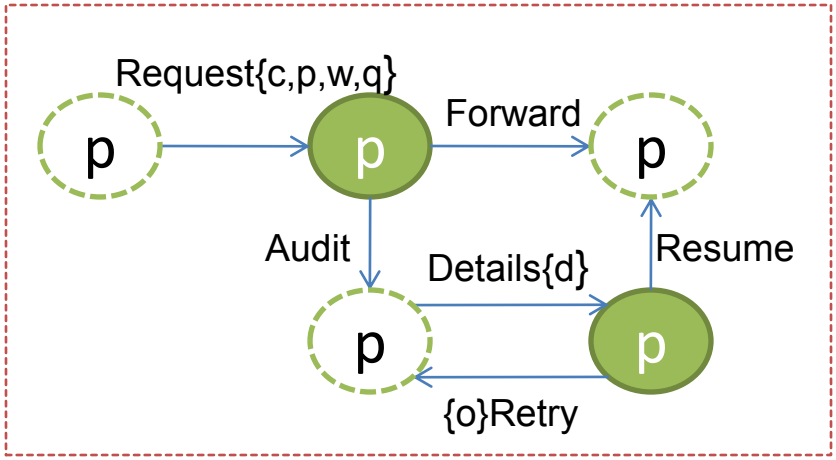
- A variable can be rebound (on some branches)



- Session participants can be dynamically selected



Local Compliance by ML Typing




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

Source file `Proxy.session`

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.

```
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
```

Generated file `Proxy.mli`

Programming With Sessions

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

```
(…)
(* 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
(…)
```

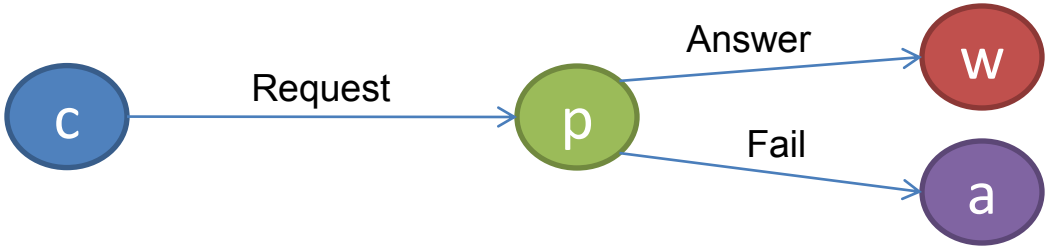
Generated file `Proxy.mli`

```
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“
    { hRequest =
        function (C _,P _,W _,Q query) ->
          match query with
            | "Simple" -> Forward ("Proxy done")
            | "Complex" -> Audit (handler_details 0)
     }
```

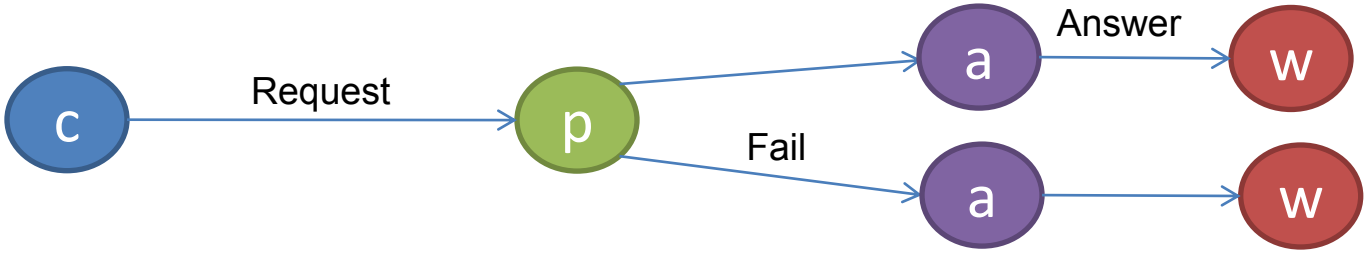
Sample of a user code file

Implementability Conditions

- We want global session integrity, not just local compliance
- Some sessions are always vulnerable:

