Secure Global Computing with XML Web Services: Theory and Practice

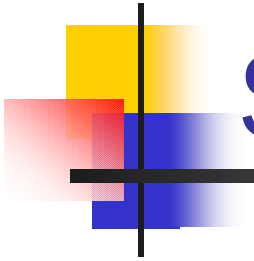
Andy Gordon

Microsoft Research

EEF Global Computing Summer School

Edinburgh, Scotland, July 7-11, 2003

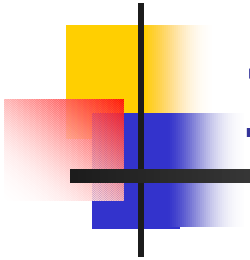
Lecture 1, Tuesday 11:00-12:30



Syllabus

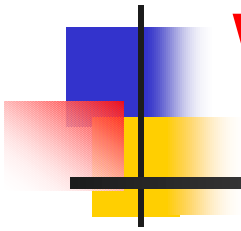
- Lecture 1, Tuesday 11:00-12:30
  - Web services
- Lecture 2, Tuesday 16:30-18:00
  - Security protocols
  - Type systems for secrecy and authenticity properties
- Lecture 3, Wednesday 9:00-10:30
  - Analysing web services security

This course reflects joint work with M. Abadi, K. Bhargavan, C. Fournet, A. Jeffrey, and R. Pucella



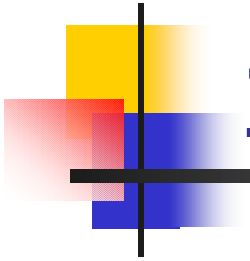
1: Web Services

- Basics of Web Services
- Demo: Constructing a Web Service
- SOAP-Level Security
- Abstract vs XML Views of Simple Sample
- Demo: Signing Message using WS-Security



Part I: Basics of Web Services

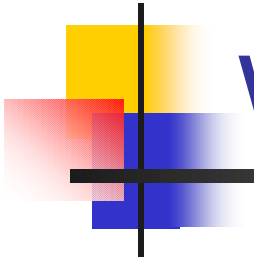
Websites for computers not humans; XML versus HTML; SOAP, WSDL, and HTTP; current state of adoption



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

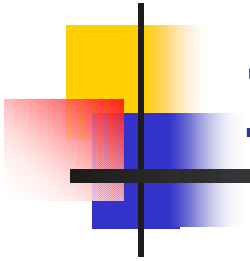
From a programming language perspective, an unusual, for 1992, model of computation, but rather effective



Websites for Computers B.X.E.

- "A web service is a website intended for computer programs rather than human beings" (Barclay et al)
- 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

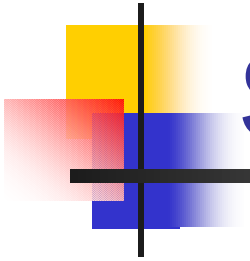
A (Slightly Simplified) Abstract Syntax for XML

Label	::= `anyLegalXmlName`	element or attribute name
String	::= `any legal XML string`	XML string
Att	::= *Label*`="`*String*`"`	attribute
Atts	::= $Att_1 \ldots Att_n$	attribute sequence
Item	::= *Element* | *String*	item
Items	::= $Item_1 \ldots Item_n$	item sequence
Element	::= <*Label Atts*>*Items*</*Label*>	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 specifications:
  - SOAP – message format
    - Syntax of request, response, fault messages
  - 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
    - Opportunity for clean and simple alternative...?



SOAP

- Simple Object Access Protocol, early version in 1999
- Descends, in part, from earlier XML-RPC proposal
  - Deployed commercially, but lacked extensibility
- SOAP envelope – request, response, or fault – sent from one end-point to another, possibly via intermediaries
  - Has optional header and mandatory body
- Conventions for encoding requests and responses
- Most commonly, SOAP sent over HTTP transport, but other underlying transports possible, such as HTTPS/SSL
- Usually request/response pattern, others possible
- Many implementations
  - .NET is Microsoft's implementation of SOAP web services

A Sample Web Service

Smart client for checking orders

The Internet

www.bobspetshop.com

```
PetShopService ws = new PetShopService();
Order o = ws.GetOrder(20);
```

```
[WebMethod]
public Order GetOrder(int orderId) {
  return orderWebService.GetOrder(orderId); }
```

