

# Internship at Microsoft Research?

- 12 week research projects, undertaken at MSR Cambridge, typically by grad students mid-way through their PhD.
- Goal: complete research project with an MSR researcher (and publish if possible):
  - A. Gordon and R. **Pucella**, *Validating a web service security abstraction by typing*, 2002 ACM Workshop on XML Security, Fairfax VA, USA
  - F. Besson, T. **Blanc**, C. Fournet, A. Gordon, *From Stack Inspection to Access Control: A Security Analysis for Libraries*, 2004 IEEE CSFW, Asilomar CA, USA
- Applications for Summer 2005 are due by 28 February 2005

<http://research.microsoft.com/aboutmsr/jobs/internships/cambridge.aspx>





# Web Services and Security

---

Andy Gordon

*Microsoft Research*

**4th International School on**

**Foundations of Security Analysis and Design (FOSAD)**

September 6-11, 2004, Bertinoro University Residential Centre



# Syllabus

---

- Lecture 1, Tuesday 15:00-16:20
  - Introduction to Web Services
- Lecture 2, Tuesday 16:40-18:00
  - Nominal Calculi and Security
- Lecture 3, Wednesday 15:00-16:20
  - Modelling WS-Security in a Nominal Calculus
- Lecture 4, Wednesday 16:40-18:00
  - Modelling Trust, Secure Conversation, and Policy
- The syllabus reflects joint works with M. Abadi, K. Bhargavan, R. Corin, C. Fournet, A. Jeffrey, and R. Pucella
- Some cryptographic details drawn from M. Kuhn's security course at Cambridge University



# 1: Introduction to Web Services

---

- Basics of Web Services
- Demo: Calling a Web Service
- SOAP-Level Security
- Demo: Signing and Encrypting Messages using WS-Security, via passwords or public-key crypto
- Attacks on Web Services

# Part I: Basics of Web Services



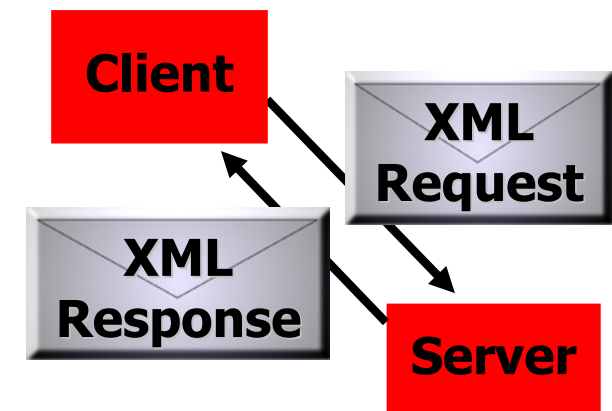
---

Websites for computers not humans; XML  
versus HTML; SOAP, WSDL, and HTTP;  
current state of adoption

# What's a Web Service?



- "A web service is a web site intended for use by computer programs instead of human beings." (Barclay et al)
- So XML not HTML
- Service messages in SOAP format:
  - Envelope/Header – addressing, security, and transactional headers
  - Envelope/Body – actual payload
- Service metadata in WSDL format:
  - For each SOAP endpoint, list of operations
  - For each operation, request and response types





# 1993: Websites for Humans

---

- HTTP: very simple access protocol
  - Text-based so bulkier than binary; latency insensitive
- URLs: pointers to remote documents
  - Lack referential integrity of familiar pointers
- HTML: document model
  - Mixes raw data with presentational markup
- MIME: very coarse type system for web documents
  - (The Semantic Web initiative: typed pointers, roughly)
- CGI: remote procedure calls
  - But streaming data model, unlike local-area RPC

Broke many of the principles of late 80s/early 90s distributed object systems; still, it did ok



# Websites for Computers B.X.E.

---

- Can trace the origins of web services before the XML Era: much work on software to access the web programmatically
  - Programmatic browsing: spiders, Cardelli and Davies' service combinators, ...
    - *Every algorithmic behaviour of web browsing should be scriptable*
    - URL = pointer + bandwidth
  - Programmatic data access: "screen-scraping", Perl, Marais' WebL
    - Widely downloaded, but didn't take off
      - The thing you have to remember about pioneers is that a lot of them got shot*





# 1998: XML

---

- Standard syntax for labelled ordered trees
  - Two kinds of label: elements and attributes
  - `<MyElement MyAttrib="fred">chas</MyElement>`
- Namespaces for modularity
  - URI qualifying element and attribute names
- Type systems: regular expressions for trees, roughly
  - DTDs – early, simple, but no namespaces
  - XML Schema – later, complex, but standard
    - The one that matters for SOAP web services
  - Relax NG – simpler, has human readable syntax
- Query languages:
  - XPath – W3C standard
  - Many PhDs and papers...



# Essential XML

## A (Slightly Simplified) Abstract Syntax for XML

<i>Label</i>	::= anyLegalXmlName	element or attribute name
<i>String</i>	::= any legal XML string	XML string
<i>Att</i>	::= <i>Label</i> ="String"	attribute
<i>Atts</i>	::= <i>Att</i> <sub>1</sub> ... <i>Att</i> <sub>n</sub>	attribute sequence
<i>Item</i>	::= <i>Element</i>   <i>String</i>	item
<i>Items</i>	::= <i>Item</i> <sub>1</sub> ... <i>Item</i> <sub>n</sub>	item sequence
<i>Element</i>	::= < <i>Label</i> <i>Atts</i> > <i>Items</i> </ <i>Label</i> >	element

- Resembles the official W3C data model, the *Infoset*
- XML intended originally as standard semi-structured data model for database integration
- XML as a general-purpose messaging format came later



# Websites for Computers X.E.

---

- “XML Web Services” refers to SOAP stack of specs:
  - SOAP – message format
    - Syntax of request, response, fault messages
  - WS-Addressing – message addressing
    - Syntax of to, from, replyto, etc, headers
  - WSDL – service description
    - Interface: function name, parameter and return types
  - UDDI – service discovery
    - Search for service by attributes (like Yellow Pages)
    - Not yet widely used in practice
  - BPEL4WS – service composition
    - Programming language for automating business processes, such as B2B order processing
    - Some sort of merger of IBM WSFL, Microsoft XLANG, and Sun WSCI ... so quite complex

# A Sample Web Service



Smart client for  
getting quotes

The Internet

www.contoso.com

```
StockService proxy = new StockService();  
string[] symbols = {"FABRIKAM", "CONTOSO"};  
StockQuote[] quotes = proxy.Request( symbols );
```

```
[WebMethod]  
public StockQuote[] Request(string[] symbols) {  
    return database.Request(symbols);  
}
```

Implementation via  
proxy class and  
HTTP transport

Implementation via  
WebService classes  
in Web Server

SOAP  
Request

SOAP  
Response

Vendor-neutral  
XML-encoding  
over HTTP

Financial  
database



# 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>
    <StockQuoteRequest xmlns="http://stockservice.contoso.com">
      <symbols>
        <Symbol>FABRIKAM</Symbol>
        <Symbol>CONTOSO</Symbol>
      </symbols>
    </StockQuoteRequest>
  </soap:Body>
</soap:Envelope>
```

- Says: "get me quotes for symbols FABRIKAM and CONTOSO"
- 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>  
    <StockQuoteRequest>  
      <symbols>  
        <Symbol>"FABRIKAM"</>  
        <Symbol>"CONTOSO"</>  
      </symbols>  
    </StockQuoteRequest>  
  </Body>  
</Envelope>
```

- Says: "get me quotes for symbols FABRIKAM and CONTOSO"
- XML not meant to be read by humans, so we'll omit namespace info, trailing brackets, and quote strings...that's better



# A Sample SOAP Response

```
<Envelope>
  <Header>
    <Timestamp>
      <Created>2003-03-11T23:36:06Z</>
      <Expires>2003-03-11T23:41:06Z</>
    </Timestamp>
  </Header>
  <Body>
    <StockQuotes>
      <StockQuote>
        <Symbol>
          "FABRIKAM"
        </Symbol>
        <Last>
          "120"
        </Last>
      </StockQuote>
    </StockQuotes>
    ...
  </Body>
</Envelope>
```

Optional header

Mandatory body

- Unlike the client making the request, the server has included a timestamp in the optional Header



# WSDL

---

- Web Services Description Language
  - Early version in 2000,
  - Like IDL in CORBA/DCOM, etc, published by a server, and consumed by client to construct proxy
  - Most of what you need to know to consume a service
    - But nothing about security, for example
- A WSDL document has 5 kinds of named description
  - Type: most commonly an XML Schema
  - Message: type for the body of a SOAP envelope
  - Port type: set of operations (function signatures) with input/output message types
  - Binding: concrete transport protocol for a port type, e.g., SOAP over HTTP, HTTP GET, HTTP POST
  - Service: set of ports, each a binding plus address



# WSDL for the Sample (1)

The descriptions begins by declaring types for request and response XML, using XML Schema

```
<definitions>
  <types>
    <schema>
      <element name="StockQuoteRequest">
        <complexType>
          <sequence>
            <element minOccurs="1" maxOccurs="1"
              name="symbols" type="ArrayOfString">
      <complexType name="ArrayOfString">
        <sequence>
          <element minOccurs="0" maxOccurs="unbounded"
            name="Symbol" type="string">
      <element name="StockQuotes">
        <complexType>
          <sequence>
            <element minOccurs="0" maxOccurs="unbounded"
              name="StockQuote" type="StockQuote">
      <complexType name="StockQuote">
        <sequence>
          <element minOccurs="0" maxOccurs="1" name="Symbol" type="s:string">
          <element minOccurs="1" maxOccurs="1" name="Last" type="s:double">
          ...
    
```

StockQuoteRequest ::=  
<StockQuoteRequest>  
 <symbols>  
 \*<Symbol>string</>

StockQuotes ::=  
<StockQuotes>  
 <StockQuote>  
 ?<Symbol>string</>  
 <Last>double</>

# WSDL for the Sample (2)

...

```
<message name="StockQuoteRequestSoapIn">
  <part name="parameters" element="StockQuoteRequest">
<message name="StockQuoteRequestSoapOut">
  <part name="parameters" element="StockQuotes">
<portType name="StockServiceSoap">
  <operation name="StockQuoteRequest">
    <input message="stockQuoteRequestSoapIn">
    <output message="StockQuoteRequestSoapOut">
<binding name="StockServiceSoap" type="StockServiceSoap">
  <binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
  <operation name="StockQuoteRequest">
    <operation soapAction="http://stockservice.contoso.com/StockQuoteRequest"
      style="document">
    <input><body use="literal">
    <output><body use="literal">
<service name="StockService">
  <port name="StockServiceSoap" binding="StockServiceSoap">
    <address location="http://localhost/.../UsernameSigningService.asmx">
```

Request and  
response messages

Input/Output association

Binding determines  
transport (http) and  
data encoding styles

Finally, service  
associates a transport  
address, port-type,  
and transport binding



# Demo: Calling a Web Service

---

High-level RPC model; SOAP messages;  
WSDL descriptions

Next: A couple of common misconceptions,  
and what actually is new here.



# “SOAP” Not Tied to Objects

---

- You’d be forgiven for thinking otherwise
  - At the start, 1998, “Simple Object Access Protocol”
  - By 2003, SOAP 1.2, “SOAP not spelt out”, just a name
- Externally, in fact, SOAP is not object-oriented
  - No instances, allocation, de-allocation
  - No distributed garbage collection
  - No classes or inheritance
  - No state at SOAP level—but see OGSII for stateful grids
- Internally, SOAP processors may be object-oriented
  - Code your service in any language you like
  - Don Box “Objects are to services what ICs are to devices”

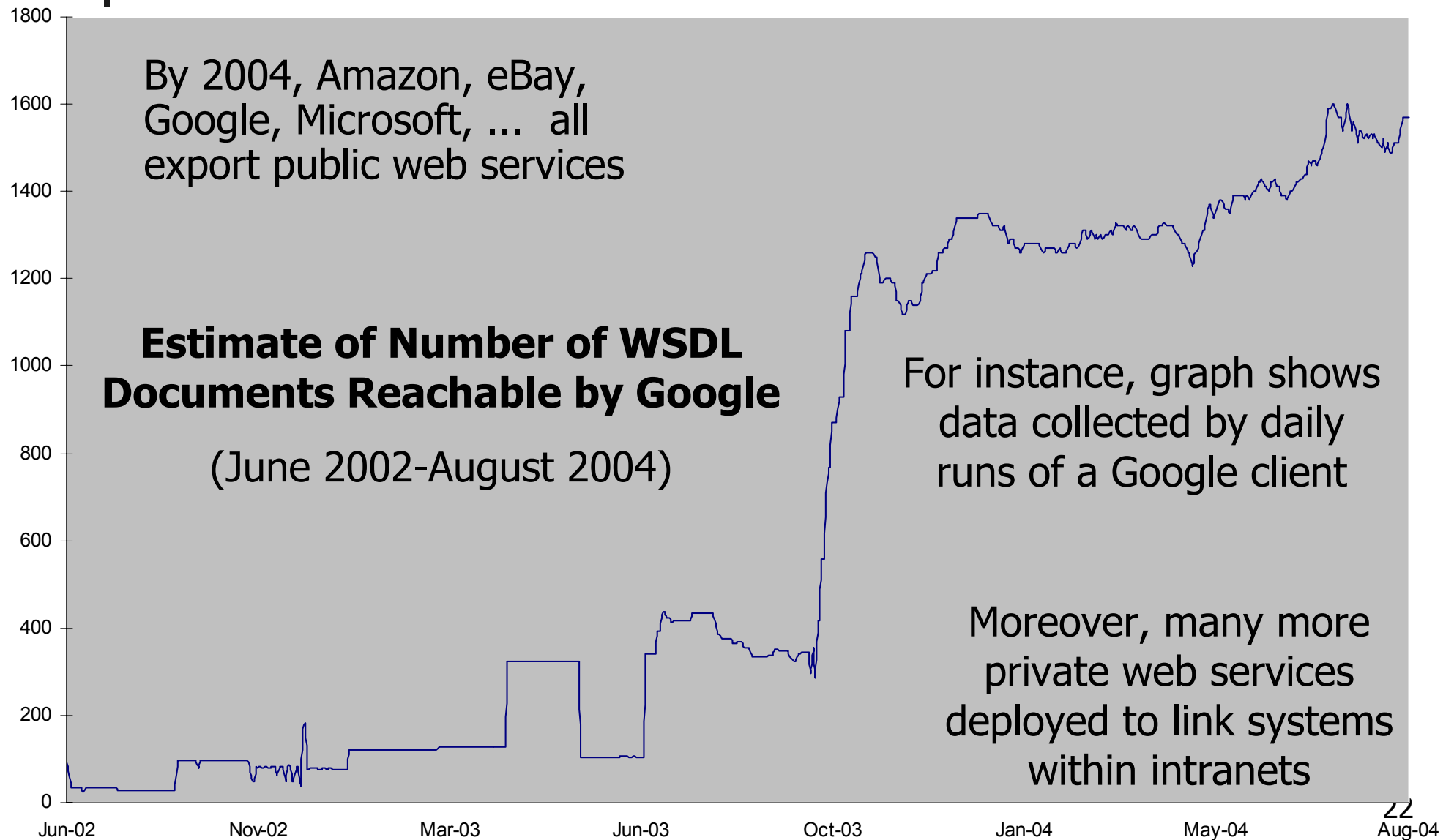


# “WS” Not Tied to the Web

---

- Originally, web services were generalization of CGI
  - Seen as “RPC over HTTP via XML” (Dave Winer, 27-Feb-98)
  - Navigating firewalls via port 80 an explicit goal
- In fact, SOAP based on asynchronous messaging
  - RPC composition of two symmetric single-shot messages
- SOAP allows for multiple transports
  - HTTP—still the common case
  - TCP—web services without a web server!
  - Message Queues—common in enterprise data centres
  - SMTP—easily supports mobile users
- SOAP allows for multiple intermediaries
  - Firewalls between trust domains
  - Gateways between transports, eg, SOAP-Mail

# Usage on Internet & Intranets





# Web Services: What's New?

---

- Though their core is roughly XML-encoded RPC – rather old! – what's new about SOAP web services is the combination of:
  - Vendor-neutral, Internet-scale, high-level tools
- Signs of fervour,
  - Wide support from commercial & OSS suppliers
  - Weekly news of progress at OASIS and W3C
  - Applications stretching from devices to the grid
- yet reasons for caution,
  - Cost of SOAP encoding?
  - Lack of SOAP security?
  - Proliferation of competing specs?
- and some competition,
  - Fielding's REST: HTTP-based web services
  - ebXML: XML version of earlier UN EDI format



# Part II: WS-Security

---

WS-Security specifies how to achieve message-level security by embedding crypto into SOAP messages

It builds on the XML-Enc and XML-DSig standards

<http://www.oasis-open.org/committees/download.php/5941/oasis-200401-wss-soap-message-security-1.0.pdf>





# The 2002 Security Story

---

- The 2002 best practice was to build secure web services using an SSL (as in HTTPS) transport
- SSL gives transport- not application-level security
  - Messages secured point-to-point not end-to-end
  - Messages cannot securely be filtered or routed
  - Messages not encrypted in files or databases
  - Moreover, SSL has scalability problems
- Party line (aka *Web Services Security Roadmap*) security within SOAP envelopes is better:
  - For end-to-end, application-level security, independent of underlying transports



# Since 2002...

---

- IBM/MS plus others publish specs...
  - Security Roadmap, Apr 2002
  - WS-Security spec, Apr 2002
  - WS-Trust, WS-SecurityPolicy, ..., Dec 2002
  - "Secure, Reliable, Transacted Web Services", Sep 2003
  - OASIS standard: SOAP Message Security 1.0, May 2004
- and release various implementations...
  - MS WSE (Web Service Enhancements)
    - 1.0, Dec 2002, implements WS-Security, etc
    - 2.0, May 2004, implements policy driven security, etc
  - Other products from IBM and others



# 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

# WS-Security: Syntax Summary

**Security** element is child of SOAP Header

*Security ::= <Security ?Actor> \*SecurityElement </>*

*SecurityElement ::=*

*<UsernameToken>*

**UsernameToken**  
identifies particular user

*<Username> String</>*

*?<Password Type="PasswordType"> String</>*

*?<Created> String</>*

*?<Nonce> Base64Binary</>*

*| <BinarySecurityToken> Base64Binary</>*

**BinarySecurityToken**  
embeds an existing format such as an X509 public-key certificate, or a Kerberos certificate

*| <SecurityTokenReference>*

*<Reference URI="Uri">*

*| <KeyInfo> \*KeyInfoItem</>*

*| <Signature> SignedInfo SignatureValue</>*

*| <ReferenceList> +<DataReference URI="Uri"/> </>*

*| EncryptedKey*

*| EncryptedData*



# Structure of an XML Signature

---

- **Signature/SignedInfo**

- Elements specifying a canonicalization algorithm (typically exc-c14n) and a signature algorithm (typically hmac-sha1 or rsa-sha1)
- Elements referring to other parts of the message, and including their hashes

- **Signature/SignatureValue**

- Outcome of applying the signature algorithm and key, to the canonicalized **SignedInfo**