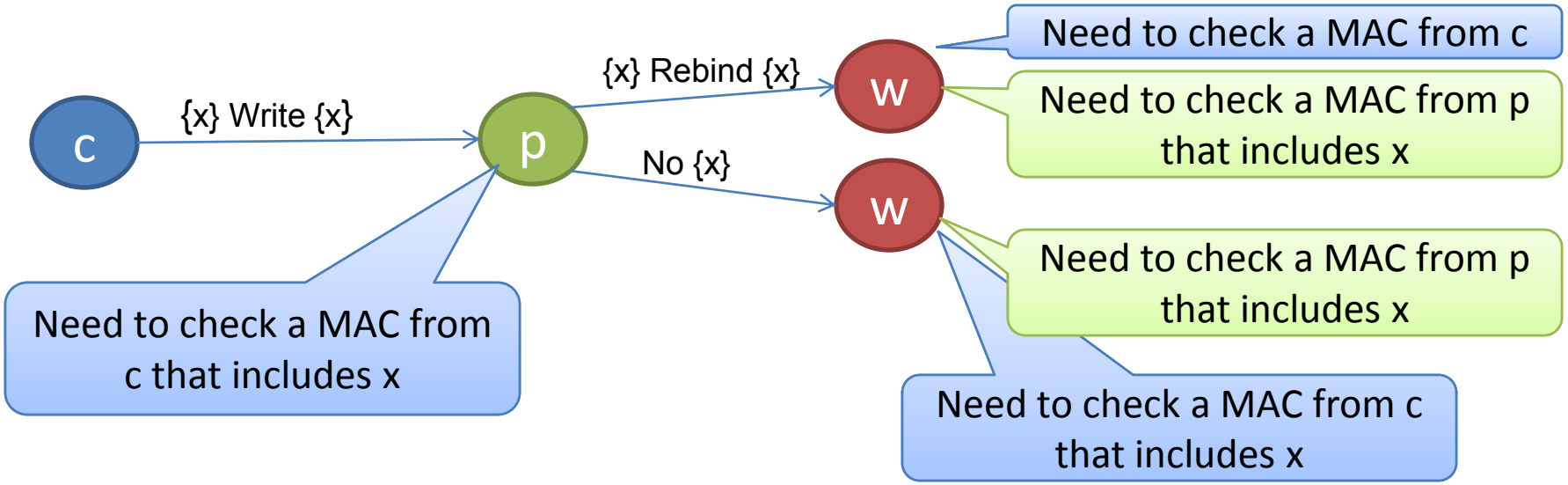


- We detect them and rule them out
  - They can be turned into safe sessions with extra messages



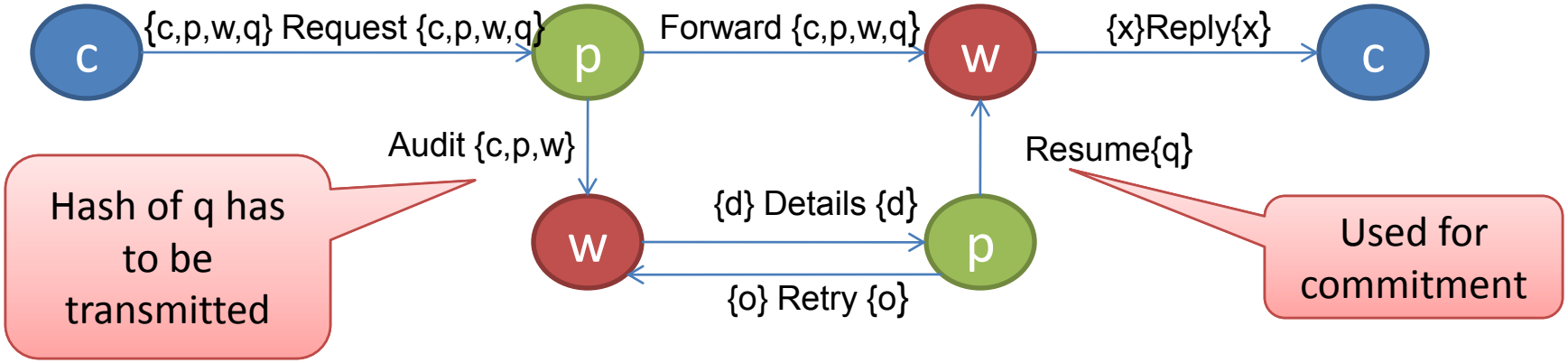
Ensuring global session integrity

- 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 involvment
  - Some variables need to be MACed as well.