Implementation via proxy class and HTTP transport

SOAP Request

SOAP Response

Implementation via WebService classes in Web Server

Pet Shop database

Vendor-neutral XML-encoding over HTTP

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>
    <GetOrder xmlns="http://microsoft.com/petshop">
      <orderId>20</orderId>
    </GetOrder>
  </soap:Body>
</soap:Envelope>
```

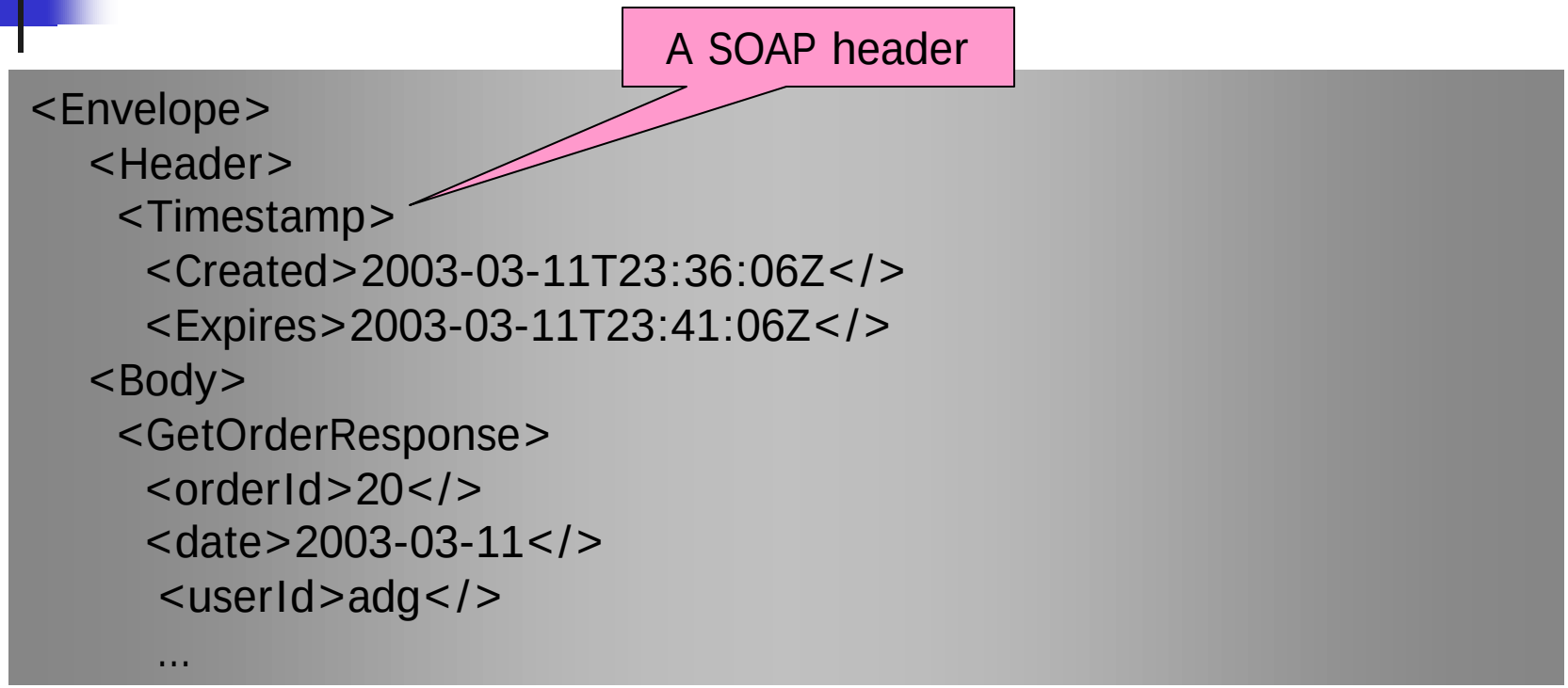
- Says: “get me status of order 20”
- XML not meant to be read by humans, so we’ll omit namespace info, and trailing brackets...

A Sample SOAP Request

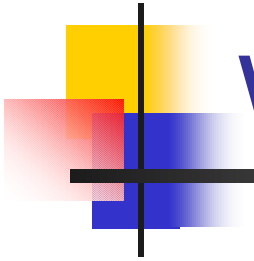
```
<Envelope>
   <Body>
     <GetOrder>
       <OrderId>20</>
```

- Says: “get me status of order 20”
- XML not meant to be read by humans, so we’ll omit namespace info, and trailing brackets...that’s better

A Sample SOAP Response



- 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

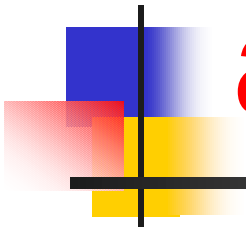
```
AddInt ::= <AddInt> <a>int</> <b>int</></>
AddIntResponse ::= <AddIntResult>int</>
```