- **Signature/KeyInfo**

- Pointer to signing key, such as a key derived from a user's password



# Secure Hash Functions

- A **hash function** is a pseudo random function mapping an arbitrary length input to  $n$  bits
- Additionally, a **secure hash function** satisfies:
  - *One-way*: For given  $y$ , computationally infeasible to find  $x$  with  $y=h(x)$
  - *Weak collision resistance*: For given  $x$ , computationally infeasible to find  $x'$  with  $h(x)=h(x')$
  - *Collision resistance*: It is computationally infeasible to find any  $x, x'$  with  $h(x)=h(x')$
- Examples: MD5 ( $n=128$ ), SHA-1 ( $n=160$ )
- In Dolev-Yao formal models, a secure hash function is represented as a symbolic constructor with no inverse

# (1) Password-Based Signature

```
<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=1>
        <Username>"adg"
        <Nonce>"mTbzQM84RkFqza+IIes/xw=="
        <Created>"2004-09-01T13:31:50Z"
      <Signature>
        <SignedInfo>
          <SignatureMethod Algorithm=hmac-sha1>
          <Reference URI=#2>
            <DigestValue>"U9sBHidIkVvKA4vZo0gGKxMhA1g="
          <SignatureValue>"8/ohMBZ5JwzYyu+POU/v879R01s="
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI=#1 ValueType=UsernameToken>
      <Body Id=2>
        <StockQuoteRequest>
          <symbols>
            <Symbol>"FABRIKAM"
            <Symbol>"CONTOSO"
```

**UsernameToken** assumes  
both parties know adg's  
secret password  $p$

Each **DigestValue**  
is the sha1 hash of  
the URI target

$hmacsha1(key, \mathbf{SignedInfo})$  where  
 $key \approx psha1(p + nonce + created)$

# 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/anonymous"
    <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+IIes/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>
            <StockQuoteRequest>
              <symbols>
                <Symbol> "FABRIKAM"
                <Symbol> "CONTOSO"
```

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





# Demo: Signing with WS-Security

---

To try this at home, you need:

Windows XP Pro (not Home Edition)

Visual Studio .NET or Visual Studio .NET 2003

WSE <http://msdn.microsoft.com/webservices/building/wse/default.aspx>



# Encryption Algorithms

---

- Symmetric cryptography uses a single key for both encryption and decryption
  - US Data Encryption Standard (DES): IBM, 1977, 56 bits
  - US Advanced Encryption Standard (AES): "Rijndael", KU Leuven, 2000, 128 bits
- Asymmetric cryptography uses a key pair: a public key for encryption, and a private key for decryption
  - Rivest, Shamir, Adelman (RSA): 1977, various key lengths
  - RSA decryption and encryption commute, so that "decryption" with private key is a signature, that may be verified by "encrypting" with public key
  - Asymmetric operations orders of magnitude slower than symmetric, so efficient to encrypt a message with a fresh symmetric key  $K$ , and then asymmetrically encrypt  $K$ .



# X.509 Public Key Certificates

---

- The X.509 standard is a remnant of the now abandoned early 80s Open Systems Interconnection (OSI) project, but is now widely used with Internet standards such as SSL/TLS.
- An X.509 certificate binds a public key to a human-readable subject name, and is signed by an issuer.
- An X.509 cert may include the following fields:
  - issuer's and subject's name (an X.500 directory name)
  - validity period (start and end date in UTC)
  - subject's public key (algorithm and public key data)
  - issuer's and subject's generic name (eg domain, URI)
  - identifier for issuer's policy (how was the identity verified?)
  - issuer's signature for the entire certificate

## (2) X.509-Based Signature

**X.509** is an ASN.1 format not XML, so needs to be binary encoded

```
<Envelope>
  <Header>
    <Security>
      <BinarySecurityToken ValueType=X509v3 Id=1>
        "MIIBxDCCAW6gAwIBAgIQxUSXFzWJYYtOZnmnuOMK..."
      <Signature>
        <SignedInfo>
          <SignatureMethod Algorithm=rsa-sha1>
            <Reference URI=#2>
              <DigestValue>"U9sBHidIkVvKA4vZo0gGKxMhA1g="
            <SignatureValue>"8/ohMBZ5JwzYyu+POU/v879R01s="
          <KeyInfo>
            <SecurityTokenReference>
              <Reference URI=#1 ValueType=X509v3>
            <Body Id=2>
              <StockQuoteRequest>
                <symbols>
                  <Symbol>"FABRIKAM"
                  <Symbol>"CONTOSO"
```

*rsasha1(key, **SignedInfo**)* where  
*key* is public key in X.509 cert

## (3) X.509-Based Encryption

```
<Envelope>
  <Header>
    <Security>
      <EncryptedKey>
        <EncryptionMethod Algorithm=rsa-1_5>
          <KeyInfo>
            <SecurityTokenReference>
              <KeyIdentifier ValueType=X509SubjectKeyIdentifier>
                "bBwPfItvKp3b6TNDq+14qs58VJQ="
            <CipherData>
              <CipherValue>
                "gXWRbUNSo7H5EeAO9GhE7nrq5VdBTjScMFbiftmW..."
            <ReferenceList>
              <DataReference URI=#2>
            <Body>
              <EncryptedData Id=2 Type=Content>
                <EncryptionMethod Algorithm=aes128-cbc>
                  <CipherData>
                    <CipherValue>
                      "v8XMS3XmttksWJDTnCJ86lxPW1L0cA+s16nFQgNM..."
```

$rsa-enc(key, K)$   
where  $key$  is the  
server's public key  
and  $K$  is a fresh key

$aes-enc(body, K)$



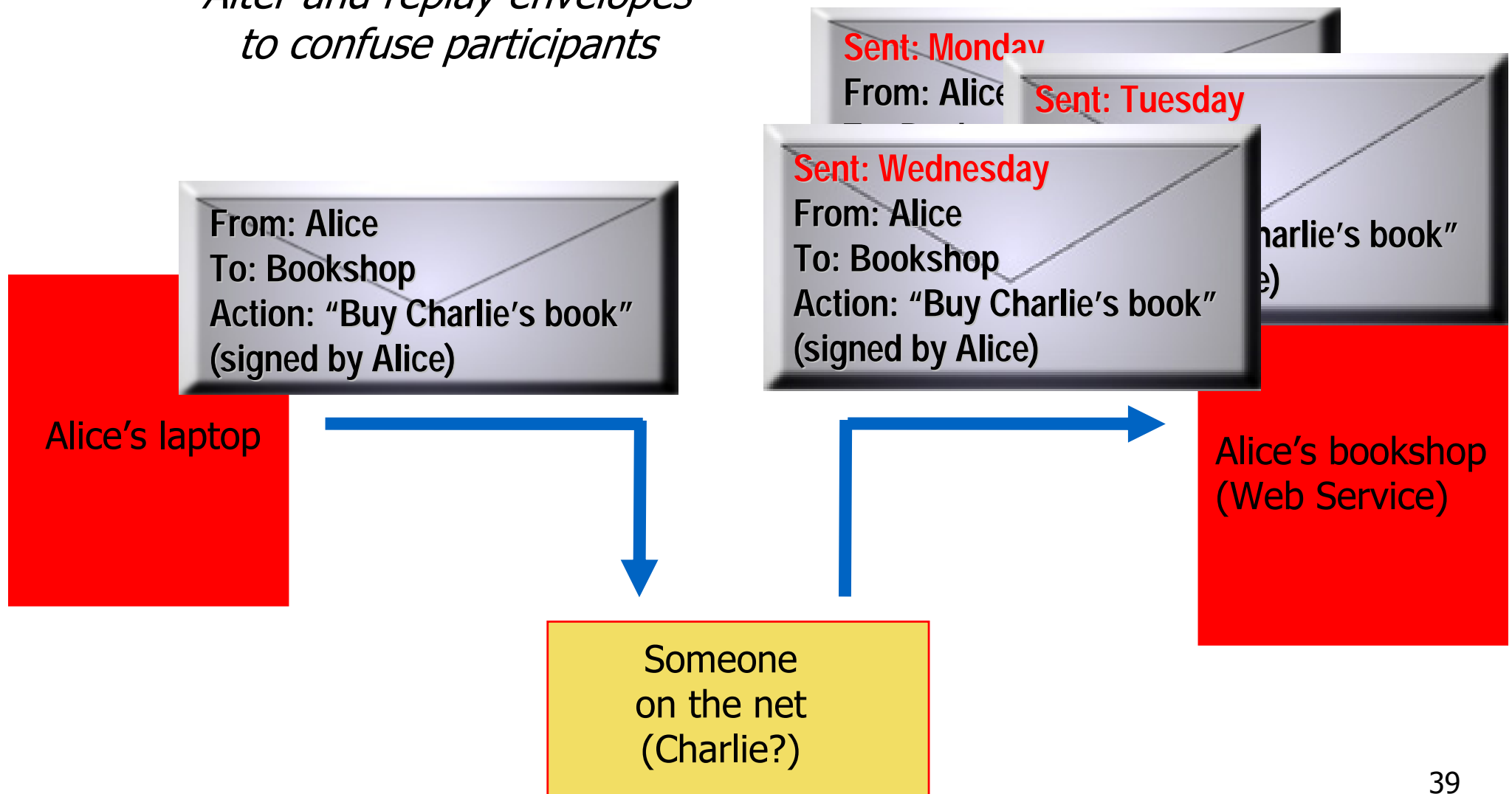
# Attacks on SOAP security

---

- Web services vulnerable to same sorts of attacks as conventional websites
  - Buffer overrun attacks, “SQL injection” attacks, etc
  - Existing “web application” security tools also applicable to web services; see eg Scambay, Shema *Hacking Web Applications Exposed* (2002)
- Moreover, there is a range of potential Needham-Schroeder attacks on SOAP messages
  - In our example, neither <To> nor <Action> is signed, so message for one server could be replayed to another
  - WS-Security needs to be flexible to support interoperability, and hazy requirements ... but flexibility is usually the enemy of security
  - We have found a range of such problems in sample code, thus motivating our research on theory and tools

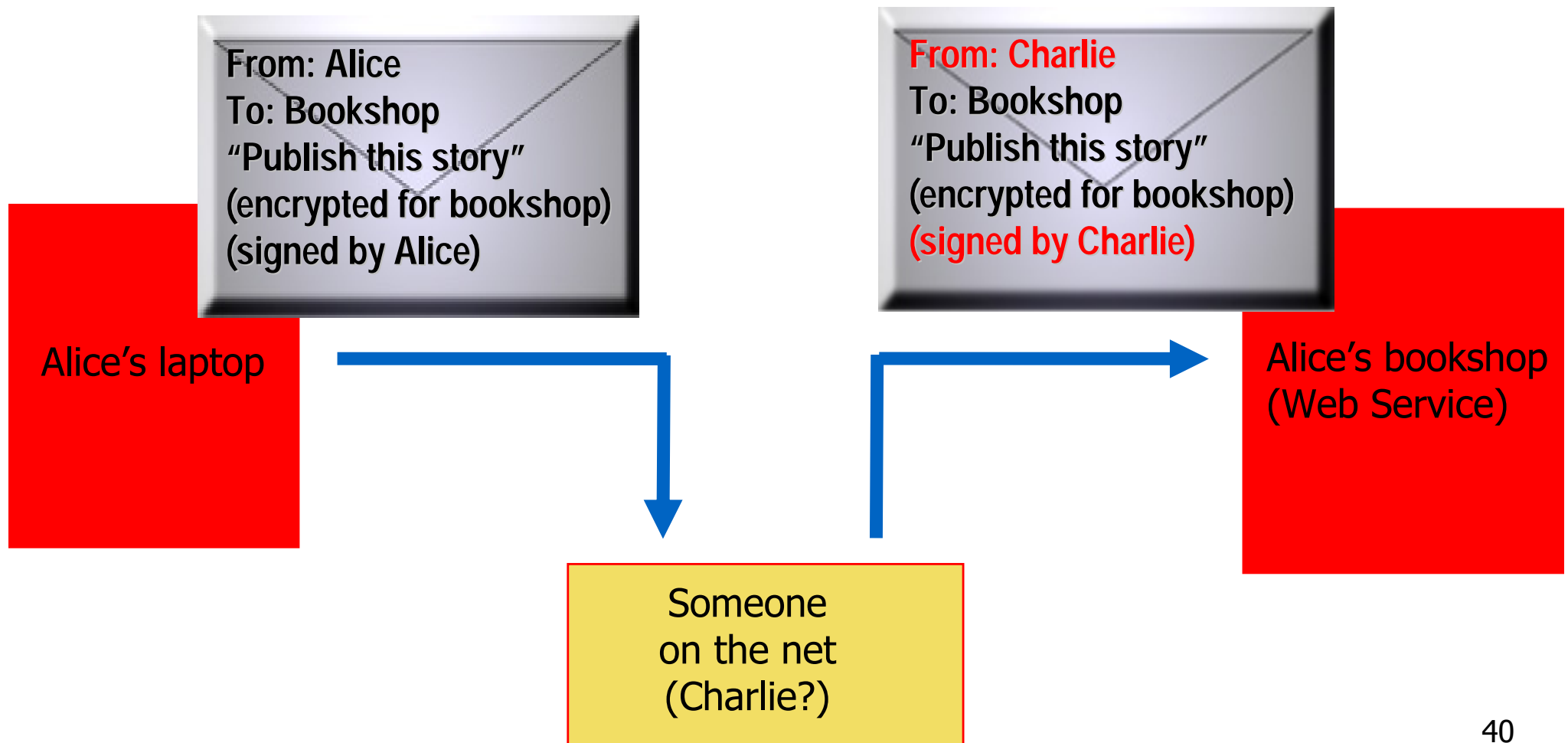
# An XML Rewriting Attack

*Alter and replay envelopes  
to confuse participants*



# Another XML Rewriting Attack

*Take credit for  
someone else's data...*







# Lecture 1: Summary

---

- SOAP and WSDL implement a fairly standard RPC mechanism on top of HTTP
  - but that has achieved unprecedented interoperability, and works on a global scale,
  - and indeed goes beyond RPC to arbitrary messaging
- XML-DSIG, XML-ENC, and WS-Security are a basis for end-to-end security guarantees, from encryption, signatures, and embedded security tokens;
  - novel features include abstraction from underlying crypto technologies, and flexibility of signatures
- XML or not, new crypto protocols are often wrong
  - Lecture 2: nominal calculi and security
  - Lectures 3 & 4: application to SOAP-level security



# Lecture 1: Resources

---

- Cardelli and Marais websites, for service combinators, and WebL
  - <http://www.luca.demon.co.uk/>
  - <http://www.hannesmarais.com/>
- Standards tracks and whitepaper
  - <http://www.w3.org/2002/ws/>
  - <http://www.oasis-open.org>
  - <http://msdn.microsoft.com/webservices/understanding/specs/default.aspx>
- Abiteboul of INRIA and Lehman of IBM, on web services:
  - <http://www-rocq.inria.fr/~abitebou/PRESENTATION/WebServices-EDBT02.pdf>
  - <http://www.btw2003.de/proceedings/proceedings.en.shtml>
- My Top Three Web Service Blogs...
  - <http://www.gotdotnet.com/team/dbox/rss.aspx>
  - <http://weblogs.cs.cornell.edu/AllThingsDistributed/index.rdf>
  - <http://www.scottishlass.co.uk/rss.xml>
- ...and my aggregate feed
  - <http://www.bloglines.com/public/adg>

End of  
Lecture 1



## 2: Nominal Calculi and Security

---

- Cryptographic Protocols and Attacks
- Nominal Calculi
- Crypto Protocols in Nominal Calculi
- Application – A WS Security Abstraction
- Cryptyc, Authenticity by Typing
- Validating our Abstraction



# Part I: Crypto Protocols and Attacks

---

Crypto protocols were invented long before web services

Protocols are quite short, and are often specified by message sequences.

Even assuming perfect crypto algorithms, replay and impersonation attacks are possible.

# Ex I: Woo and Lam (1991)



- Principal  $A$  wishes to prove its presence to principal  $B$ , via an authentication server  $S$
- Although  $A$  and  $B$  have no keys in common, the protocol can exploit secret keys  $KAS$  and  $KBS$  that  $A$  and  $B$  share with  $S$



# Ex I: Message Sequence

Message 1	$A \rightarrow B:$	$A$
Message 2	$B \rightarrow A:$	$NB$
Message 3	$A \rightarrow B:$	$\{NB\}_{KAS}$
Message 4	$B \rightarrow S:$	$B, \{A, \{NB\}_{KAS}\}_{KBS}$
Message 5	$S \rightarrow B:$	$\{NB\}_{KBS}$

- Message 5 meant to prove to  $B$  that  $A$  is currently running the protocol
- But it doesn't mention  $A$ , so by manipulating parallel sessions, an attacker  $C$  may login as  $A$



# Attacking Ex I

---

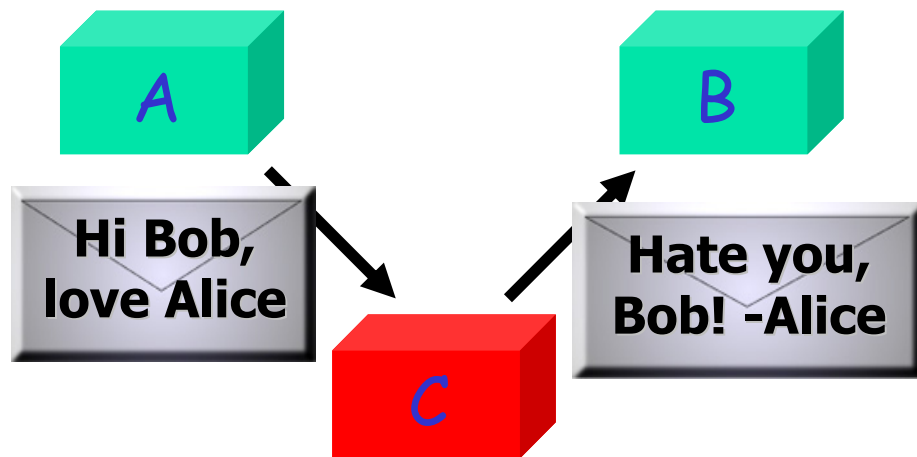
1.  $C \rightarrow B: A$
2.  $B \rightarrow C: NB_A$
3.  $C \rightarrow B: \{NB_A\}_{KCS}$
4.  $B \rightarrow S: B, \{A, \{NB_A\}_{KCS}\}_{KBS}$
5.  $S \rightarrow B: \{\dots\}_{KBS}$

1.  $C \rightarrow B: C$
2.  $B \rightarrow C: NB_C$
3.  $C \rightarrow B: \{NB_A\}_{KCS}$
4.  $B \rightarrow S: B, \{C, \{NB_A\}_{KCS}\}_{KBS}$
5.  $S \rightarrow B: \{NB_A\}_{KBS}$

- Here  $A$  is offline, but insider  $C$  runs two parallel sessions which end with  $B$  believing  $A$  has logged in.
- To fix, include the identity of  $A$  in messages 3 and 5.



# A Potted History



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 found on N&S symmetric key protocol
- 1983: Dolev and Yao first formalize secrecy properties wrt N&S threat model, using formal algebra
- 1987: Burrows, Abadi, Needham invent authentication logic; neither sound nor complete, but useful
- 1994: Hickman (Netscape) invents SSL; holes in 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 1999: Several FMs for “D&Y problem”: tradeoff between accuracy and approximation
- circa 2004: Many FMs now developed; several deliver both accuracy and automation



# Informal Methods

Experts' principles codified in articles and textbooks since mid-90s:

**Principle 1** Every message should say what it means: the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content—though if there is a suitable formalism available that is good too.

Abadi and Needham *Prudent engineering practice for crypto protocols* 1995

For instance, in our corrected protocol, Message 5 can be read “server  $S$  vouches that  $A$  encrypted  $NB$  intending to talk to  $B$ ”

Message 3	$A \rightarrow B:$	$\{B, NB\}_{KAS}$
Message 4	$B \rightarrow S:$	$B, \{A, \{B, NB\}_{KAS}\}_{KBS}$
Message 5	$S \rightarrow B:$	$\{A, NB\}_{KBS}$



# Formal Methods

---

- Dolev&Yao first formalize N&S model in early 80s
- Various FMs now exist to check properties such as:
  - Message authentication – against impersonated access
  - Message integrity – against parameter manipulation
  - Message confidentiality – against eavesdropping
  - Message freshness – against replays
  - Identity protection – preserve privacy of participants
- Certain things typically outside scope of these FMs:
  - Weak passwords – but recent progress by Lowe, Cohen, ...
  - Code defects – buffer overruns in C need different FMs
  - Denial of service – seems hard in practice and in theory
  - “Computational” models of crypto – but now active area



# Part II: Nominal Calculi

---

A **pure name** is “nothing but a bit pattern that is an identifier, and is only useful for comparing for identity with other bit patterns” (Needham 1989).

A common abstraction in computer science, formalized usefully and elegantly in the pi-calculus and its variants.

Since keys and nonces are pure names, security protocols are an important application area for nominal calculi.



# The Pi-Calculus and Mobility

---

- The pi-calculus is a tiny yet highly expressive concurrent language, with precise semantics, rich theory, and several implementations
  - Milner, Parrow, Walker (1989); Milner (1999); Sangiorgi and Walker (2000)
  - Computation is name-passing between concurrent processes on named channels
  - Each name has a mobile scope, that tracks the processes that can and cannot communicate on the name
  - Equational laws determine scope mobility and allow garbage collection of deadlocked processes
  - Pi has spawned a family of related *nominal calculi*



# Syntax of the Pi-Calculus

$x, y, z$	names
$P, Q, R ::=$	processes
<b>out</b> $x(y_1, \dots, y_n)$	output tuple on $x$
<b>in</b> $x(z_1, \dots, z_n); P$	input tuple off $x$
<b>new</b> $x; P$	new name in scope $P$
$P \mid Q$	composition
$!P$	replication
<b>stop</b>	inactivity

Names  $x, y, z$  are the only data

Processes  $P, Q, R$  are the only computations

Beware: non-standard syntax



# Example: Lists as Processes

- A list located at  $p$  awaits repeatedly for a pair of probes  $(c0, c1)$ 
  - if the list is empty, it outputs on  $c0$
  - if the list is hd consed onto  $t$ , it outputs  $(h, t)$  on  $c1$

process Nil( $p$ ) = !in  $p(c0, c1)$ ; out  $c0$  (null).

process Cons( $h, t, p$ ) = !in  $p(c0, c1)$ ; out  $c1$  ( $h, t$ ).

process Singleton( $x, p$ ) = new nil; Cons( $x, nil, p$ ) | Nil( $nil$ ).

- To return results from processes, use continuations:

process Head( $p, k$ ) = new  $c0$ ; new  $c1$ ; out  $p(c0, c1)$  | in  $c1(h, t)$ ; out  $k$  ( $h$ ).

process Tail( $p, k$ ) = new  $c0$ ; new  $c1$ ; out  $p(c0, c1)$  | in  $c1(h, t)$ ; out  $k$  ( $t$ ).

process Just( $x, k$ ) = new  $p$ ; Singleton( $x, p$ ) | out  $k(p)$ .

trace new  $k$ ; Just( $fred, k$ ) | in  $k(p)$ ; Head( $p, main$ ).

- Outcome of computation (including deadlocked processes)

new nil; new  $p$ ; ( out main( $fred$ ) | Nil( $nil$ ) | Cons( $fred, nil, p$ ) )



# Example: Garbage Collection

- Up to *structural congruence* of processes, can re-arrange scopes, provided no names are captured:

```
out main(fred) | new nil; ( Nil(nil) | new p; Cons(fred,nil, p) )
```

- Up to *behavioural equivalence* of processes, can delete a restriction around a blocked input (twice here)

```
out main(fred) | new nil; ( Nil(nil) )  
out main(fred)
```

- Structural congruence identifies processes “we’d never want to distinguish”, used to define *reduction*
- Behavioural equivalences comes in many varieties, derived from the operational semantics





# Example: Nondeterminism

- The lists below may be appended in either order

```
process Append(p,q,k) =  
  new c0; new c1; out p (c0,c1)  
  |(in c0(null); out k (q))  
  |(in c1(h,t); new k2; new r;  
    (Append(t,q,k2) | in k2(tq); new r; Cons(h,tq,r) | out k(r) )).  
  
trace Just(fred,k) | Just(chas,k) | in k(p); in k(q); Append(p,q,main).
```

- Pict (Pierce, Turner 1995) is a typed programming language based on the pi-calculus and essentially this style of programming



# Semantics of the Pi-Calculus

$P \equiv Q$  means  $P$  and  $Q$   
are equivalent states

$P \equiv P$   
 $P \equiv Q \Rightarrow Q \equiv P$   
 $P \equiv Q, Q \equiv R \Rightarrow P \equiv R$

$P \mid \text{stop} \equiv P$   
 $P \mid Q \equiv Q \mid P$   
 $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

$P \rightarrow Q$  means state  $P$   
reduces to state  $Q$

$\text{! stop} \equiv \text{stop}$   
 $\text{! } P \equiv P \mid \text{! } P$   
 $\text{new}(x); \text{stop} \equiv \text{stop}$   
 $\text{new}(x); \text{new}(y); P \equiv \text{new}(y); \text{new}(x); P$   
 $\text{new}(x); (P \mid Q) \equiv P \mid \text{new}(x); Q$  if  $x \notin \text{fn}(P)$

$P \equiv Q \Rightarrow \text{new}(x); P \equiv \text{new}(x); Q$   
 $P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$   
 $P \equiv Q \Rightarrow \text{! } P \equiv \text{! } Q$   
 $P \equiv Q \Rightarrow \text{in } x(z_1, \dots, z_n); P \equiv \text{in } x(z_1, \dots, z_n); Q$

$\text{out } x(y_1, \dots, y_n) \mid \text{in } x(z_1, \dots, z_n); P \rightarrow P\{z_1 \leftarrow y_1, \dots, z_n \leftarrow y_n\}$   
 $P \rightarrow Q \Rightarrow \text{new}(x); P \rightarrow \text{new}(x); Q$   
 $P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$   
 $P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$



# Some Applications of Pi Family

## Pi as formal semantics

- Functions, objects (pi)
- Crypto (spi, applied pi)
- Async, distributed programming and algorithms (pi, join, dpi)
- Thread, device mobility, security perimeters (ambients, seal)
- Unifying frameworks for nominal calculi (action calculi, bigraphs)
- Biomolecular modelling (stochastic pi, brane)

## Pi as source code

- Pict: channel types, concurrency, objects
- JoCaml: distribution
- Nomadic Pict: mobile agents, transactions
- Iota: untyped XML scripting for home area networking
- XLANG, BPEL: web services composition
- C $\omega$ : C# + XML + join

## Pi as a formal method

- Equivalences and refinements (eg applied to security)
- Logics: extensional eg HML, intensional eg spatial logics
- Behavioural types: graph types, secrecy & authenticity types (Cryptyc), CCS procs as types for pi procs (Behave)

# Part III: Crypto Protocols in Nominal Calculi



---

We might use any one of a great many formalisms.

As it's the basis of several type systems for security, we pick the untyped spi-calculus

M. Abadi and A. D. Gordon, *A calculus for cryptographic protocols: the spi calculus*. **Information and Computation** 148:1-70 (1999)

Related process calculi include the sjoin-calculus, the applied pi-calculus, and several others



# The Spi-Calculus in One Page

The statement **decrypt**  $M$  is  $\{x\}_N;P$  means:

“if  $M$  is  $\{x\}_N$  for some  $x$ , run  $P$ ”

Decryption evolves according to the rule:

$$\text{decrypt } \{L\}_N \text{ is } \{x\}_N;P \rightarrow P\{x \leftarrow L\}$$

- Decryption requires having the key  $N$
- Decryption with the wrong key gets stuck
- There is no way to extract  $N$  from  $\{L\}_N$
- There is no way to extract  $\{L'\}_N$  from  $\{L\}_N$

(Symmetric, authenticated encryption)



## Ex II: Specifying Authenticity

Event 1	$A$ begins	(Sender sent $msg$ )
Message 1	$B \rightarrow A:$	$NB$
Message 2	$A \rightarrow B:$	$\{msg, NB\}_{KAB}$
Event 2	$B$ ends	(Sender sent $msg$ )

- Each end-event has preceding begin-event with same label
- Attacks show up as violations of these assertions
- Named *correspondence assertions* by Woo and Lam, but also *agreements* by Lowe. Two varieties:
  - Each begin justifies a single end (*one-to-one / injective*)
    - When desired to rule out replay attacks
  - Each begin justifies many ends: (*one-to-many / non-injective*)
    - When replays do not matter, or prevented outside formalism

# Authenticity Specified in Spi

A

```
sys(msg1, ..., msgn)  $\triangleq$   
new(k);
```

```
(send(msg1, k) | ... | send(msgn, k) |  
! recv(k))
```

B

```
send(msg, k)  $\triangleq$   
in net(no);  
begin "Sender sent msg";  
out net ({msg, no}k);
```

```
recv(k)  $\triangleq$   
new(no); out net(no); in net(u);  
decrypt u is {msg, no'}k;  
check no' is no; end "Sender sent msg";
```



# Secrecy Specified in Spi

For all  $(msg_{L1}, msg_{R1}), \dots, (msg_{Ln}, msg_{Rn}),$   
 $sys(msg_{L1}, \dots, msg_{Ln}) \simeq sys(msg_{R1}, \dots, msg_{Rn})$

- No opponent  $O$  should be able to distinguish runs carrying different messages.
- We interpret  $P \simeq Q$  as may-testing equivalence.
  - A test is a process  $O$  plus a channel  $c$ .
  - A process passes a test  $(O, c)$  iff  $P|O$  may eventually communicate on  $c$ .
  - Two processes equivalent iff they pass the same tests.
- In fact, our example fails this spec...





# A Small Information Leak

- Consider  $\text{sys}(\text{msg}_1, \text{msg}_2)$ . The opponent certainly cannot obtain either of the messages in the clear, but can it tell whether they are equal?
- One may reason that  $A$ 's inclusion of the nonces always distinguishes  $\{\text{msg}_1, \text{no}_1\}_k$  and  $\{\text{msg}_2, \text{no}_2\}_k$
- But the opponent may feed its own nonce  $\text{no}$  twice to  $A$ , cause  $A$  to emit  $\{\text{msg}_1, \text{no}\}_k$  and  $\{\text{msg}_2, \text{no}\}_k$ , and hence can tell whether  $\text{msg}_1 = \text{msg}_2$
- To fix this,  $A$  sends  $\{\text{msg}, \text{no}, \text{co}\}_k$  for some fresh *confounder*  $\text{co}$  instead of simply  $\{\text{msg}, \text{no}\}_k$



# Many Variations Are Possible

---

- There is a vast literature on equationally defined information flow, e.g., “non-interference properties”
  - Focardi and Gorrieri (JCS 1994) were pioneers in the setting of process calculi
- As usual, the formalism (choice of equivalence and operational semantics) may abstract too much
  - Our spec is insensitive to covert timing channels
  - Mitchell et al study more refined calculi (CCS98...)
- Still, we now have specs of authenticity and secrecy...

# Part IV: Application – A Web Service Security Abstraction



With Riccardo Pucella

An initial experiment to verify and implement a design of  
a security abstraction for web services

A. D. Gordon and R. Pucella, *Validating a web service security  
abstraction by typing*. In **2002 ACM Workshop on XML Security**,  
pp18-29, Fairfax VA, November 22, 2002.



# Application-Level Abstraction

- Each web method has one of three security levels
  - **None**
    - Request and response sent in the clear
  - **Auth**
    - Request identity & integrity authenticated to callee
    - Response identity & integrity authenticated to caller
  - **AuthEnc**
    - Same as **Auth**, plus
    - Both request and response bodies encrypted
- Akin to, for example, secure network objects (van Doorn, Abadi, Burrows, Wobber, 1996)
- Enough to support various authorisation mechanisms
- Assume in the following, each method **AuthEnc**



# An Example Web Invocation

- Home banking is a likely application for web services
- Alice has an account at Bob's bank
  - Alice's account number is 12345
  - Bob's site is `w=http://BobsBank.com/BankingService`
- One of Bob's web methods:

```
class BankingServiceClass {  
    Id CallerId;  
    [WebMethod]  
    [SecurityLevel=AuthEnc]  
    Num balance(Num account) {  
        if (account==12345 && this.CallerId==Alice)  
            return €100 ... }}  
}
```

- Alice's secure RPC to Bob: `w:balance(12345)`



# A SOAP-Level Implementation

Message 1	$A \rightarrow B:$	$w, \text{req}(\text{getNonce}())$
Message 2	$B \rightarrow A:$	$w, \text{res}(\text{getNonce}(\text{NB}))$
Message 3	$A \rightarrow B:$	$A, \{\text{req}(w, \text{balance}(\text{acc}), t, \text{NB})\}_{KAB}, \text{NA}$
Message 4	$B \rightarrow A:$	$B, \{\text{res}(w, \text{balance}(\text{amount}), t, \text{NA})\}_{KAB}$

- We assume key  $KAB$  shared between  $A$  and  $B$ 
  - In paper version, we consider key establishment with certs
- Messages 1/2 establish security context: fresh nonce
  - Could avoid first roundtrip by including timestamps
- Messages 3/4 are the actual call/return
- Implemented using SOAP extensions in VS.NET



# An AuthEnc Envelope

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope ...>
  <soap:Header>
    <DSHeader ...>
      <callerid>Alice</callerid>
      <calleeid>Bob</calleeid>
      <np>13</np>
      <nq>-1</nq>
      <signature>4E:00:6F:00</signature>
    </DSHeader>
  </soap:Header>
  <soap:Body>
    9D:8F:95:2B:BC:60:B1:73:A7:C4:82:F5:39:20:97:F7:69:71:66:
    D3:A3:A0:90:B9:9B:FE:71:0A:65:C1:EF:EE:99:CB:4D:8A:40:37:
    CA:1E:D0:03:50:34:76:8C:E3:F3:30:DD:C9:34:19:D4:04:CB:39:
    7D:1A:84:2F:CA:30:DA:68:7E:E1:CB:07:9C:EB:79:F9:E9:4B:47:
    5B:94:56:D7:22:0E:02:CD:AA:F5:D3:40:C1:EC:13:FB:B9:E6:4F:
    13:CD:70:FD:BA:18:80:FC:50:F3:75:F2:2F:95:50:5D:41:7E:C8:
    8B:BB:AB:76:C9:59:BA:E2:3B:E5:4D:79:71:E4:AD:18:5A:4B:EA:
    29:17:30:90:66:08:27:ED:B4:BD:2E:89:06:6D:0B:56:40:43:35:
    A1:77:AE:12:7E:4B:19:26:B5:24:1A:D9:67:3D:A0:9E
  </soap:Body>
</soap:Envelope>
```



# What Do We Have So Far?

---

- We have outlined a new “security abstraction”
  - Defined by custom attributes on web methods
  - Implemented by SOAP extensions
- Next, to validate using Cryptyc:
  - We formalise the abstraction as an object calculus
  - We specify its semantics by translation into spi, a process representation of the Dolev-Yao model
  - Since the translation preserves typings, attacks representable in spi are impossible
  - Verification of formal model, not running code





# Part V: Cryptyc, Authenticity by Typing

---

With Alan Jeffrey (De Paul University)

Tool typechecks authenticity and secrecy properties of cryptographic protocols in spi

Series of papers develops the theory

A. D. Gordon and A. Jeffrey, *Authenticity by typing for security protocols*. **Journal of Computer Security** 11(4):451-521, 2003.



# How to Type Authenticity

- We introduce a type and effect system
- Judgment  $E \vdash M : T$  means message  $M$  has type  $T$
- Judgment  $E \vdash P : [L_1, \dots, L_n]$  means process  $P$  has *effect*  $[L_1, \dots, L_n]$ , a (multiset) bound on the events that  $P$  may **end** but not **begin**
  - If  $L:T$  then **end**  $L : [L]$
  - If  $L:T$  and  $P:e$  then **begin**  $L;P : e-[L]$
- Metaphor: **end**'s and **begin**'s like costs and benefits that must be balanced.
- **Goal:** assign a complete system the empty effect.



# Types for Symmetric Crypto

---

We include standard types such as dependent pairs and tagged unions

Messages of type **Un** are data known to the untyped opponent

Messages of type **Key(T)** are names used as symmetric keys for encrypting type **T**

If  $M:T$  and  $N:\text{Key}(T)$  then  $\{M\}_N:\text{Un}$

If  $M:\text{Un}$  and  $N:\text{Key}(T)$  and  $x:T \vdash P : e$ ,  
then **decrypt**  $M$  **as**  $\{x:T\}_N;P : e$

Messages of type **Nonce** $[L_1, \dots, L_n]$  prove begin-events labelled  $L_1, \dots, L_n$  have previously occurred



# Authenticity by Typing

A process  $P$  is **safe** iff in every execution trace, there is a distinct **begin**  $L$  for every **end**  $L$ .

(Formalizes one-to-one correspondences; can do one-to-many also)

A process  $P$  is **robustly safe** iff for all **begin**- and **end**-free opponents  $O$ ,  $P|O$  is safe.

## Theorem (Robust Safety)

If  $x_1, \dots, x_n : \mathbf{Un} \vdash P : []$  then  $P$  is robustly safe.