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:

	Initially, c, p, w *share symmetric keys for mac and enc*		
	c :	*Assign* c, p, w, q	*Application at* c *assigns* c,p,w,q
	c :	*Fresh* s	*Fresh session identifier* s
(Request)	c → p :	let $h_0 = \text{Request}(s, 0)$ in	*Request header*
		let $m_0 = h_0 \mid \mathit{Vars}(\text{c}, \text{p}, \text{w})$ in	*Plaintext message from* c *to* p
		let $a_0 = h_0 \mid \mathit{Hashes}(\text{c}, \text{p}, \text{w}, \text{q})$ in	*Authenticated message from* c *to* p, w
		$m_0 \mid \mathit{enc}^{\text{c}}_{\text{p}}\{\text{q}\} \mid \mathit{mac}^{\text{c}}_{\text{p}}\{a_0\} \mid \mathit{mac}^{\text{c}}_{\text{w}}\{a_0\}$	*Formatted Request message*
(Forward)	p → w :	let $h_1 = \text{Forward}(s, 1)$ in	*Forward header*
		let $m_1 = h_1 \mid \mathit{Vars}(\text{c}, \text{p}, \text{w})$ in	*Plaintext message from* p *to* w
		let $a_1 = h_1 \mid \mathit{Hashes}(\text{c}, \text{p}, \text{w}, \text{q})$ in	*Authenticated message from* p *to* w, c
		$m_1 \mid \mathit{enc}^{\text{p}}_{\text{w}}\{\text{q}\} \mid \mathit{mac}^{\text{c}}_{\text{w}}\{a_0\} \mid \mathit{mac}^{\text{p}}_{\text{w}}\{a_1\} \mid \mathit{mac}^{\text{p}}_{\text{c}}\{a_1\}$	*Formatted Forward message*
	w :	*Assign* x	*Application at* w *assigns* x
(Reply)	w → c :	let $h_2 = \text{Reply}(s, 2)$ in	*Reply header*
		let $a_2 = h_2 \mid \mathit{Hashes}(\text{c}, \text{p}, \text{w}, \text{q}, \text{x})$ in	*Authenticated message from* w *to* c
		$h_2 \mid \mathit{enc}^{\text{w}}_{\text{c}}\{\text{x}\} \mid \mathit{mac}^{\text{p}}_{\text{c}}\{a_1\} \mid \mathit{mac}^{\text{w}}_{\text{c}}\{a_2\}$	*Formatted Reply message*

Generated Protocol Example (2)

- Protocol for lower path of Proxy example:

Initially c, p, w *share symmetric keys for mac and enc*			
	c :	*Assign* c, p, w, q	*Application at* c *assigns* c,p,w,q
	c :	*Fresh* s	*Fresh session identifier* s
(Request)	c → p :	let h_0 = Request(s, 0) in	*Request header*
		let $m_0 = h_0 \mid Vars(\mathrm{c},\mathrm{p},\mathrm{w})$ in	*Plaintext message from* c *to* p
		let $a_0 = h_0 \mid Hashes(\mathrm{c},\mathrm{p},\mathrm{w},\mathrm{q})$ in	*Authenticated message from* c *to* p, w
		$m_0 \mid enc^{\mathrm{c}}_{\mathrm{p}}\{\mathrm{q}\} \mid mac^{\mathrm{c}}_{\mathrm{p}}\{a_0\} \mid mac^{\mathrm{c}}_{\mathrm{w}}\{a_0\}$	*Formatted Request message*
(Audit)	p → w :	let h_1 = Audit(s, 1) in	*Audit header*
		let $m_1 = h_1 \mid Vars(\mathrm{c},\mathrm{p},\mathrm{w}) \mid Hashes(\mathrm{q})$ in	*Plaintext message from* p *to* w