```
<message name="AddIntSoapIn">
  <part name="parameters" element="AddInt"/>
<message name="AddIntSoapOut">
  <part name="parameters" element="AddIntResponse"/>
```

```
<portType name="AddNumbersSoap">
    <operation name="AddInt">
      <input message="AddIntSoapIn" />
      <output message="AddIntSoapOut" />
```

```
<binding name="AddNumbersSoap" type="AddNumbersSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
   <soap:operation name="AddInt">
     <operation soapAction="http://microsoft.com/.../AddInt" />
```

```
<service name="AddNumbers">
   <port name="AddNumbersSoap" binding="AddNumbersSoap">
    <soap:address location="http://localhost/.../usernamesigning.asmx"/>
```



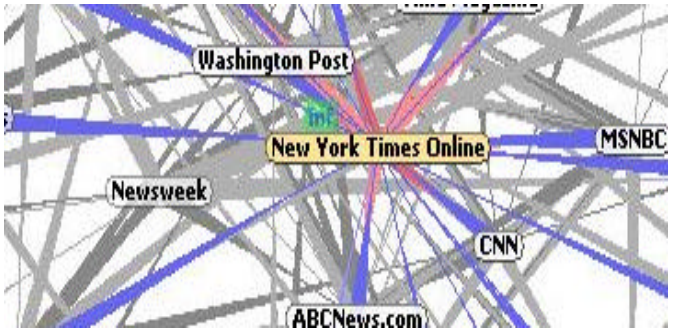
Demo: Constructing a Web Service

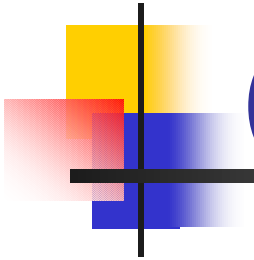
High-level RPC model; SOAP messages; WSDL descriptions

Next: Three ways in which SOAP web services are being used, and what's actually new?

(1) On the Public Internet

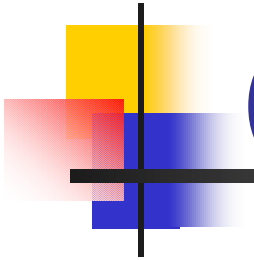
- **Aim:** outsource user-interface to specialists
- Specialist storefronts
  - Ex: Products sold through Amazon, but associate site gets commission
  - 25k developers since Jul'02
- Specialist visualizations
  - Ex: TouchGraph re Google
- Specialist smart clients
  - Ex: SourceForge project





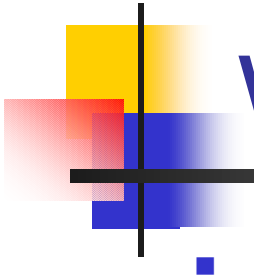
(2) Within a Private Intranet

- **Aim:** XML as common language to integrate systems obtained from different vendors
- Ex: single service to calculate quotes uniformly no matter how customer contacts an insurance company
  - call centre
  - sales force
  - website
- Today, biggest growth area for web services, in part because there are few new security concerns