Corollary: example robustly safe via:

$\mathbf{Msg} \triangleq \mathbf{Un}$

$\mathbf{MyNonce}(m) \triangleq \mathbf{Nonce} [(\text{Sender sent } m)]$

$\mathbf{MyKey} \triangleq \mathbf{Key} (m : \mathbf{Msg}, \mathbf{MyNonce}(m))$

# Typing Ex I (Woo and Lam)

Event 1	$A$ begins	" $A$ proving presence to $B$ "
Message 1	$A \rightarrow B:$	$A$
Message 2	$B \rightarrow A:$	$NB$
Message 3	$A \rightarrow B:$	$\{\text{tag3}(B, NB)\}_{KAS}$
Message 4	$B \rightarrow S:$	$B, \{\text{tag4}(A, \{\text{tag3}(B, NB)\}_{KAS})\}_{KBS}$
Message 5	$S \rightarrow B:$	$\{\text{tag5}(A, NB)\}_{KBS}$
Event 2	$B$ ends	" $A$ proving presence to $B$ "

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3}(p) + \text{Cipher4}(p) + \text{Cipher5}(p))$

$\text{Cipher3}(A) \triangleq (B:\text{Un}, NB:\text{Nonce}["A \text{ proving presence to } B"])$

$\text{Cipher4}(B) \triangleq (A:\text{Un}, \text{cipher}:\text{Un})$  --seems redundant

$\text{Cipher5}(B) \triangleq (A:\text{Un}, NB:\text{Nonce}["A \text{ proving presence to } B"])$

# Typing Ex I, again

Event 1	A begins	"A proving presence to B"
Message 1	$A \rightarrow B:$	A
Message 2	$B \rightarrow A:$	NB
Message 3	$A \rightarrow B:$	$\{\text{tag3}(B, \text{NB})\}_{KAS}$
Message 4	$B \rightarrow S:$	$A, \{\text{tag3}(B, \text{NB})\}_{KAS}$
Message 5	$S \rightarrow B:$	$\{\text{tag5}(A, \text{NB})\}_{KBS}$
Event 2	B ends	"A proving presence to B"

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3}(p) + \text{Cipher5}(p))$

$\text{Cipher3}(A) \triangleq (B:\text{Un}, \text{NB}:\text{Nonce}[\text{"A proving presence to B"}])$

$\text{Cipher5}(B) \triangleq (A:\text{Un}, \text{NB}:\text{Nonce}[\text{"A proving presence to B"}])$

# Back to our SOAP protocol

Specify authenticity via event **correspondences**

Message 1	$A \rightarrow B:$	$w, \text{req}(\text{getNonce}())$
Message 2	$B \rightarrow A:$	$w, \text{res}(\text{getNonce}(\text{NB}))$
Event 1	$A$ begins	$\text{req}(A, B, w, M, t)$
Message 3	$A \rightarrow B:$	$A, \{\text{req}(w, M, t, \text{NB})\}_{K_{AB}}, \text{NA}$
Event 2	$B$ ends	$\text{req}(A, B, w, M, t)$
Event 3	$B$ begins	$\text{res}(A, B, w, N, t)$
Message 4	$B \rightarrow A:$	$B, \{\text{res}(w, N, t, \text{NA})\}_{K_{AB}}$
Event 4	$A$ ends	$\text{res}(A, B, w, N, t)$

Verify via generic types for crypto keys and nonces

$\text{SharedKey}(a, b) \triangleq \text{Key}(\text{Union}(\text{req}(w:\text{Un}, m:\text{Un}, t:\text{Un}, \text{Nb}:\text{Nonce}[\text{end req}(a, b, w, m, t)]), \text{res}(w:\text{Un}, n:\text{Un}, t:\text{Un}, \text{Na}:\text{Nonce}[\text{end res}(a, b, w, n, t)])))$

# Part VI: Validating our Abstraction



---

With Riccardo Pucella.

We formalize the application-level within an object calculus, and the SOAP-level within the spi-calculus.

The validation is a type-preserving semantics of the object calculus in the spi-calculus.





# A Calculus of Web Services

---

- Object calculi are OO-langs in miniature
  - Small enough for formal proof
  - Big enough for study of specific features
  - Abadi and Cardelli “A Theory of Objects”; Igarashi, Pierce, and Wadler FJ; Gordon and Syme BIL; ...
- We include an application-level view of a web service
  - A service is neither an object nor a value
    - WSDL neither object-oriented nor higher-order
  - But a service implemented via a server class
    - Recall the *BankingServiceClass*
  - And may be accessed directly or via a proxy class

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

$a, b ::=$	method body
$v$	value
<b>let</b> $x=a$ <b>in</b> $b$	let
<b>if</b> $u=v$ <b>then</b> $a$ <b>else</b> $b$	conditional
$v.f$	field lookup
$v.l(u_1, \dots, u_n)$	method call
$w.l(u_1, \dots, u_n)$	service call

$A, B ::=$	type
<b>Id</b>	principal
$c$	object
$\text{sig} ::= B(A_1 \ x_1, \dots, A_n \ x_n)$	signature

$x, y, z$	variable
$u, v ::=$	value
$x$	variable
<b>null</b>	null
<b>new</b> $c(v_1, \dots, v_n)$	object
$p$	principal

- For each class  $c \in \textit{Class}$ ,
  - map  $\textit{fields}(c)$  defines field names and types
  - map  $\textit{methods}(c)$  defines method names, signatures, and bodies
- For each service  $w \in \textit{WebService}$ ,
  - principal  $\textit{owner}(w)$  hosts the service
  - class  $\textit{class}(w)$  implements the service
    - constraint:  $\textit{fields}(w) = \textit{Id CallerId}$



# An Informal Semantics

- How to evaluate a body  $b$  as principal  $p$ :
  - To evaluate  $v$ , terminate with  $v$  at once
  - To evaluate **let**  $x=a$  **in**  $b\{x\}$ , first evaluate  $a$  as  $p$  to  $v$ , then evaluate  $b\{v\}$  as  $p$
  - To evaluate **if**  $u=v$  **then**  $a_{\text{true}}$  **else**  $a_{\text{false}}$ , evaluate  $a_{u=v}$  as  $p$
  - To evaluate  $v.f$ , when  $v=\text{new } c(v_1, \dots, v_n)$  and  $f$  is the  $i^{\text{th}}$  field of  $c$ , terminate with  $v_i$
  - To evaluate  $v.l(u_1, \dots, u_n)$ , when  $v=\text{new } c(v_1, \dots, v_n)$  and  $l$  in  $c$  has signature  $B(A_1 x_1, \dots, A_n x_n)$  and body  $b\{\text{this}, x_1, \dots, x_n\}$ , evaluate  $b\{v, u_1, \dots, u_n\}$  as  $p$
  - To evaluate  $w:l(u_1, \dots, u_n)$ , evaluate the method call **new**  $\text{class}(w)(p).l(u_1, \dots, u_n)$  as  $\text{owner}(w)$



# A Formal Semantics

- We map type  $B$  to spi message type  $\llbracket B \rrbracket$
- We map value  $v$  to spi message  $\llbracket v \rrbracket$
- We map body  $b$  running as  $p$  to spi process  $\llbracket b \rrbracket pk$  where  $k$  is a continuation channel – like encoding of lists
- We represent SOAP envelopes as spi messages
- We represent security guarantees by embedding begin- and end-assertions
- These security guarantees (that is, robust safety) follow as a corollary of type preservation

## Theorem (Type Preservation)

If  $E \vdash b : B$  then  $\llbracket E \rrbracket, k:\text{Ch}(\llbracket B \rrbracket) \vdash \llbracket b \rrbracket pk : []$



# Assessment

---

- Strengths
  - Developing implementation in parallel with theory – firm basis for answering “is this secure” – is there an exploit?
  - Developing high-level abstraction of SOAP processing – for sake of usability, security should be expressed at application level
- Weaknesses
  - Calculus stateless and sequential – too abstract to be useful?
  - Spec just message authentication, not correlation
  - Non-standard message formats, not WS-Security
  - Hiding details of SOAP message format – not clear which details are safe to omit
  - Cryptoc threat model does not include insider attacks – unrealistic in loosely coupled web services world
  - Model-based formal method – not directly checking actual source code, so gap between spec and implementation

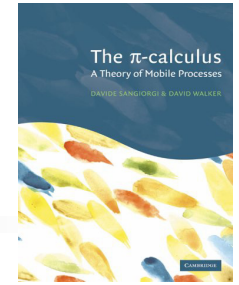
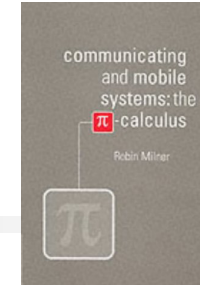


# Lecture 2: Summary

---

- Crypto protocols are hard to get right
  - Like programming Satan's not Murphy's computer
- Nominal calculi help by making protocol behaviour and intended properties explicit
- Type systems like Cryptyc give strong guarantees, without bounding principals or sessions
  - But annotations required, no insider attacks
- Application of Cryptyc to web services allows verification of design, but has weaknesses
  - Still, inspired current work, presented in Lectures 3 and 4

# Lecture 2: Resources



- Pi Calculus
  - R. Milner, *Communicating and Mobile Systems: the Pi-Calculus* (CUP, 1999)
  - D. Sangiorgi, D. Walker, *The Pi Calculus: A Theory of Mobile Systems* (CUP, 2003)
- Cryptyc
  - <http://cryptyc.cs.depaul.edu>
  - <http://research.microsoft.com/~adg/cryptyc.htm>
- Recent related work: Jif, Apollo, FlowCaml
  - <http://www.cs.cornell.edu/jif/>
  - <http://www.cis.upenn.edu/~stevez/sol/>
  - <http://cristal.inria.fr/~simonet/soft/flowcaml/>
- Critique of “attacker is the network” model
  - Eric Rescorla “The Internet is Too Secure Already”  
<http://www.rtfm.com/TooSecure-usenix.pdf>

End of  
Lecture 2





## 3: Modelling WS-Security

---

- An RPC protocol
  - Terms and predicates in TulaFale
  - Modelling the RPC messages
  - Processes and assertions in TulaFale
  - Modelling and verifying the protocol
- 
- K. Bhargavan, C. Fournet, and A. Gordon, *A Semantics for Web Services Authentication*. In 2004 ACM Symposium on Principles of Programming Languages, pp198-209, Venice, January 14-16, 2004.
  - K. Bhargavan, C. Fournet, A. Gordon, and R. Pucella, *TulaFale: A Security Tool for Web Services*. In International Symposium on Formal Methods for Components and Objects (FMCO 2003), Leiden. Springer LNCS, Revised Lectures, 2004.



# Samoa Project: Summary

---

- If misconfigured or mis-implemented, WS-Security protocols vulnerable to XML rewriting attacks
  - We found such attacks on code that uses MS WSE toolkit
- TulaFale tool can show the absence of such attacks given a description of the protocol
  - First analysis tool for XML-based crypto protocols
  - Automatic analysis of hand-written models by appeal to Blanchet's ProVerif tool for applied pi
- Generator and Analyzer tools compile TulaFale scripts from declarative XML policy files that drive WSE 2.0
  - Hence, can directly analyze WSE 2.0 configurations
  - We believe to be first source-based formal verification of interoperable implementation of crypto protocols

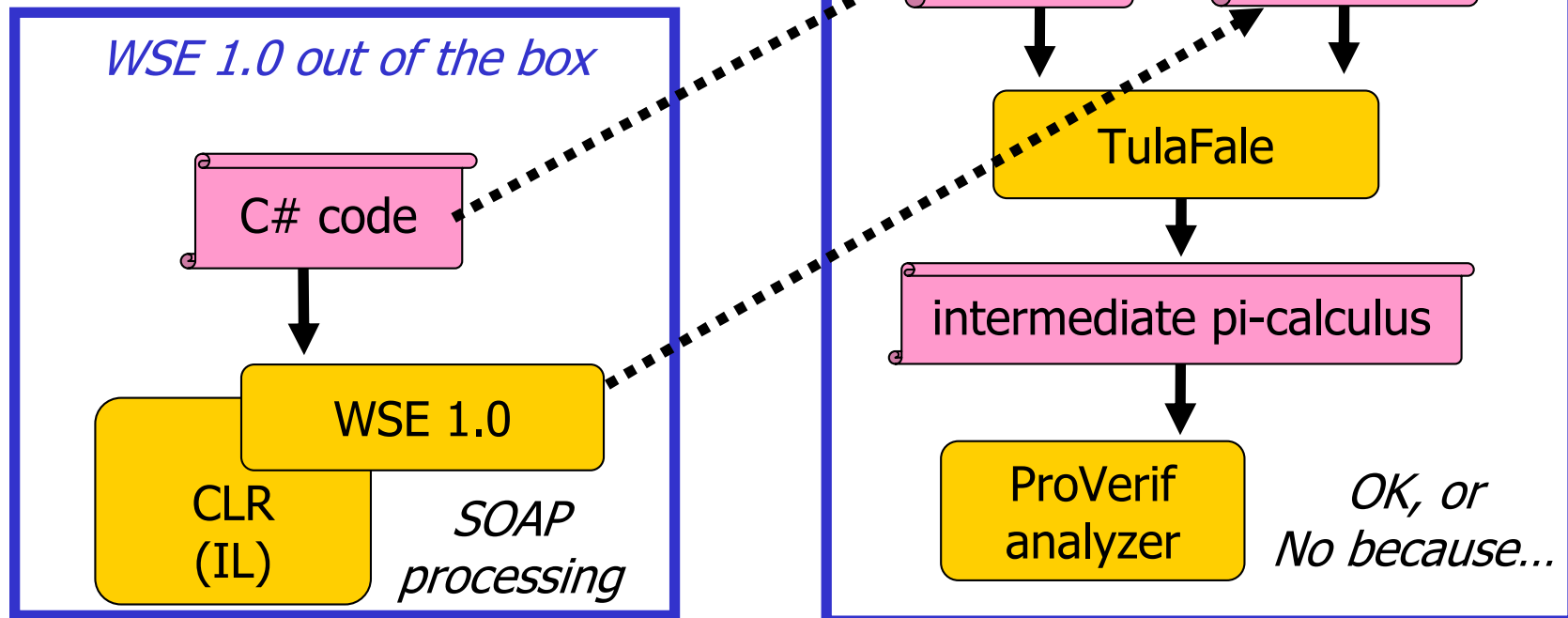
**This  
lecture**

**Next  
lecture**

# TulaFale: A Security Tool for WS

TulaFale = pi + XML + predicates + assertions

In work published at FMCO'03 and POPL'04, we designed and implemented TulaFale, and hand-wrote models for series of WSE protocols





# Part I: An RPC Protocol

---

We describe an RPC protocol as easily coded using the WSE implementation of WS-Security



# Model and Goals

---

- A typical system model:
  - A single certification authority (CA) issuing X.509 public-key certificates for services, signed with the CA's private key.
  - Two servers, each equipped with a public key certified by the CA and exporting an arbitrary number of web services
  - Multiple clients, acting on behalf of human users
- Threat model: an active attacker, in control of network, but knowing none of:
  - The private key of the CA
  - The private key of any public key certified by the CA
  - The password of any user in the database
- Security goals: authentication of each message, and correlation of request and response, but not confidentiality



# Client and Server Processes

Client(kr,U)

kr = *public key of CA*

U = <User>

<Username>u</>

<Password>pwd</></>

Server(sx,cert,S)

sx = *subject's private key*

cert = *X509 certificate,*

*signed by kr,*

*that subj owns sx*

S = <Service>

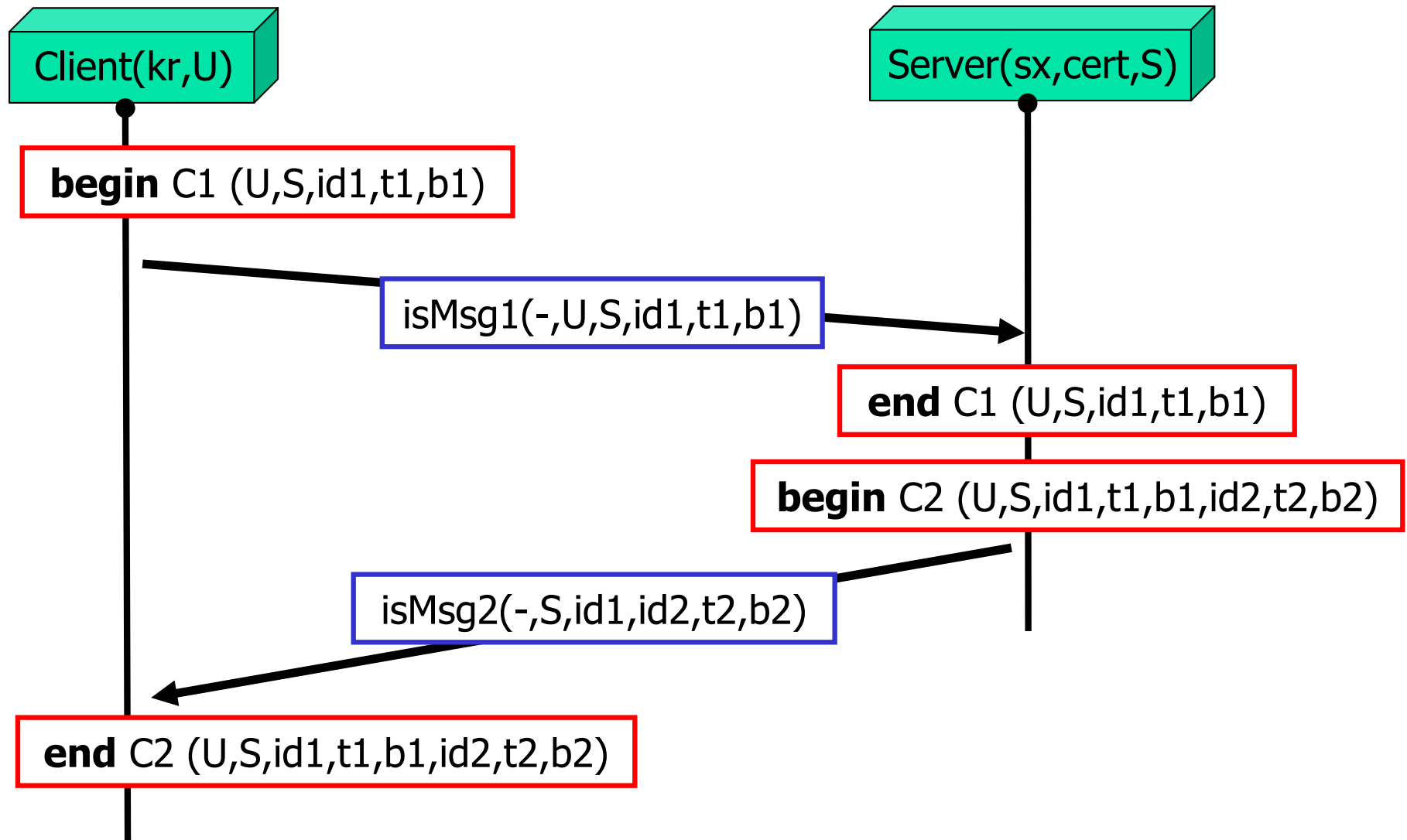
<To>uri</>

<Action>ac</>

<Subject>subj</></>

Our system model includes arbitrarily many such clients and servers;  
in fact, attacker can control how many are generated