		let $a_1 = h_1 \mid Hashes(\mathrm{c},\mathrm{p},\mathrm{w},\mathrm{q})$ in	*Authenticated message from* p *to* w
		$m_1 \mid mac^{\mathrm{c}}_{\mathrm{w}}\{a_0\} \mid mac^{\mathrm{p}}_{\mathrm{w}}\{a_1\}$	*Formatted Audit message*
	w :	*Assign* d	*Application at* w *assigns* d
(Details)	w → p :	let h_2 = Details(s, 2) in	*Details header*
		let $a_2 = h_2 \mid Hashes(\mathrm{c},\mathrm{p},\mathrm{w},\mathrm{q},\mathrm{d})$ in	*Authenticated message from* w *to* p
		$h_2 \mid enc^{\mathrm{w}}_{\mathrm{c}}\{\mathrm{d}\} \mid mac^{\mathrm{w}}_{\mathrm{p}}\{a_2\}$	*Formatted Details message*
	p :	*Assign* o	*Application at* p *assigns* o
(Retry)	p → w :	let h_3 = Retry(s, 3) in	*Retry header*
		let $a_3 = h_3 \mid Hashes(\mathrm{d},\mathrm{o})$ in	*Authenticated message from* p *to* w
		$h_3 \mid enc^{\mathrm{w}}_{\mathrm{c}}\{\mathrm{o}\} \mid mac^{\mathrm{p}}_{\mathrm{w}}\{a_3\}$	*Formatted Retry message*
	w :	*Assign* d′	*Application at* w *re-assigns* d
(Details)	w → p :	let h_4 = Details(s, 4) in	*Details header*
		let $a_4 = h_4 \mid Hashes(\mathrm{o},\mathrm{d}')$ in	*Authenticated message from* w *to* p
		$h_4 \mid enc^{\mathrm{w}}_{\mathrm{c}}\{\mathrm{d}'\} \mid mac^{\mathrm{w}}_{\mathrm{p}}\{a_4\}$	*Formatted Details message*
(Resume)	p → w :	let h_5 = Resume(s, 5) in	*Resume header*
		let $a_5 = h_5 \mid Hashes(\mathrm{d}')$ in	*Authenticated message from* p *to* w
		let $a'_5 = h_5 \mid Hashes(\mathrm{c},\mathrm{p},\mathrm{w},\mathrm{q},\mathrm{o},\mathrm{d}')$ in	*Authenticated message from* p *to* c
		$h_5 \mid enc^{\mathrm{p}}_{\mathrm{w}}\{\mathrm{q}\} \mid mac^{\mathrm{p}}_{\mathrm{w}}\{a_5\} \mid mac^{\mathrm{p}}_{\mathrm{c}}\{a'_5\}$	*Formatted Resume message*
	w :	*Assign* x	*Application at* w *assigns* x
(Reply)	w → c :	let h_6 = Reply(s, 6) in	*Reply header*
		let $m_6 = h_6 \mid Hashes(\mathrm{o},\mathrm{d}')$ in	*Hashes for MAC verification*
		let $a_6 = h_6 \mid Hashes(\mathrm{c},\mathrm{p},\mathrm{w},\mathrm{q},\mathrm{o},\mathrm{d}',\mathrm{x})$ in	*Authenticated message from* w *to* c
		$m_6 \mid enc^{\mathrm{w}}_{\mathrm{c}}\{\mathrm{x}\} \mid mac^{\mathrm{p}}_{\mathrm{c}}\{a'_5\} \mid mac^{\mathrm{w}}_{\mathrm{c}}\{a_6\}$	*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 Σ
- 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 Σ
  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 1st order logic proof obligations given to the Z3 SMT solver
- All generated protocols verified automatically