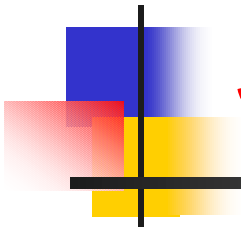
(3) Between Private Intranets

- **Aim:** support inter-institution workflows
  - E-business transactions
  - E-science GRID
- This is the hardest of the three cases to secure
- Needs being addressed by crypto-based security specs
  - **Single sign-on**: scientist may access 100 sites
  - **Federation**: eg can company A permit just certain company B employees to access shared service?
  - **Delegation**: eg can I permit online travel agent to consult my calendar service, make bookings?



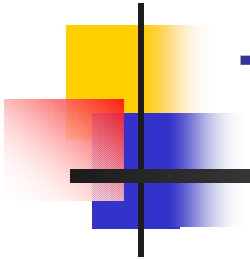
Web Services: What's New?

- 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
- yet reasons for caution,
  - Cost of SOAP encoding?
  - Lack of SOAP security?
- and some competition,
  - Fielding's REST: HTTP-based web services
  - ebXML: XML version of earlier UN EDI format



Part II: SOAP-Level Security

Transport- versus application-level security; WS-Security for embedding a range of security tokens within SOAP headers



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, independently of underlying transports

WS-Security: Syntax Summary

Security element is child of SOAP Header

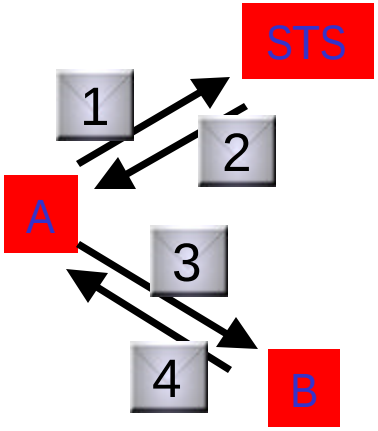
Security ::= <Security ?*Actor*> **SecurityToken* </>
SecurityToken ::=
  <UsernameToken>
    <Username>*String*</>
    ?<Password Type="*PasswordType*">*String*</>
    ?<Created>*String*</>
    ?<Nonce>*Base64Binary*</>
| <BinarySecurityToken>*Base64Binary*</>
| <SecurityTokenReference>
    <Reference URI="*Uri*">
| <KeyInfo>**KeyInfoItem*</>
| <Signature>*SignedInfo SignatureValue*</>
| <ReferenceList>+<DataReference URI="*Uri*"/></>
| *EncryptedKey*
| *EncryptedData*

UsernameToken identifies particular user

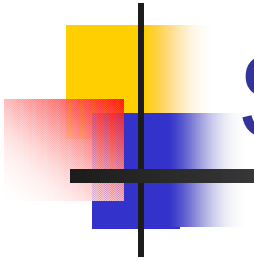