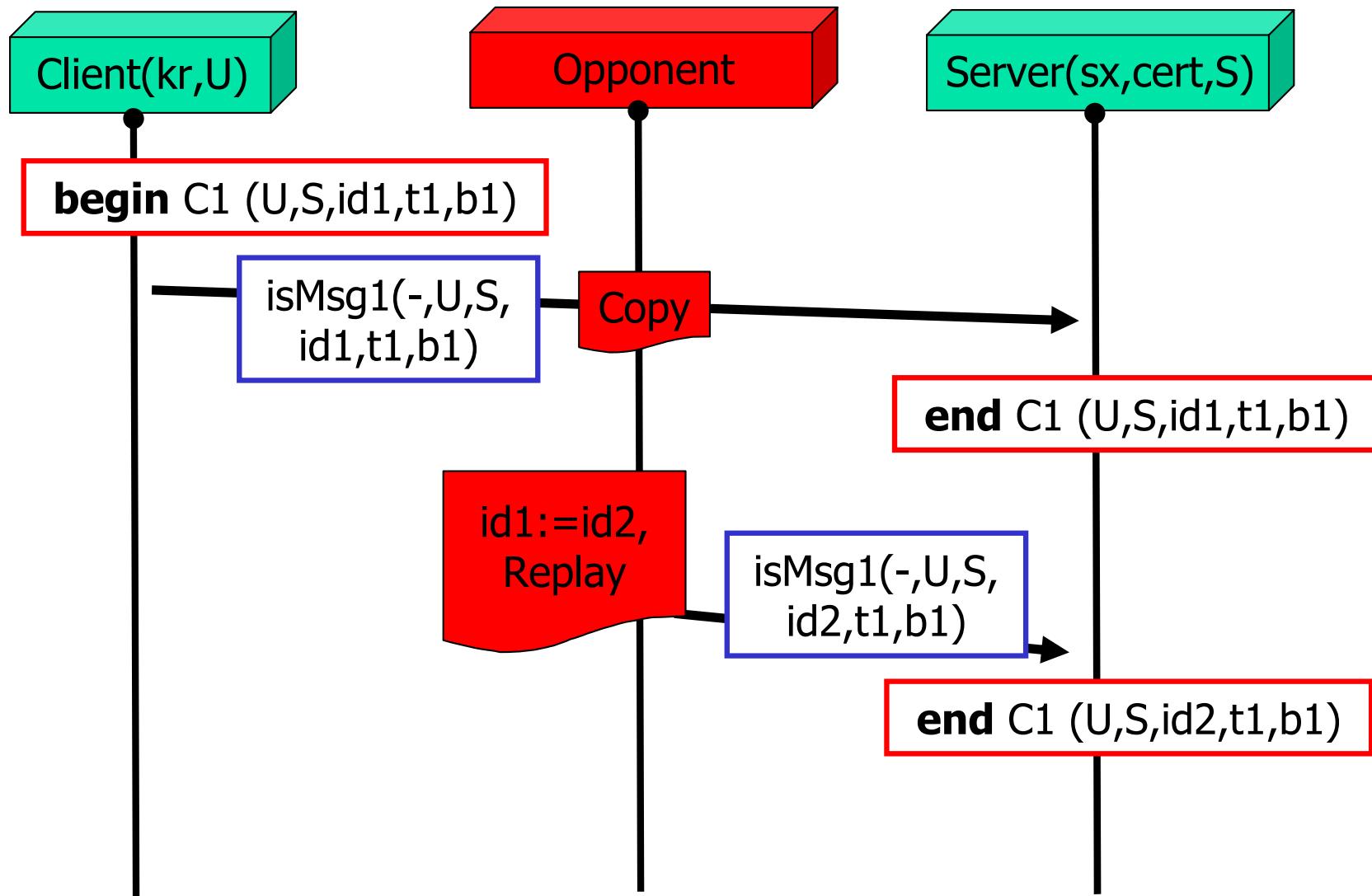
*An intended run of the protocol*



Msg 1 includes signature of  $S, id1, t1, b1$  under key derived from username token for  $U$

Msg 2 includes signature of  $id1, id2, t2, b2$  under public key of  $S$

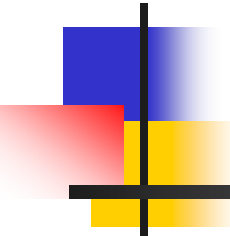
*Suppose a client does not sign the message identifier id1...*



Pair (id1,t1) uniquely identifies the message only if id1 and t1 are signed

We found and fixed faults like this in preliminary WSE samples





# Part II: Terms and Predicates in TulaFale

---

TulaFale = pi + XML + predicates + assertions



# XML Elements and Attributes

## Terms:

```
term ::=
  string                // literal string
  ide                   // variable
  "_"                  // anonymous var
  "<" ide termList ">" termList "</>" // XML element
  ide "=" term          // XML attribute
termList ::= term1 ... termn [ "@" term ]
```

- Represents valid, parsed XML
- Adapted from Siméon and Wadler's model (POPL'03)
- Resembles the W3C Infoset recommendation



# Ex: Outline of Message 1

---

```
<Envelope>
  <Header>
    <To>uri</>
    <Action>ac</>
    <MessageId>id</>
    <Security>
      <Timestamp><Created>"2004-03-19T09:46:32Z"</></>
      utok
      sig
    </>
  </>
  <Body Id="1">request</>
</>
```

- As in Lecture 1: no namespaces, all string data quoted, trailing tags omitted



# Sorts for Terms

---

## Sorts:

```
sort ::=
  "string"      // XML string (subsort of "item")
  "item"        // either a string or an element
  "items"       // finite sequence of items
  "att"         // XML attribute
  "atts"        // finite sequence of attributes
  "bytes"       // symbolic byte array (not XML)
```



# Function Symbols

## Additional Term for Function Applications:

```
term ::= ide "(" term1 "," ... "," termn ")"
```

**constructor** `utf8(string):bytes`.

**destructor** `iutf8(bytes):string` with `iutf8(utf8(x))=x`.

**constructor** `c14n(item):bytes`.

**destructor** `ic14n(bytes):item` with `ic14n(c14n(x))=x`.

**constructor** `concat(bytes,bytes):bytes`.

**destructor** `fst(bytes):bytes` with `fst(concat(a,b)) = a`.

**destructor** `snd(bytes):bytes` with `snd(concat(a,b))=b`.

**constructor** `base64(bytes):string`.

**destructor** `ibase64(string):bytes` with `ibase64(base64(x))=x`.

The equations associated with destructors induce an equational theory on terms

Destructors typically invoked by pattern matching



# Function Symbols for Crypto

constructor **sha1**(bytes):bytes.  
constructor **psha1**(string,bytes):bytes.  
constructor **hmacsha1**(bytes,bytes):bytes.

No destructors for  
hash functions, etc

constructor **aes**(bytes,bytes):bytes.  
destructor **decaes**(bytes,bytes):bytes with  $\text{decaes}(k, \text{aes}(k, b)) = b$ .

constructor **pk**(bytes):bytes.  
constructor **rsasha1**(bytes,bytes):bytes.  
destructor **checkrsasha1**(bytes,bytes,bytes):bytes with  
 $\text{checkrsasha1}(\text{pk}(k), x, \text{rsasha1}(k, x)) = \text{pk}(k)$ .  
constructor **rsa**(bytes,bytes):bytes.  
destructor **decrsa**(bytes,bytes):bytes with  $\text{decrsa}(k, \text{rsa}(\text{pk}(k), b)) = b$ .

pk(s) is public key  
corresponding to  
the private key s

As often with Dolev-Yao formalisms, not attempting  
to capture all algebraic properties of RSA



# Logical Predicates

## Predicates and Formulas:

```
declaration ::=
  "predicate" ide "(" sortingTuple ")" ":-" formula "."
formula ::=
  formula "," formula           // conjunction
  term "=" term                 // term equality
  ide "in" term                 // list membership
  ide "(" termTuple ")"         // named predicate
sorting ::= ide ":" sort
sortingTuple ::= sorting1 "," ... "," sortingn
```

- Given certain implementability constraints, these logic programs given formal semantics in Abadi and Fournet's applied pi calculus (see our POPL'04 paper)



# Ex: RSA Encryption

---

```
predicate mkEncryptedData (encrypted:item,plain:item,ek:bytes) :—  
    cipher = rsa(ek,c14n(plain)),  
    encrypted = <EncryptedData>  
                <CipherData>  
                <CipherValue>base64(cipher)</></></>.
```

```
predicate isEncryptedData (encrypted:item,plain:item,dk:bytes) :—  
    encrypted = <EncryptedData>  
                <CipherData>  
                <CipherValue>base64(cipher)</></></>,  
    c14n(plain) = decrsa(dk,cipher).
```

Direct RSA encryption and decryption,  
illustrating computation by pattern-matching





# Ex: X509 Certs and Tokens

**constructor** x509(bytes,string,string,bytes):bytes.

**destructor** x509key(bytes):bytes with x509key(x509(s,u,a,k))=k.

**destructor** checkx509(bytes,bytes):bytes with

checkx509(x509(s,u,a,k),pk(s))=pk(s).

**destructor** x509user(bytes):string with x509user(x509(s,u,a,k))=u.

**destructor** x509alg(bytes):string with x509alg(x509(s,u,a,k))=a.

**predicate** isX509Cert (xcert:bytes,kr:bytes,u:string,a:string,k:bytes) :—

checkx509(xcert,kr) = kr,

u = x509user(xcert), k = x509key(xcert), a = x509alg(xcert).

**predicate** isX509Token (tok:item,kr:bytes,u:string,a:string,k:bytes) :—

tok = <BinarySecurityToken ValueType="X509v3">base64(xcert)</>,

isX509Cert (xcert,kr,u,a,k).



# Ex: Username Tokens

```
predicate isUser (U:item,u:string,pwd:string) :—  
    U = <User><Username>u</><Password>pwd</></>.
```

Entry in an  
internal  
database,  
never sent on  
the wire

```
predicate isUserTokenKey (tok:item,U:item,n:bytes,t:string,k:bytes) :—  
    isUser(U,u,pwd),  
    tok = <UsernameToken>  
        <Username> u </>  
        <Nonce> base64(n) </>  
        <Created> t </> </>,  
    k = pshal(pwd,concat(n,utf8(t))).
```

WS-Security  
security token  
signifying a  
user entry

# Recall Structure of Signatures

```
<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=1>
        <Username>"adg"
        <Nonce>"mTbzQM84RkFqza+IIes/xw=="
        <Created>"2004-09-01T13:31:50Z"
      <Signature>
        <SignedInfo>
          <SignatureMethod Algorithm=hmac-sha1>
          <Reference URI=#2>
            <DigestValue>"U9sBHidIkVvKA4vZo0gGKxMhA1g="
          <SignatureValue>"8/ohMBZ5JwzYyu+POU/v879R01s="
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI=#1 ValueType=UsernameToken>
          <Body Id=2>
            <StockQuoteRequest>
              <symbols>
                <Symbol>"FABRIKAM"
                <Symbol>"CONTOSO"
```

**UsernameToken** assumes  
both parties know adg's  
secret password  $p$

Each **DigestValue**  
is the sha1 hash of  
the URI target

$hmacsha1(key, \text{SignedInfo})$  where  
 $key \approx psha1(p + nonce + created)$



# Ex: Document Refs

```
<SignedInfo>
  <Reference URI="#..."><DigestValue>Ego0...</>
  <Reference URI="#..."><DigestValue>5GHI...</>
  <Reference URI="#..."><DigestValue>efb0...</>
```

- The group bound together in a signature is given by a finite sequence of references
  - URI points to an item  $t$ , typically a node in the envelope
  - DigestValue is a secure hash of the item
- $ref(t, r)$  means that  $r$  is such a reference to  $t$   
**predicate**  $ref(t:item, r:item) :-$   
 $r = \text{<Reference \_> \_ \_<DigestValue> base64(sha1(clean(t))) </> </> .}$
- When checking a signature, we know what's to be signed; the URI attribute is an untrusted processing hint



# Formalizing Signatures

```
<Signature>
  <SignedInfo>
    <Reference URI="#..."><DigestValue>dFGb...</>...
    <SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
    <KeyInfo><SecurityTokenReference><Reference URI="#..."/>
```

```
predicate isSigVal (sv:bytes,si:item,k:bytes,a:string) :—
  "hmacsha1" = a, sv = hmacsha1(k,c14n(si)).
```

```
predicate isSigVal (sv:bytes,si:item,p:bytes,a:string) :—
  "rsasha1" = a, p = checkrsasha1(p,c14n(si),sv).
```

```
predicate isSigInfo (si:item,a:string,ts:item) :—
  si = <SignedInfo> _ <SignatureMethod Algorithm=a> </> @ refs</>,
  rs = <list>@ refs</>, isRefs(ts,rs).
```

```
predicate isSignature (sig:item,a:string,k:bytes,ts:item) :—
  sig = <Signature> si <SignatureValue> base64(sv) </> @ _ </>,
  isSigInfo(si,a,ts), isSigVal(sv,si,k,a).
```

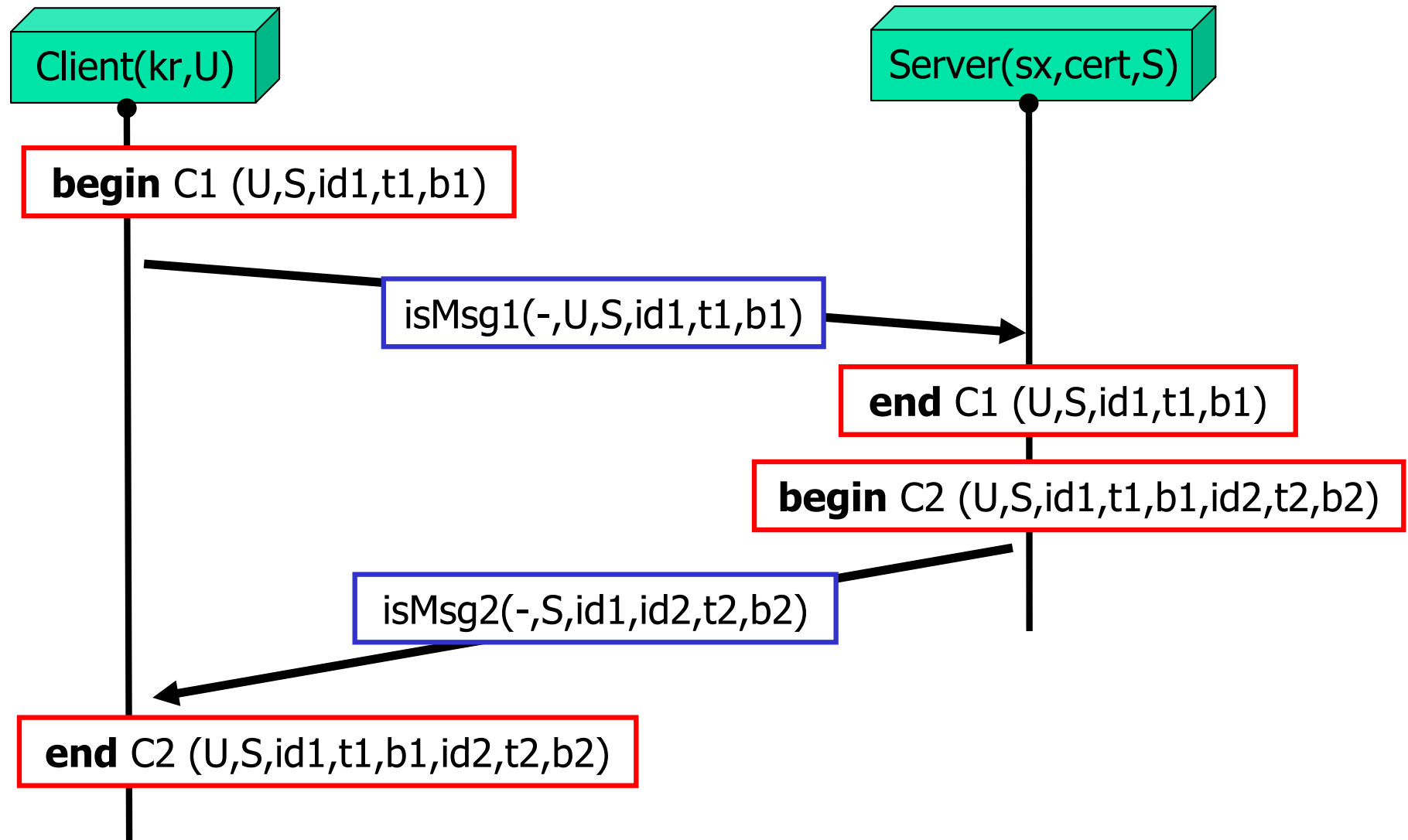
# Part III: Modelling the RPC Messages



---

Constructing and checking XML messages by logic programming

The protocol again...



Msg 1 includes signature of  $S, id1, t1, b1$  under key derived from username token for  $U$

Msg 2 includes signature of  $id1, id2, t2, b2$  under public key of  $S$



# Goal C1

---

Our goal **C1** is to reach agreement on the data

$(U, S, id1, t1, b1)$

where

$U = \langle \text{User} \rangle \langle \text{Username} \rangle u \langle / \rangle \langle \text{Password} \rangle \text{pwd} \langle / \rangle \langle / \rangle$

$S = \langle \text{Service} \rangle \langle \text{To} \rangle \text{uri} \langle / \rangle \langle \text{Action} \rangle \text{ac} \langle / \rangle \langle \text{Subject} \rangle \text{subj} \langle / \rangle \langle / \rangle$

after receiving and successfully checking Message 1.





# Structure of Message 1

```
predicate env1(msg1:item,uri:item,ac:item,id1:string,t1:string,  
               eutok:item,sig1:item,b1:item) :—
```

```
msg1 =
```

```
<Envelope>
```

```
<Header>
```

```
<To>uri</>
```

```
<Action>ac</>
```

```
<MessageId>id1</>
```

```
<Security>
```

```
<Timestamp><Created>t1</></>
```

```
eutok
```

```
sig1</></>
```

```
<Body>b1</></>.
```

Single predicate  
used in two  
different modalities  
to construct and  
parse Message 1

**TulaFale Wishlist:**  
syntax to indicate  
modes of parameters...



# Making Message 1

```
predicate mkMsg1(msg1:item,U:item,S:item,kr:bytes,cert:bytes,  
                n:bytes,id1:string,t1:string,b1:item) :—  
    S = <Service><To>uri</> <Action>ac</> <Subject>subj</></>,  
    isX509Cert(cert,kr,subj,"rsasha1",ek),  
    isUserTokenKey(utok,U,n,t1,sk),  
    mkEncryptedData(eutok,utok,ek),  
    mkSignature(sig1,"hmacsha1",sk,  
        <list>  
            <Body>b1</>  
            <To>uri</>  
            <Action>ac</>  
            <MessageId>id1</>  
            <Created>t1</>  
            eutok</>),  
    env1(msg1,uri,ac,id1,t1,eutok,sig1,b1).
```

Username token  
encrypted to prevent  
dictionary attacks on  
weak passwords



# Checking Message 1

---

```
predicate isMsg1(msg1:item,U:item,sx:bytes,cert:bytes,S:item,
                id1:string,t1:string,b1:item) :-
    env1(msg1,uri,ac,id1,t1,eutok,sig1,b1),
    S = <Service><To>uri</> <Action>ac</> <Subject>subj</></>,
    isEncryptedData(eutok,utok,sx),
    isUserTokenKey(utok,U,n,t1,sk),
    isSignature(sig1,"hmacsha1",sk,
        <list>
            <Body>b1</>
            <To>uri</>
            <Action>ac</>
            <MessageId>id1</>
            <Created>t1</>
            eutok</>).
```



# Goal C2

---

Our goal **C2** is to reach agreement on the data

$(U, S, id1, t1, b1, id2, t2, b2)$

where

$U = \langle \text{User} \rangle \langle \text{Username} \rangle u \langle / \rangle \langle \text{Password} \rangle \text{pwd} \langle / \rangle \langle / \rangle$

$S = \langle \text{Service} \rangle \langle \text{To} \rangle \text{uri} \langle / \rangle \langle \text{Action} \rangle \text{ac} \langle / \rangle \langle \text{Subject} \rangle \text{subj} \langle / \rangle \langle / \rangle$

after successful receipt of Message 2, having already agreed on

$(U, S, id1, t1, b1)$

after receipt of Message 1.