Session S (see Figure 1)	Roles	S.session (lines)	Application (lines)	Graph (.dot lines)	Refined Graph (.dot lines)	S_protocol.ml (lines)	S_protocol.ml7 (lines)	Verification (seconds)
Rpc	2	8	24	11	18	472	315	6.1
Ws (a)	2	8	33	14	24	592	414	8.8
Commit	2	16	29	14	24	603	399	10.3
Wsn (b)	2	21	47	20	60	1,171	921	45.1
Fwd	3	15	38	11	19	581	357	8.6
Proxy (c)	3	28	65	26	80	2,181	1,939	154.1
Redirect (d)	3	21	34	17	31	788	550	29.3
Login	4	28	54	29	74	2,053	1,542	103.4

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)

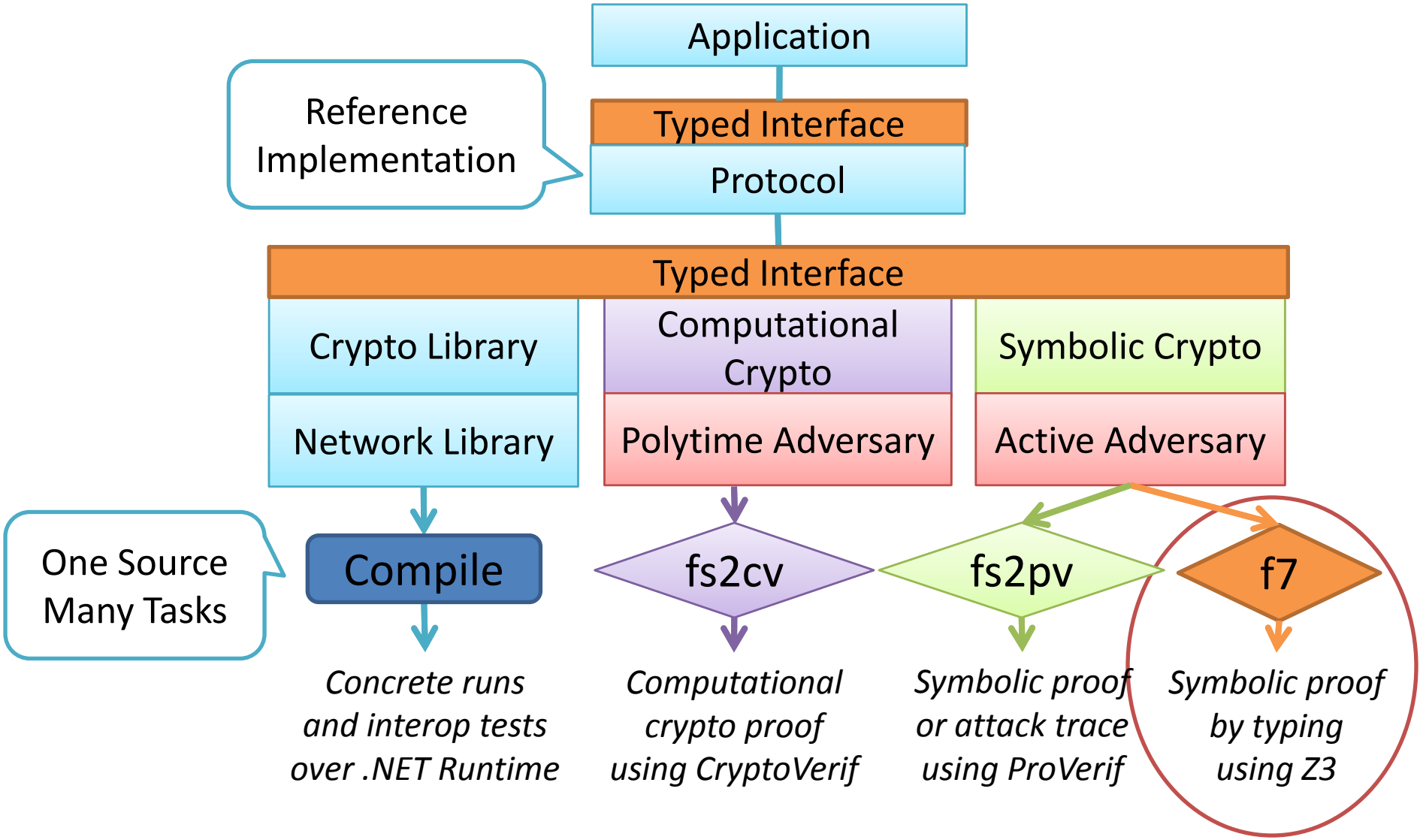
- Symbolic (Dolev-Yao) analyses of web services security protocols
  - Pi-calculus models using Proverif
    [Bhargavan, Fournet, Gordon, ... POPL'04, CCS'04, SWS'04 &'05]
  - CSP models using FDR
    [Kleiner & Roscoe ARSPA'04, MFPS'05]
  - HLPSL models using AVISPA
    [Backes, Mödersheim, Pfitzmann, Viganò FOSSACS'06]
- Computational analyses and proofs
  - Using the BPW library
    [Backes, Mödersheim, Pfitzmann, Viganò FOSSACS'06]

Related Work (Protocol Verification)

C	Goubault-Larrecq, Parrennes	2005	Csur|SPASS	FM	NSL (self-written)
Java	O'Shea	2006	LysaTool	FM	NSL, Otway-Rees (self-written)
F#	Bhargavan, Fournet, Gordon, Tse, Swamy	2006	FS2PV|PV	FM	WS protocols et al (self-written, but interoperable)
Java	Poll, Schubert	2007	JML	FSA	MIDP-SSH (independent)
F#	Bhargavan, Corin, Fournet	2007	FS2CV|CV	CM	Self-written examples

This table omits work on deriving code from models, and tools to check for insecure configurations of security protocols

Building a Cryptographic Verification Kit



Scalable Verification by Typing

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

fs2pv cannot verify

fs2pv 2.6s

fs2pv 44m

fs2pv 66m

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

  K. Bhargavan, C. Fournet, A.D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In 19th IEEE Computer Security Foundations Workshop (CSFW 2006), pp139-152, Venice, July 5-7, 2006. IEEE Computer Society.

- Verifying Web Services Security

  K. Bhargavan, C. Fournet, and A.D. Gordon. Verified reference implementations of WS-Security protocols. In 3rd International Workshop on Web Services and Formal Methods (WS-FM 2006), Vienna, September 8-9, 2006.

- Verifying Windows Cardspace

  K. Bhargavan, C. Fournet, A.D. Gordon, and N. Swamy. Verified implementations of the information card federated identity-management protocol. In ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS 2008), 123–135, 2008.

- Protocol Synthesis for Multi-party Sessions

  K. Bhargavan, R. Corin, P-M. Deniélou, C. Fournet, and J.J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In 22nd IEEE Computer Security Foundations Symposium (CSF 2009).

Questions?

All tools and papers available from

http://securing.ws

http://research.microsoft.com/~karthb

http://www.msr-inria.inria.fr/projects/sec/