BinarySecurityToken embeds an existing format such as an X509 public-key certificate, or a Kerberos certificate

Outline Security Architecture

Message 1	A → STS:	*Signed token request*
Message 2	STS → A:	*Signed tokens*
Message 3	A → B:	*Signed request*
Message 4	B → A:	*Signed response*



- Abstract family of protocols:
  - A wants to talk to B but may have insufficient security tokens to satisfy B's *security policy*
    - *Trust policy* tells which tokens are valid
    - *Authorization policy* tells if valid tokens suffice for request
  - So A gets tokens from *Security Token Server* STS
- Abstracts from underlying crypto technologies
- Will consider three party protocols (but not in XML form) in Lecture 2

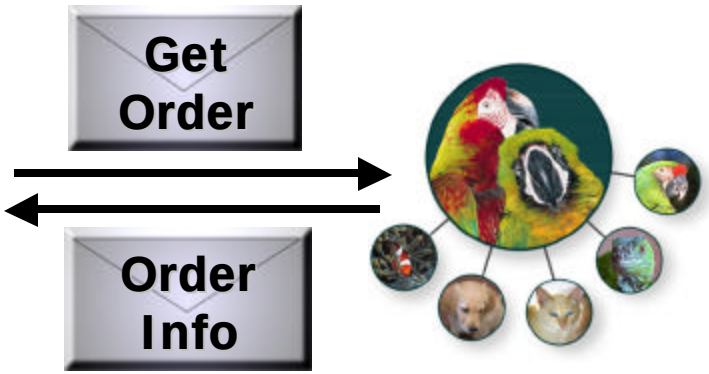


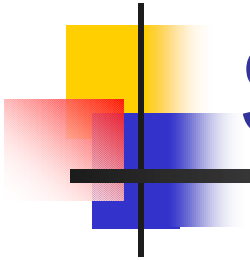
Security Spec Overview

- Several specs released since summer 2002:
  - **WS-Security**: message integrity, confidentiality, authentication; security token attachment, both XML (SAML, XrML) and binary (Kerberos, X509)
  - **WS-Trust**: request and issue security tokens, manage trust relationships
  - **WS-SecureConversation**: establish and share security contexts, derive session keys
  - **WS-SecurityPolicy**: security requirements
- Plus several implementations:
  - Microsoft WSE (Web Service Enhancements)
    - RTW Dec 2002, free plugin for VS.NET 2002 and later
    - Product implementing WS-Security, WS-Routing, and DIME attachments
  - Others from IBM, Verisign, etc

Part III: Abstract vs XML Views of Simple Sample

We illustrate WS-Security by explaining the design of a simple but typical authentication protocol





Sample Security Goals

- Suppose a human A with password p uses a client I to invoke a web service at URL S