# Structure of Message 2

```
predicate env2(msg2:item,uri:item,id1:string,id2:string,
               t2:string,cert:bytes,sig2:item,b2:item) :-
    msg2 =
        <Envelope>
            <Header>
                <From>uri</>
                <RelatesTo>id1</>
                <MessageId>id2</>
                <Security>
                    <Timestamp><Created>t2</></>
                    <BinarySecurityToken>base64(cert)</>
                    sig2</></>
                <Body>b2</></>.
```



# Checking Message 1

---

```
predicate isMsg2(msg2:item,S:item,kr:bytes,  
                id1:string,id2:string,t2:string,b2:item) :—  
  env2(msg2,uri,id1,id2,t2,cert,sig2,b2),  
  isService(S,uri,ac,subj),  
  isX509Cert(cert,kr,subj,"rsasha1",k),  
  isSignature(sig2,"rsasha1",k,  
    <list>  
      <Body>b2</>  
      <RelatesTo>id1</>  
      <MessageId>id2</>  
      <Created>t2</></>).
```



# Part IV: Processes and Assertions in TulaFale

---

TulaFale = **pi** + XML + predicates + **assertions**



# Processes and Assertions

## Processes and Assertions:

```
process ::=
  "0"
  "in" ide "(" ideTuple ")" [ ";" process ]
  "out" ide "(" termTuple ")" [ ";" process ]
  "new" "(" sorting ")" ";" process
  process "|" process
  "!" process
  "let" ide "=" term ";" process
  "filter" formula "->" sortingTuple ";" process
  ide "(" termTuple ")"
  "begin" ide "(" termTuple ")" [ ";" process ]
  "end" ide "(" termTuple ")" [ ";" process ]
  "done"
```





# Part V: Modelling and Verifying the Protocol

---

The top-level process and its verification



# System and Attacker

```
new sr:bytes; let kr = pk(sr);
new sx1:bytes; let cert1 = x509(sr,"BobsPetShop","rsasha1",pk(sx1));
new sx2:bytes; let cert2 = x509(sr,"ChasMarket","rsasha1",pk(sx2));
out publish(base64(kr));
out publish(base64(cert1));
out publish(base64(cert2));
( !MkUser(kr) | !MkService(sx1,cert1) | !MkService(sx2,cert2) |
  (!in anyUser(U); Client(kr,U)) |
  (!in anyService(sx,cert,S); Server(sx,cert,S)) )
```

- The attacker is an arbitrary process running alongside this system
- It can send and receive on the soap channel
- Moreover, it can generate arbitrarily many users and services, and initiate arbitrarily many sessions with payload data of its choosing



# Attacker Generates Users

---

```
channel genUser(string).  
private channel anyUser(item).  
process MkUser() =  
  in genUser(u);  
  new pwd:string;  
  let U = <User><Username>u</><Password>pwd</></>;  
  !out anyUser (U).
```



# Attacker Generates Services

---

```
predicate isServiceData(S:item,sx:bytes,cert:bytes) :—  
    isService(S,uri,ac,x509user(cert)),  
    pk(sx) = x509key(cert).
```

```
channel genService(item).  
private channel anyService(bytes,bytes,item).  
process MkService(sx:bytes,cert:bytes) =  
    in genService(S);  
    filter isServiceData(S,sx,cert) → ;  
    !out anyService(sx,cert,S).
```



# Attacker Initiates Clients

---

```
channel init(item,bytes,bytes,string,item).
process Client(k:bytes,U:item) =
  in init (S,certA,n,t1,b1);
  new id1:string;
  begin C1 (<list>U S id1 t1 b1</>);
  filter mkMsg1(msg1,U,S,k,certA,n,id1,t1,b1) → msg1;

  out soap(msg1);
  in soap(msg2);
  filter isMsg2(msg2,S,k,id1,id2,t2,b2) → id2,t2,b2;
  end C2 (<list>U S id1 t1 b1 id2 t2 b2</>);
done.
```



# Attacker Initiates Servers

---

```
channel accept(string,string,item).  
process Server(sx:bytes,cert:bytes,S:item) =  
  in soap(msg1);  
  in anyUser(U);  
  filter isMsg1(msg1,U,sx,cert,S,id1,t1,b1) → id1,t1,b1;  
  end C1 (<list>U S id1 t1 b1</>);  
  
  in accept (id2,t2,b2);  
  filter mkMsg2(msg2,sx,cert,S,id1,id2,t2,b2) → msg2;  
  begin C2 (<list>U S id1 t1 b1 id2 t2 b2</>);  
  out soap(msg2).
```

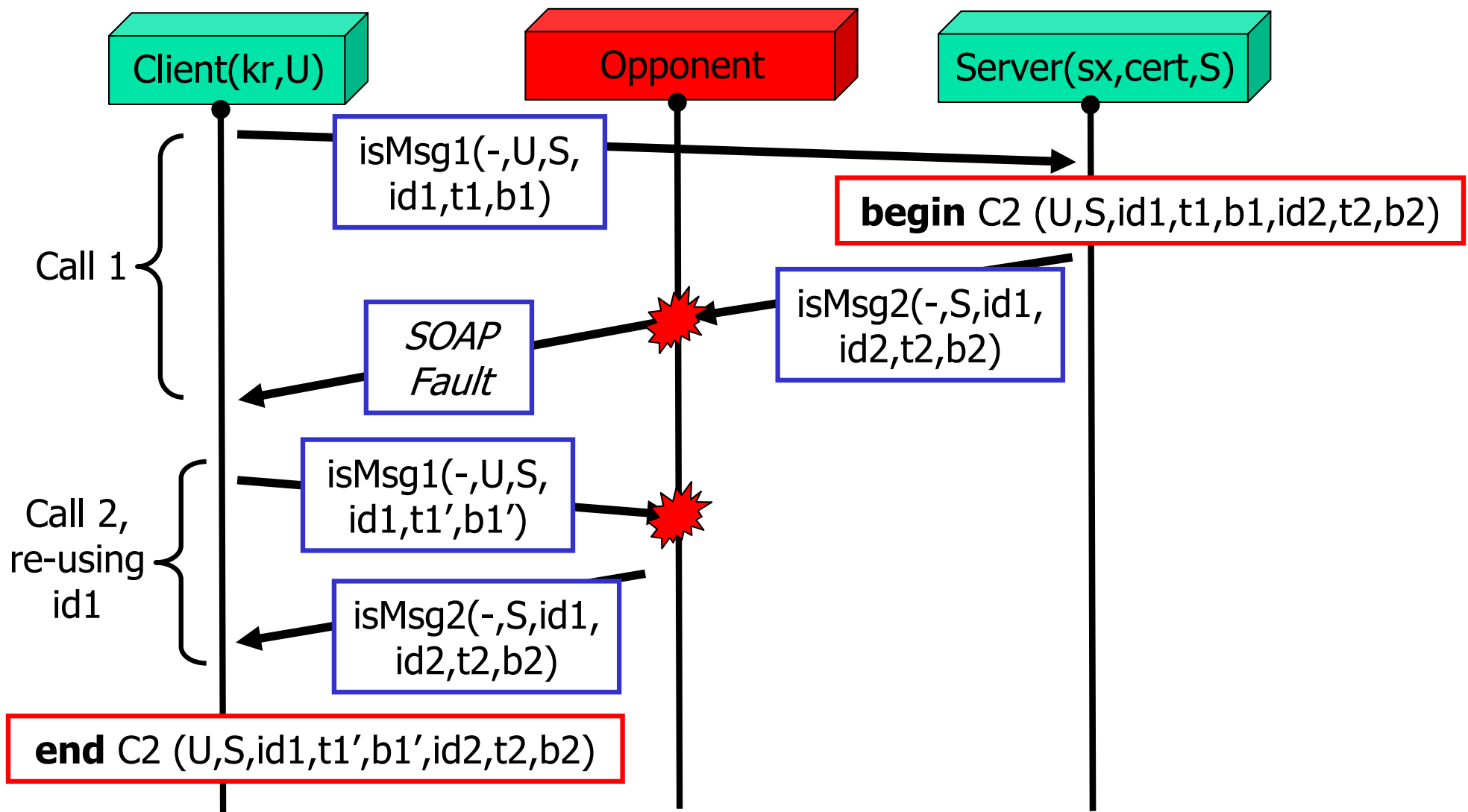


---

Verification of C1 and C2 by TulaFale/ProVerif

Next: some variations.

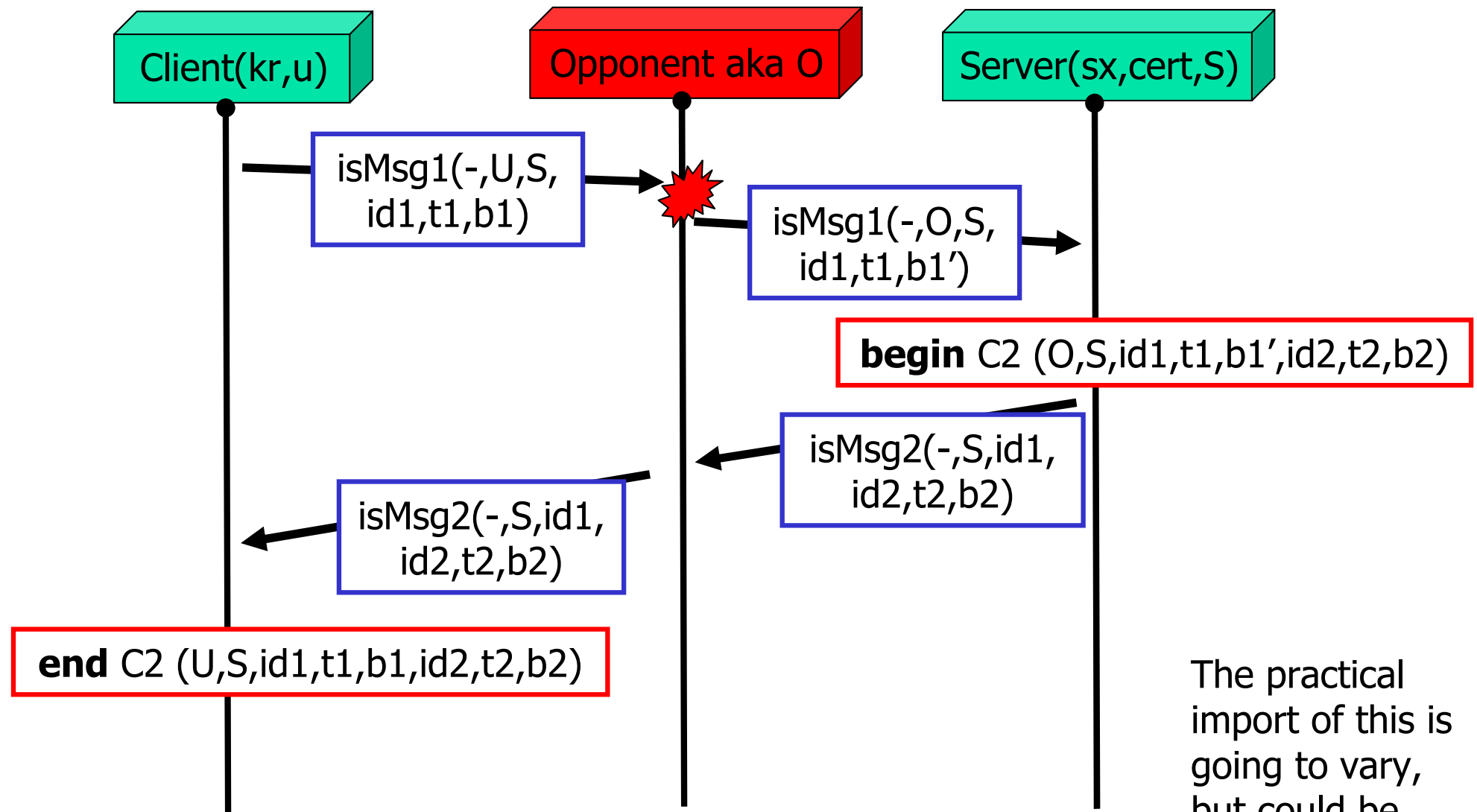
*What else might go wrong?*



If the client doesn't generate fresh id1's, then message correlation (C2) fails; the tools easily find this bug



*What about insider attacks?*



If one or more passwords are compromised, there is an insider attack on message correlation; more extensive changes to the script are needed to model this

The practical import of this is going to vary, but could be significant; eg suppose b2 is a credit history



# Lecture 3: Summary

---

- Successfully bridged gap between theoretical pi threat model and XML used in WS-Security protocols
  - Driven by actual user code
  - Faithful to XML message format
  - Found attacks within threat model
  - Proved theorems about wire-level protocols
  - Exploiting research on automated analysis
- Future work: interoperability testing between TulaFale and WSE
  - Would increase confidence in accuracy of our modelling – some non-critical (?) details are currently missing



# Lecture 3: Resources

---

- Projects: Samoa, Proverif
  - <http://Securing.WS>
  - <http://www.di.ens.fr/~blanchet/crypto-eng.html>

End of  
Lecture 3



# 4: Modelling Further Specs

---

- Analyzing WS-Trust and WS-SecureConversation
- Interlude: model-based versus source-based formal methods
- Generating and Analyzing WS-Policy



# Part I: Analyzing WS-Trust and WS-SecureConversation

---

K. Bhargavan, R. Corin, C. Fournet, A. D. Gordon, *Secure Sessions for Web Services*, submitted for publication (2004)

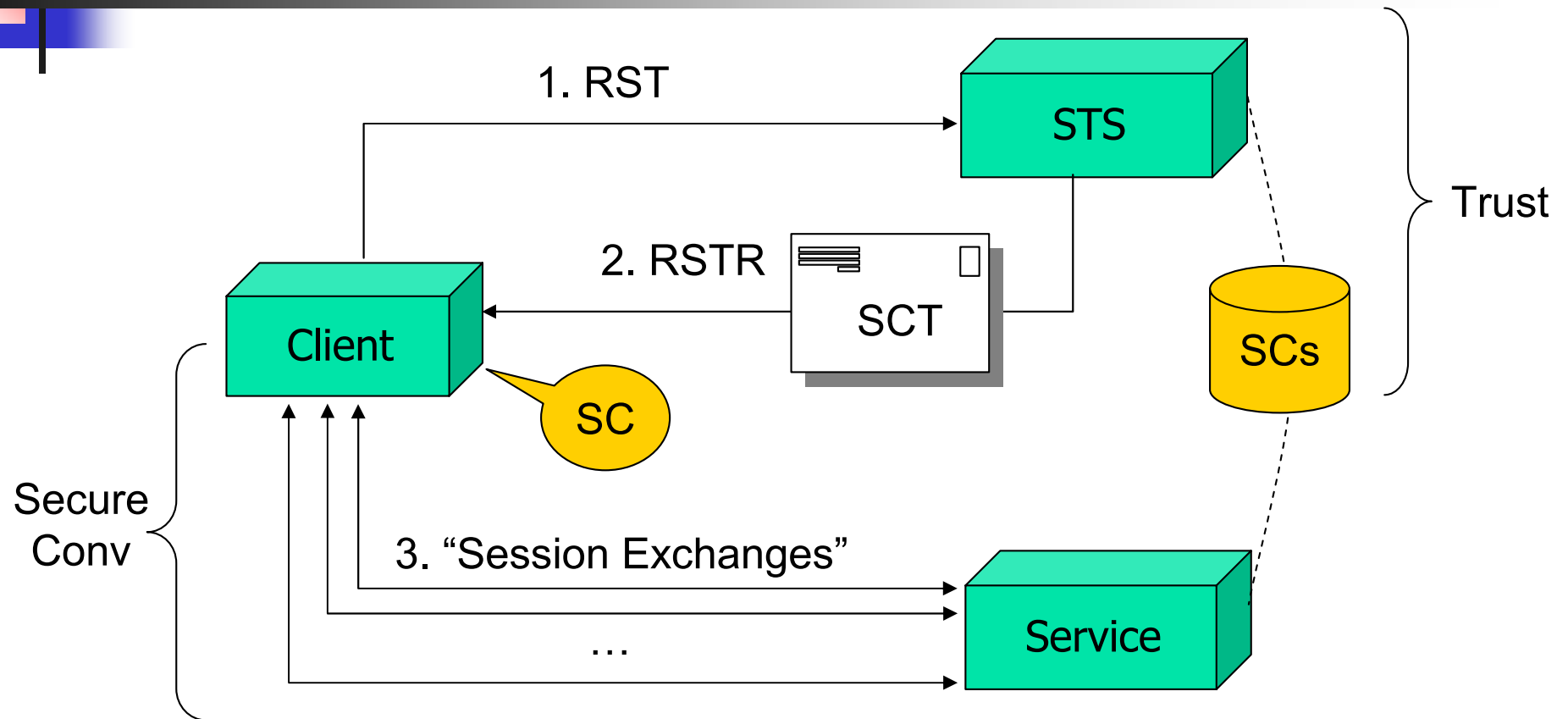


# Motivation

---

- WS-Security provides basic mechanisms to secure SOAP traffic, one message at a time
  - Signing and encryption keys derived from long-lived secrets like passwords or private keys
- If a SOAP interaction consists of multiple, related messages, WS-Security alone may be inefficient, and does not secure session integrity
  - Standard idea: establish short-lived session key
- Recent specs describe this idea at the SOAP-level
  - WS-SecureConversation defines *security contexts*, used to secure sessions between two parties
  - WS-Trust defines how security contexts are issued and obtained

# A Typical Scenario



STS = Security Token Server  
RST = Request Security Token  
RSTR = RST Response

SC = Security Context  
SCT = SC Token





# Our Analysis

---

- We develop TulaFale models of some typical usages of Trust and SecureConversation
- The models attempt to reflect the (partial) WSE implementation of these specs
  - Specs open-ended, so impossible to model completely
- We state and prove a series of core security properties for these protocols
  - TulaFale/ProVerif combination of XML syntax and automation is very effective
  - First formal analysis of these specs
- Found some limitations, and provided feedback to spec writers and WSE team



# System Model

---

- We assume the following participants:
  - A single certification authority (CA), with keypair  $kr/sr$
  - Multiple principals, each identified by a username  $u$ , and equipped with passwords or X.509 certs issued by CA.
- We assume a single trusted database (a private channel) that relates users to passwords or private keys
- Client and server processes acting on behalf of users can access this database, but not the attacker
- Principal and key compromise:
  - A principal is *safe* if none of its secrets has been leaked to the attacker; otherwise, *unsafe*
  - A security context is *safe* if it has not been leaked to the attacker; otherwise, it is *unsafe*



# Attacker Model

---

- Our system model allows the attacker:
  - To send and receive on the soap channel
  - To generate fresh passwords or certs for any principal
  - To initiate sessions and to choose parameters of clients and servers
  - To cause the leak of passwords or certificates for any principal (but not the CA)
  - To cause the leak of security contexts
- This amounts to a realistic threat model for XML rewriting attacks on web services
  - We did not consider leaks of secrets in Lecture 3
- (Additionally, one needs to consider other classes of attacks eg SQL injection, etc)