  - S = http://www.bobspetshop.com/ws/orderstatus.asmx
- Without some kind of authentication, anybody could request the private details of anyone else's order
- Simple solution to require p-based signature of:
  - Message body
    - to show request from A, and has not been modified
  - Timestamp
    - to detect replays, with cache of recent messages
  - Web server S
    - to detect redirection from another server

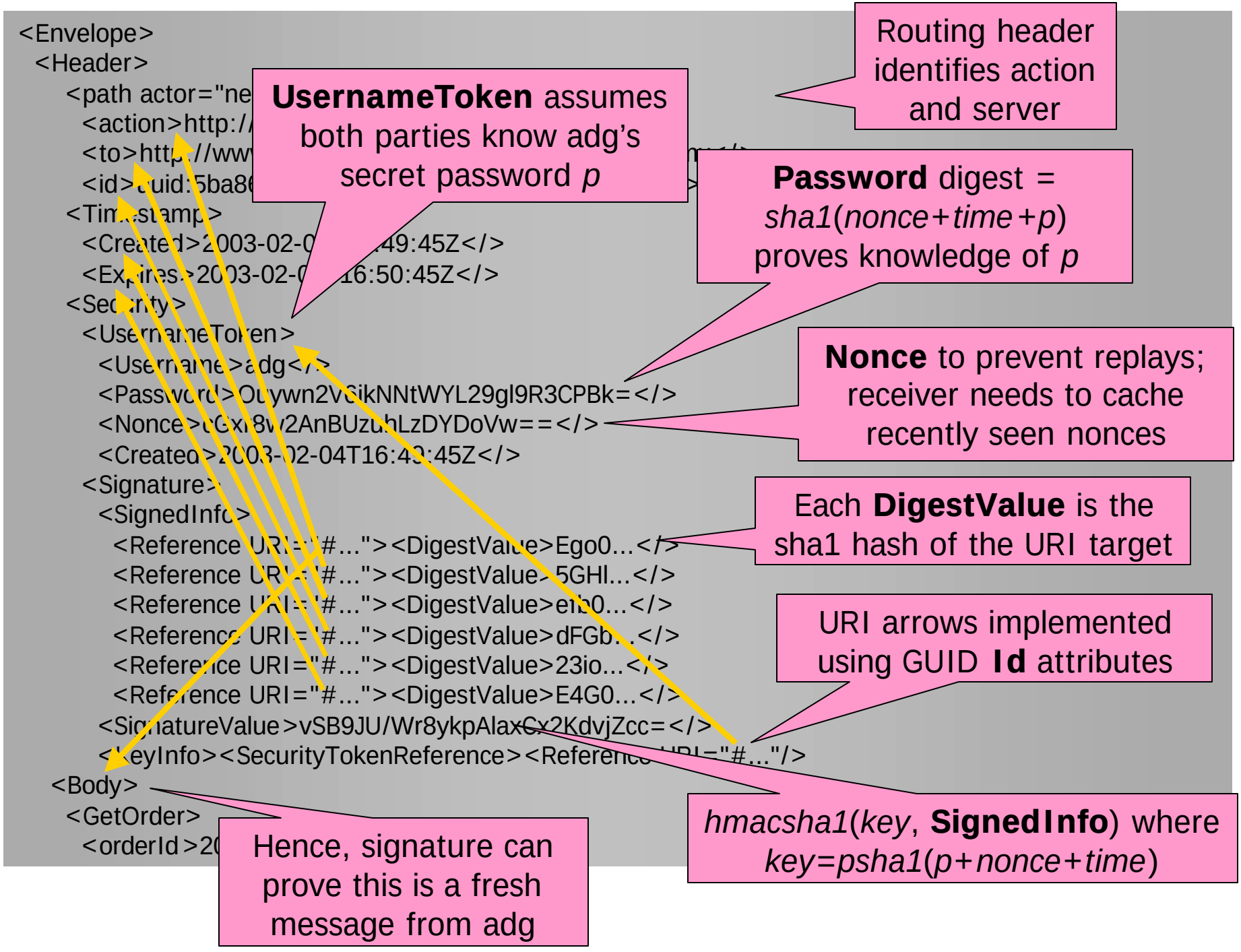
Scope of our Threat Model

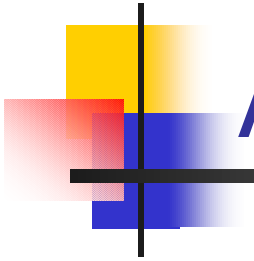
- The threat model is an attacker who can replay, redirect, assemble new messages, but cannot brute force secrets such as passwords
  - Formal statement usually credited to Dolev and Yao 1983, but basic ideas in Needham and Schroeder's pioneering 1978 work on crypto protocols
- Can verify that crypto protocols establish various safety properties in spite of such an attacker:
  - Message authentication – against unauthenticated access
  - Message integrity – against parameter manipulation
  - Message confidentiality – against eavesdropping
  - Message freshness – against replays
- Like all formal or informal methods, certain threats lie outside the model, and must be addressed separately
  - Disclosure of configuration data
  - Unauthorized access via SQL injection or cross-site scripting

An Abstract Protocol

Event 1	I logs	begin(S,A,t,orderid)
Message 1	I $\rightarrow$ S	S,A,t, $h(S,t,orderid)_p$,GetOrder(orderid)
Event 1'	S logs	end(S,A,t,orderid)
Message 2	S $\rightarrow$ I	GetOrderResponse(orderInfo)

- Security goal expressed as a correspondence
  - Each end-event corresponds to a begin-event
- In Lecture 2, we will explain how to formalize diagrams like this within the pi calculus, and verify safety with respect to any Dolev-Yao opponent