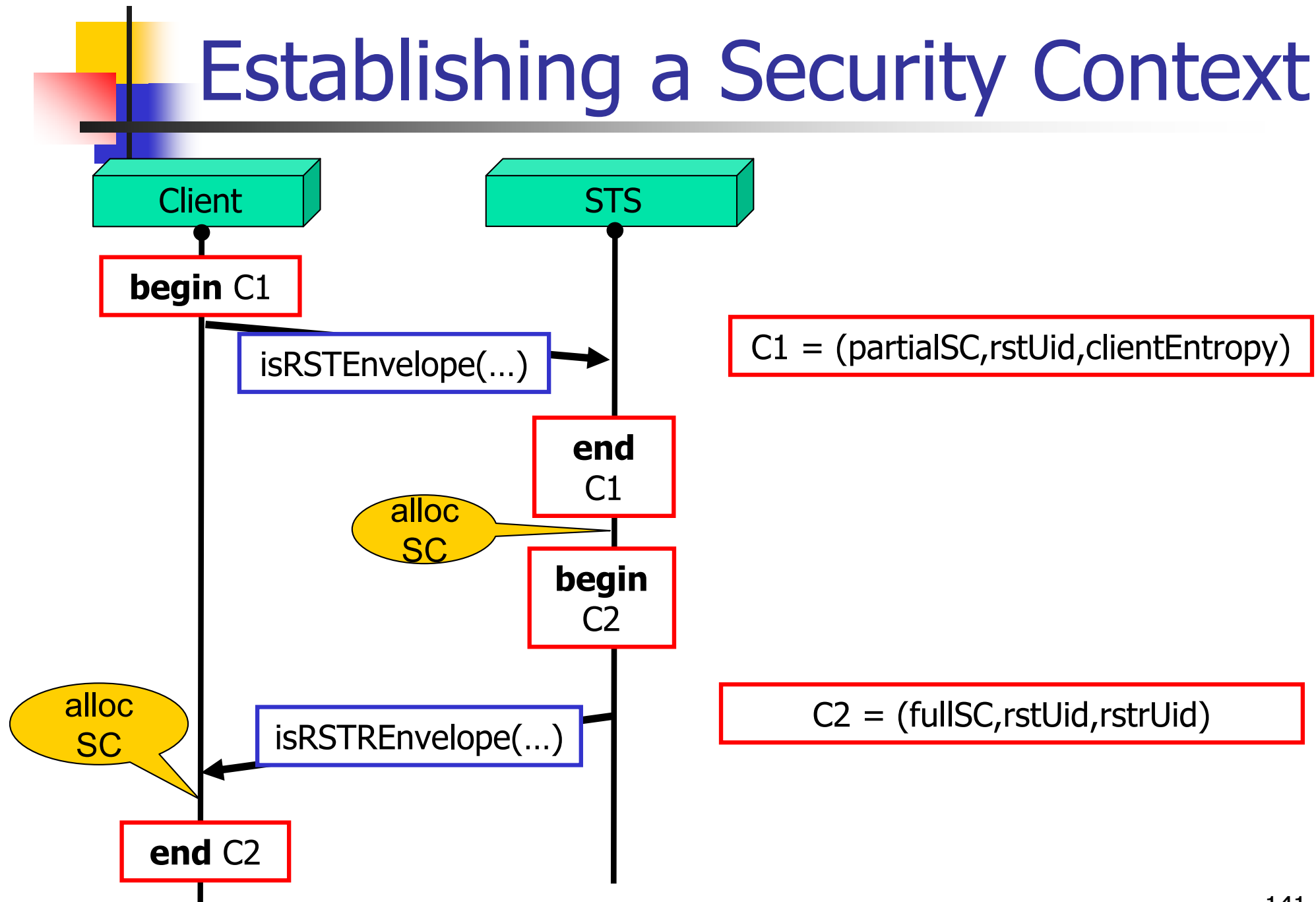


# WS-Trust

---

- C. Kaler, A. Nadalin et al, Web Services Trust Language (WS-Trust) Version 1.1 (May 2004)
  - <http://msdn.microsoft.com/ws/2004/04/ws-trust/>
- “provides a framework for requesting and issuing security tokens, and to broker trust relationships”
- Deliberatively abstract to allow flexibility, but does define a precise syntax for RST/RSTR messages
  - No specs define syntax for the SC itself, however
- For key establishment, two modes:
  - “Entropic”: client and STS provide randomness for SC key
  - “NonEntropic”: only the STS creates the key

# Establishing a Security Context





# Security Context After RST

```
predicate PartialSC(sc:item,mode:string,UserToken:item,  
                   StsInfo:item,appTo:string,extra:item) :—
```

```
sc = <SecurityContext>  
    <Base>UserToken</>  
    <STSInfo>StsInfo</>  
    <AppliesTo>appTo</>  
    <EntropyMode>mode</>  
    <ExtraInfo>extra</>  
</>.
```

UserToken, StsInfo – either  
UsernameToken or X509Token  
for client and STS respectively

appTo – URI for the service  
mode – either “both” or “server”  
extra – additional information eg  
lifetime of the context



# Security Context After RSTR

```
predicate SC(sc:item,sctid:string,sckey:bytes,mode:string,UserToken:item,  
            StsInfo:item,appTo:string,extra:item) :—
```

```
sc = <SecurityContext>  
    <Identifier>sctid</>  
    <Key>base64(sckey)</>  
    <Base>UserToken</>  
    <STSInfo>StsInfo</>  
    <AppliesTo>appTo</>  
    <EntropyMode>mode</>  
    <ExtraInfo>extra</></>.
```

UserToken, StsInfo – either  
UsernameToken or X509Token  
for client and STS respectively

appTo – URI for the service  
mode – either “both” or “server”  
extra – additional information eg  
lifetime of the context

sctid – public id of the SC

sckey – session key



# WS-Trust Security Results

**Theorem** (*Robust Safety of the Security Context*) For all runs of the script in the presence of an active attacker (hence all choices of modes):

- For each **end** *C1* with a safe client, there is a matching **begin** *C1*.
- For each **end** *C2* with safe client and STS, there is a matching **begin** *C2*.

**Theorem** (*Session-Key Secrecy*) For all runs of the script in the presence of an active attacker (and hence all choices of operation mode), for each **begin** *C2* with safe client and STS, the **Key** element recorded in **fullSC** remains secret.





# WS-SecureConversation

---

- C. Kaler, A. Nadalin et al, Web Services Secure Conversation Language (WS-SecureConversation) Version 1.1 (May 2004)
  - <http://msdn.microsoft.com/ws/2004/04/ws-secureconversation/>
- “defines mechanisms for establishing and sharing security contexts, and deriving keys from established security contexts (or any shared secret)” so as to secure series of messages
- Introduces two new security tokens
  - Security context token (SCT): refers to an agreed SC
  - Derived key token (DKT): message-specific derived key



# New Security Tokens

---

```
predicate SCT(sct:item,sctid:string):—  
    sct = <SecurityContextToken><Identifier>sctid</></>.
```

```
predicate DKSCT(dksct:item,sctid:string,nonce:bytes):—  
    dksct = <DerivedKeyToken>  
        <SecurityTokenReference>  
            <Reference>sctid</>  
            <valueType>"SCT"</></>  
            <Nonce>base64(nonce)</></>.
```

```
predicate deriveKey(dk:bytes,key:bytes,nonce:bytes):—  
    dk = pshal(base64(nonce),key).
```

Having established an SC, one can either use it directly,  
or derive from it a key or keys for each message



# SCTs: Two Modes

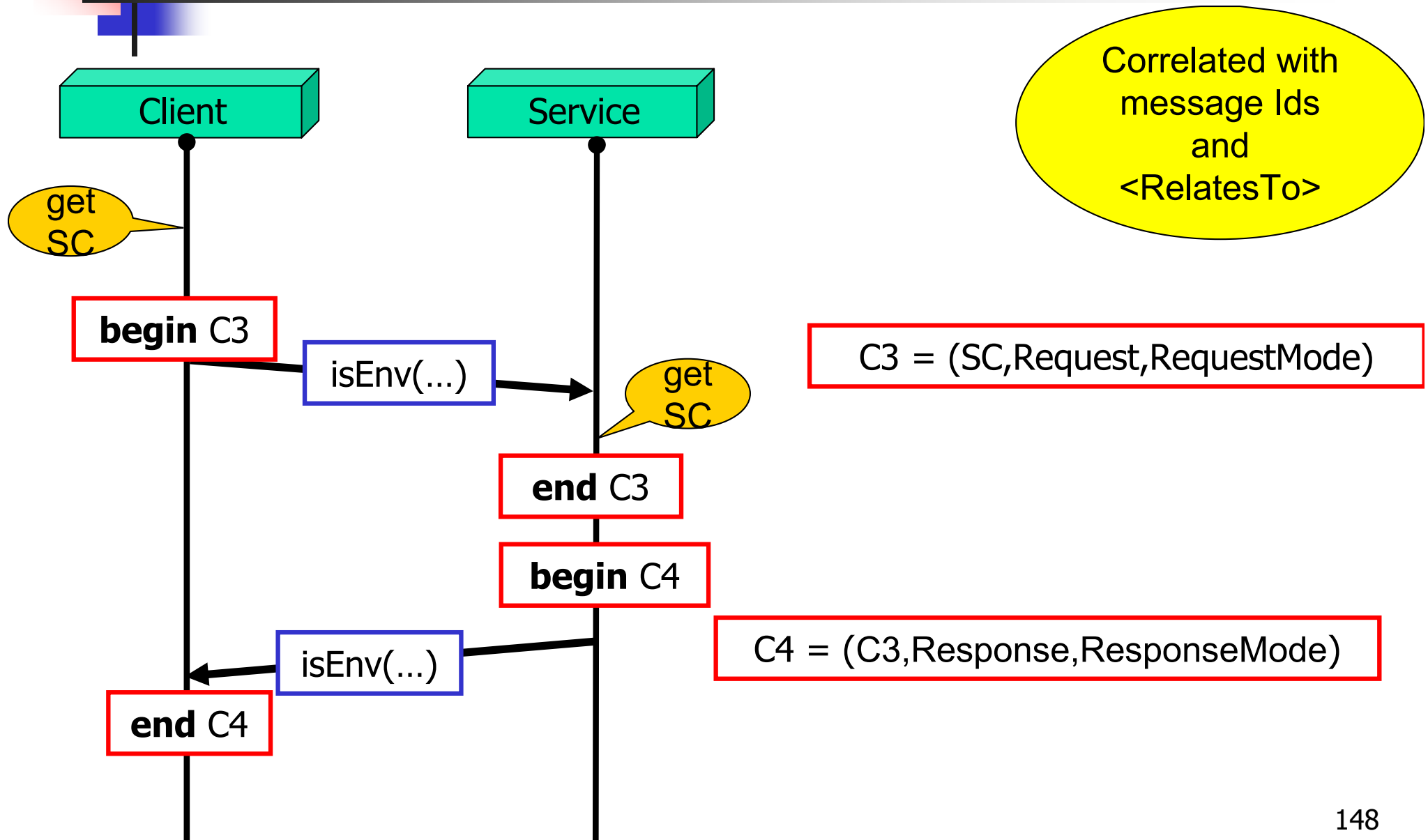
```
predicate env(Env:item, DestInfo:items, t:string, sig:item, ebody:item, sctid:string, mode:item) :-  
    Env = <Envelope>  
        <Header>  
            <Security>  
                <Timestamp><Created>t</></>  
                sct sig</> @ DestInfo</>  
            <Body>ebody</></>,  
    SCT(sct, sctid),  
    nonDerivedKeyMode(mode).
```

Either we use it directly...

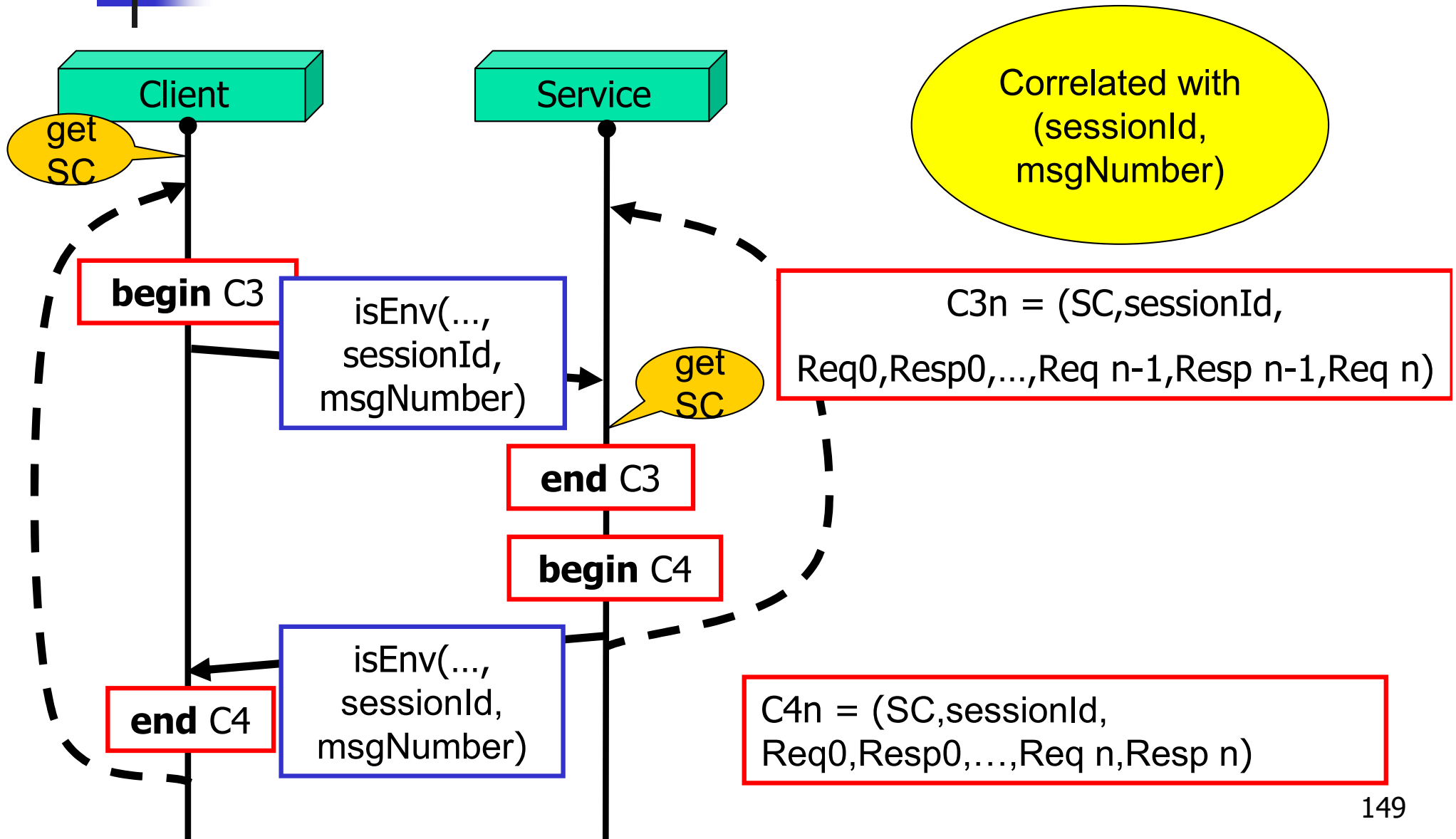
```
predicate env(Env:item, DestInfo:items, t:string, sig:item, ebody:item, sctid:string, mode:item) :-  
    Env = <Envelope>  
        <Header>  
            <Security>  
                <Timestamp><Created>t</></>  
                sct dksctEnc dksctSig sig</> @ DestInfo </>  
            <Body>ebody</></>,  
    SCT(sct, sctid),  
    DKSCT(dksctEnc, sctid, EncNonce), DKSCT(dksctSig, sctid, SigNonce),  
    derivedKeyMode(mode, EncNonce, SigNonce).
```

...or we derive  
separate signing and  
encryption keys

# Model 1: Single Exchange



# Model 2: Multiple Exchanges





# WS-SC Security Results

---

We have the following for Model 2, and a similar result for Model 1.

**Theorem** (*Robust Safety for  $C3_n$  and  $C4_n$  and secrecy of exchanged bodies*) For all runs of the script in the presence of an active attacker, and for all  $n \geq 0$ , we have:

- For each **end**  $C3_n$  with a safe security context, there is a matching **begin**  $C3_n$ .
- For each **end**  $C4_n$  with a safe security context, there is a matching **begin**  $C4_n$ .
- All request and response bodies protected by a safe security context remain secret.