  - who can replay, redirect, assemble new messages, but cannot brute force secrets such as passwords
- To justify this assumption, p must be a strong password, stored on client I, and not just a weak memorizable password subject to dictionary attacks
- Next, a wire-level view of the implementation...


<Envelope>
<Header>
<path actor="ne
<action>http://
<to>http://ww
<id>uuid:5ba8
<Timestamp>
<Created>2003-02-0 :49:45Z</>
<Expires>2003-02-0 16:50:45Z</>
<Security>
<UsernameToken>
<Username>adg</>
<Password>Otywn2vSikNNtWYL29gl9R3CPBk=</>
<Nonce>tGxi8v2AnBUzuhLzDYDoVw==</>
<Created>2003-02-04T16:49:45Z</>
<Signature>
<SignedInfo>
<Reference URI="#..."><DigestValue>Ego0...</>
<Reference URI="#..."><DigestValue>5GHl...</>
<Reference URI="#..."><DigestValue>eib0...</>
<Reference URI="#..."><DigestValue>dFGb...</>
<Reference URI="#..."><DigestValue>23io...</>
<Reference URI="#..."><DigestValue>E4G0...</>
<SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
<KeyInfo><SecurityTokenReference><Reference URI="#..."/>
<Body>
<GetOrder>
<orderId>20
UsernameToken assumes both parties know adg's secret password *p*
Routing header identifies action and server
Password digest = *sha1*(*nonce*+*time*+*p*) proves knowledge of *p*
Nonce to prevent replays; receiver needs to cache recently seen nonces
Each **DigestValue** is the sha1 hash of the URI target
URI arrows implemented using GUID **Id** attributes
hmacsha1(*key*, **SignedInfo**) where *key*=*psha1*(*p*+*nonce*+*time*)
Hence, signature can prove this is a fresh message from adg



Aside: 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

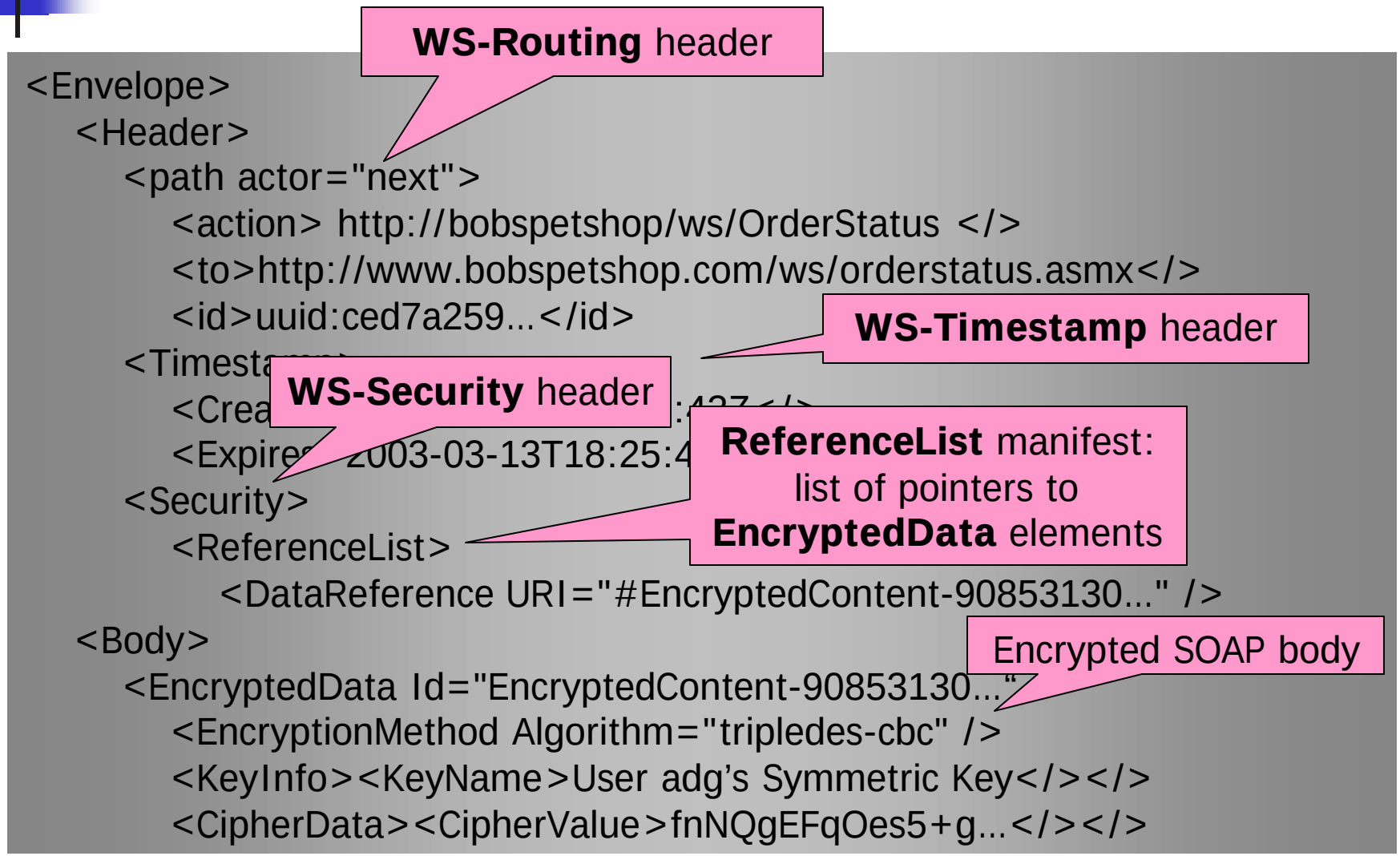
Sample: Username Signing