# Discussion

---

- First formal analysis of WS-Trust and WS-SecureConversation
  - XML syntax and automation very effective, against a demanding, realistic attacker model
  - Approx 1000 LOC - manual proofs we published at POPL'04 concerning one or two message protocols would not scale
  - Still, theorem concerning open-ended sessions proved by combination of automated proof and short hand-proof
- As is common, these specs:
  - focus on message formats for interoperability
  - are non-committal regarding security, for example, no clear spec of contents of SCs
- By making modes, data, and goals explicit, we found design and implementation bugs
  - see paper for a discussion



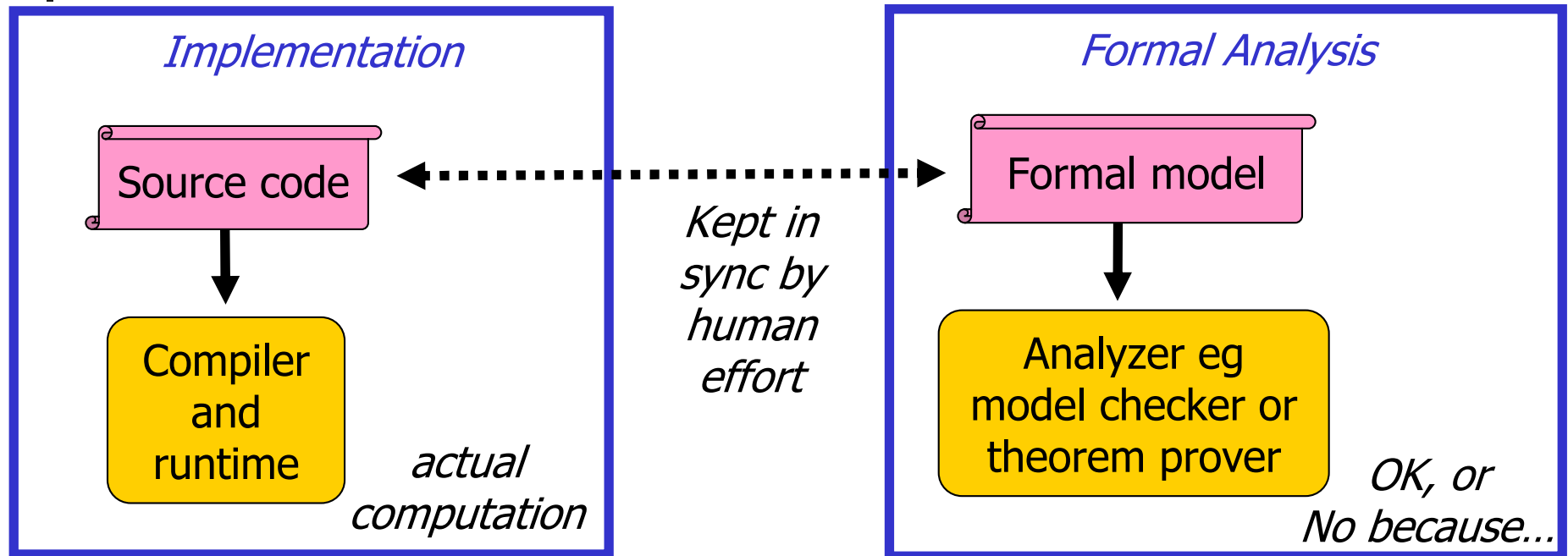
# Interlude

---

A useful distinction concerning formal methods

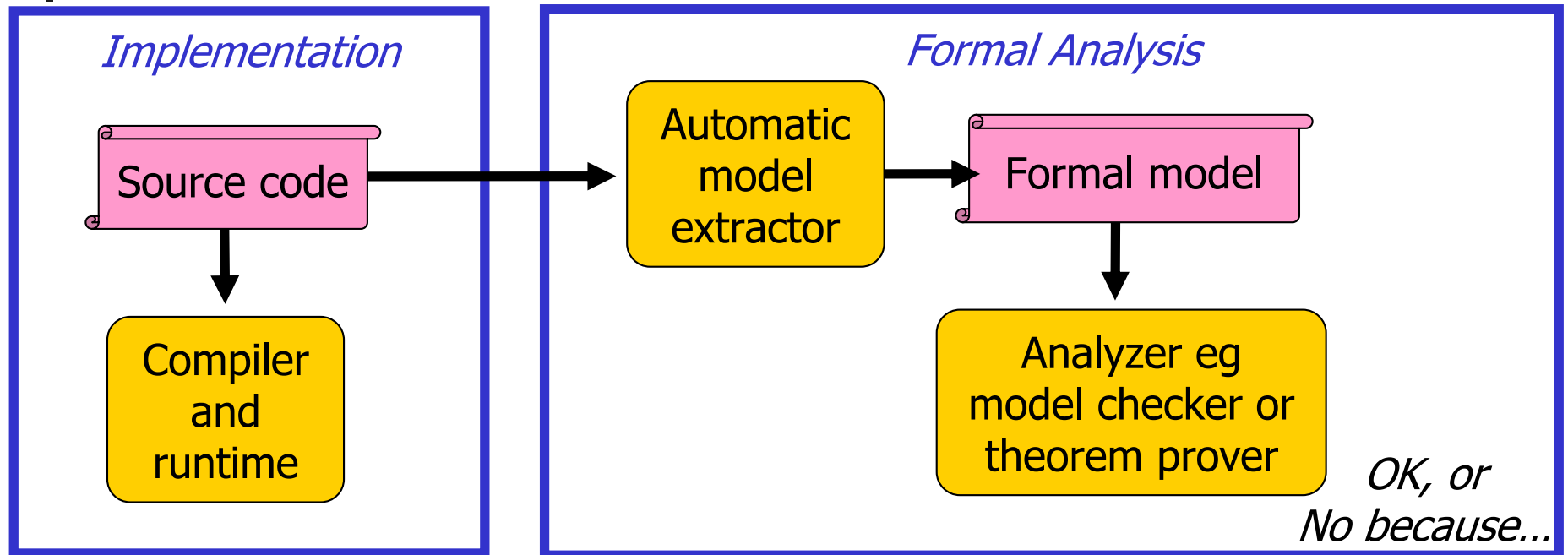


# Model-Based Formal Methods

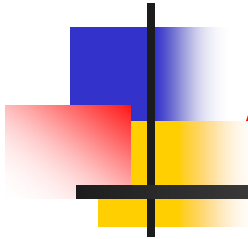


- Modelling always abstracts “real world” detail, so may miss some bugs; still, effective when studying fixed, difficult algorithms or standards
- Much worse, spec-based formal methods do not scale in practice; too costly to maintain two documents

# Source-Based Formal Methods



- By extracting the model directly from the source code, formal tools remain applicable as design evolves
- "One document. One. It's the source code. You learn everything there and you know everything there."



# Part II: Generating and Analyzing WS-Policy

---

A partially source-based approach to WS-Security

K. Bhargavan, C. Fournet, A. D. Gordon, *Verifying Policy-Based Security for Web Services*, to appear at ACM CCS (2004)

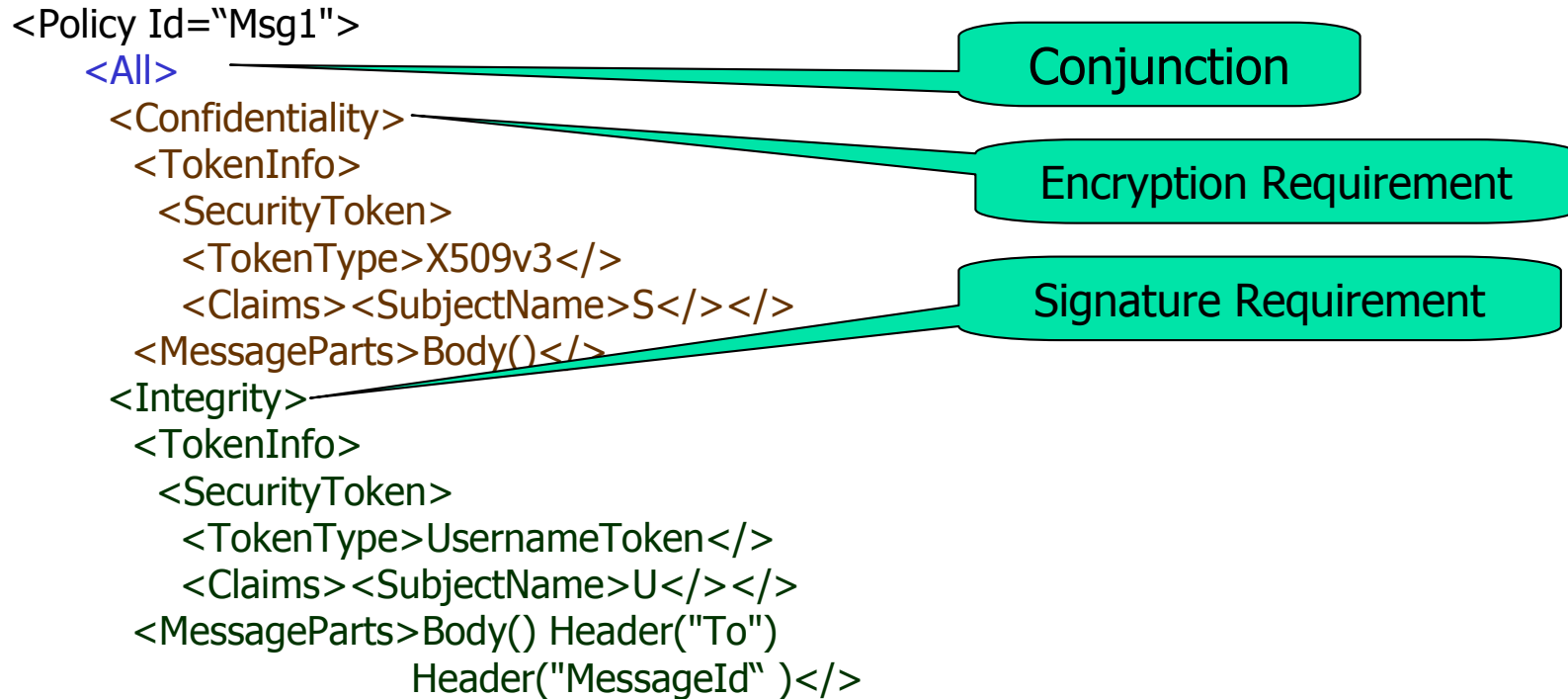


# Declarative Security Policies

---

- In general, web services export XML-encoded metadata – WSDL files are the most established
- WS-Policy and WS-SecurityPolicy define a declarative XML format for programming how web services implementations construct and check security headers.
- By expressing security checks as XML metadata instead of imperative code, policy-based web services conform to the general principle of isolating security checks from other aspects of message processing, to aid security reviews.
- Both sender and receivers have configuration files
  - A policy is a predicate on a SOAP message
  - A policy map associates incoming and outgoing messages to particular policies (via <To> and <Action>)

# An Example





# Demo

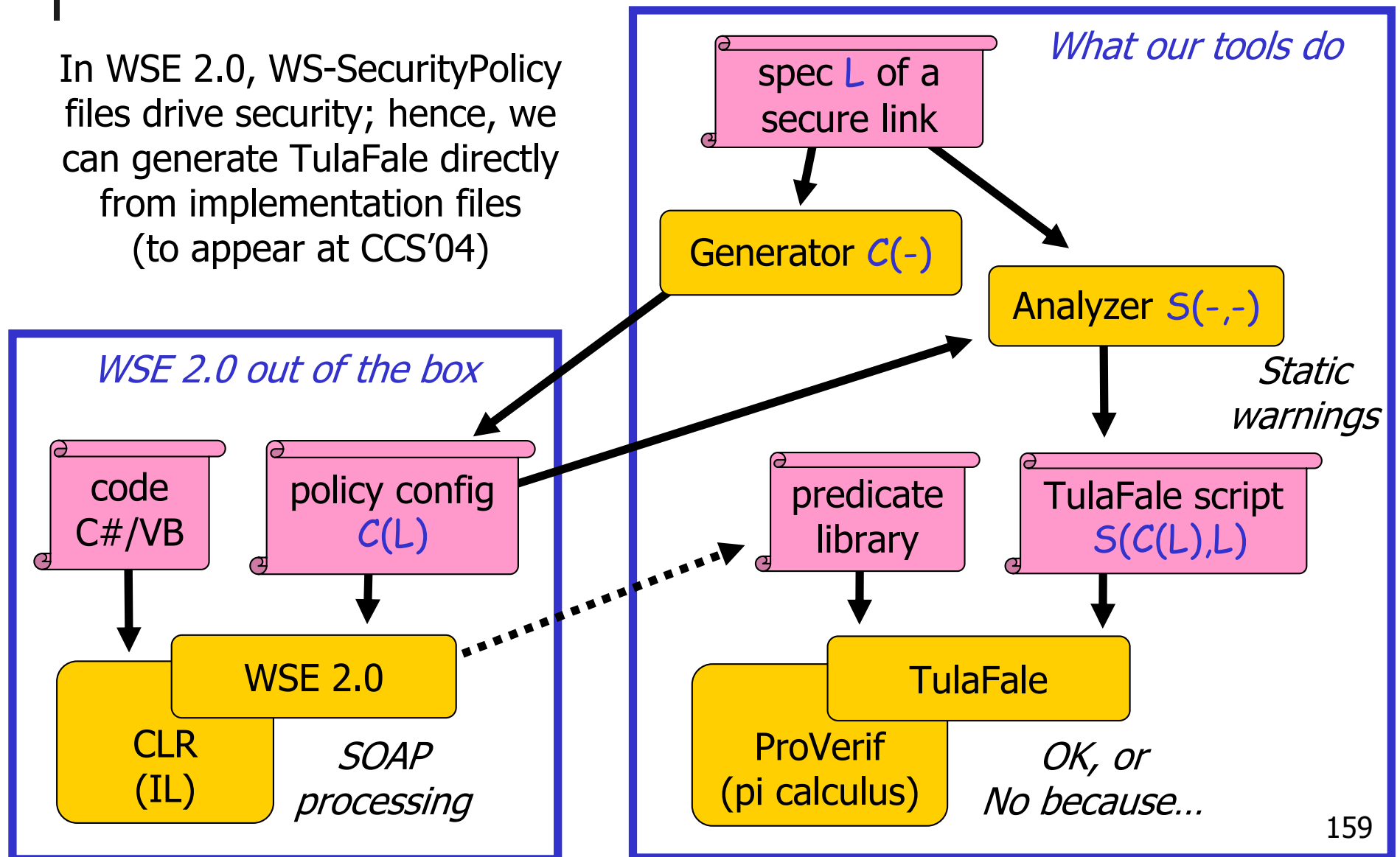
---

Policy-driven security in WSE 2.0

Next: getting policies right is still a problem...

# Tool 2: Policy Generator/Analyzer

In WSE 2.0, WS-SecurityPolicy files drive security; hence, we can generate TulaFale directly from implementation files (to appear at CCS'04)



# Translating Policies to Predicates

<Policy Id="Msg1">

<All>

Conjunction

<Confidentiality>

Encryption Requirement

<TokenInfo>

<SecurityToken>

<TokenType>X509v3</>

<Claims><SubjectName>S</></>

Signature Requirement

<MessageParts>Body()</>

<Integrity>

<TokenInfo>

<SecurityToken>

<TokenType>UsernameToken</>

<Claims><SubjectName>U</></>

<MessageParts>Body() Header("To")

Header("MessageId")</>

*predicate* hasMsg1Policy(msg1:item,U:item,pwd:string,  
S:item,skS:bytes,id1:string,req:item) :-

msg1 =

<Envelope>

<Header>

<To>S</>

<MessageId>id1</>

<Security>

utok

sig1</></>

<Body>b1</></> ,

isEncryptedData(b1,req,skS),

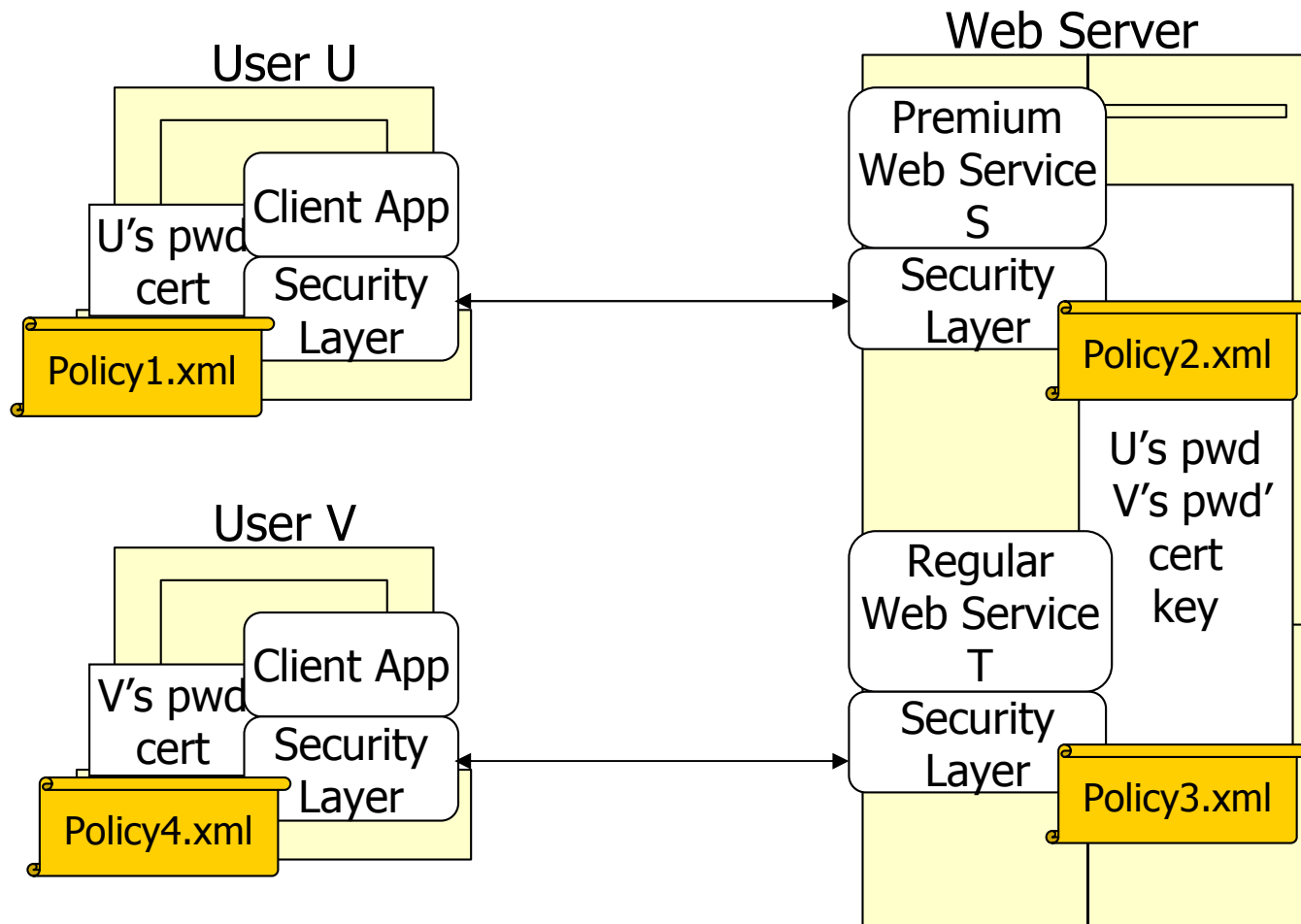
isUserTokenKey(utok,U,pwd,skU),

isSignature(sig1,"hmacsha1",skU,

[<Body>b1</> <To>S</> MessageId>id1</>]).



# Analyzing Policy Configurations



Automated tools for collecting, parsing policies from IIS Servers, Clients  
Config = [Policy1, Policy2, Policy3, Policy4]



# Link Specifications

- Link: Security spec for a single web service
- Spec = [Link1, Link2]
- Link1 =
  - {ServiceURI = "http://server/servicePremium",
  - ClientPrins = [U],
  - ServicePrin = S,
  - SecrecyLevel = Encrypted}
- Link2 =
  - {ServiceURI = "http://server/serviceRegular",
  - ClientPrins = [U, V],
  - ServicePrin = S,
  - SecrecyLevel = Clear}
- Links translate to security goals in TulaFale
  - All requests and responses on Link1 and Link2 must be secure

Web Location  
of Service

Allowed Users

Service Cert  
Subject Name

Request/Response  
Secrecy

Secrecy not required



# Security of Generated Policies

---

For particular links and policies (either generated or custom), analysis usually takes a few minutes.

We can also prove general results such as the following that formalizes the intuition that *all* configurations generated from link specifications are safe:

**Theorem** *For any link spec  $L$ , process  $\mathcal{S}(C(L), L)$  is robustly safe.*

The proof is by a combination of automated and manual proof.  
We also have secrecy and correlation results.



# Related Work

---

- Going in the opposite direction to our policy analyzer, several tools compile formal models to code:
  - Strand spaces: Perrig, Song, Phan (2001), Lukell et al (2003)
  - CAPSL: Muller and Millen (2001)
  - Spi calculus: Lashari (2002), Pozza, Sisto, Durante (2004)
  - Apparently, the resulting code cannot yet interoperate with other implementations – an important future target
- Other Dolev-Yao modelling of web services
  - Model-checking of some example WS-Security specs using FDR, uncovering similar attacks: Kleiner & Roscoe (2004)
- Other formalizations of XML and web services specs
  - XPath, XSLT, XQuery: Wadler et al (since 1999)
  - WS-RM: Johnson, Langworthy, Lamport, Vogt (2004)



# Lecture 4: Summary

---

- These projects demonstrate use of TulaFale
  - As a model-based FM for analyzing uses of WS-Trust and WS-SecureConversation
  - As a partially source-based FM for analyzing policy-driven WSE installations
- The idea of a link spec is related to the declarative attributes we discussed in Lecture 2: both emphasise the need for application-level, not just SOAP-level, views of security
- This is current work; we intend to develop both directions
  - There is much to do...

End of  
Lecture 4