```
// Create an instance of the Web service
AddNumbers serviceProxy = new AddNumbers();
SoapContext requestContext = serviceProxy.RequestSoapContext;

// Configure URL in proxy from external config file
ConfigureProxy(serviceProxy);

// Add the security token and request a signature
UsernameToken token = new UsernameToken("adg", "OpenSesame",
                                        PasswordOption.SendHashed);
requestContext.Security.Tokens.Add(token);
requestContext.Security.Elements.Add(new Signature(token));

// Call the service
Console.WriteLine("Calling {0}", serviceProxy.Url);
int sum = serviceProxy.AddInt(a, b);

// Success!
string message = string.Format("{0} + {1} = {2}", a, b, sum);
Console.WriteLine("Web Service called successfully: {0}", message);
```

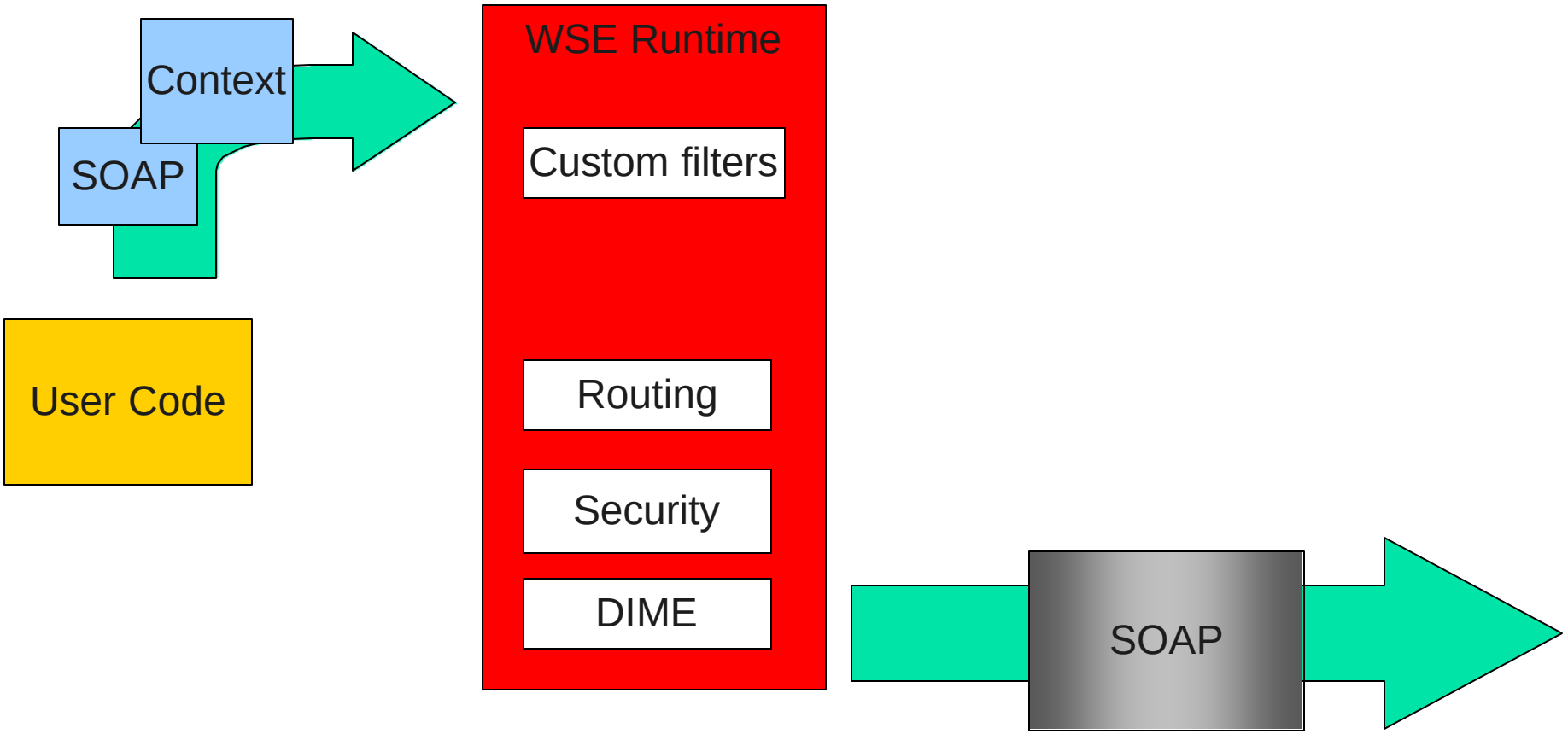
After proxy constructs SOAP message, the security filter will compute the signature

After call, ResponseSoapContext describes tokens and signatures on response

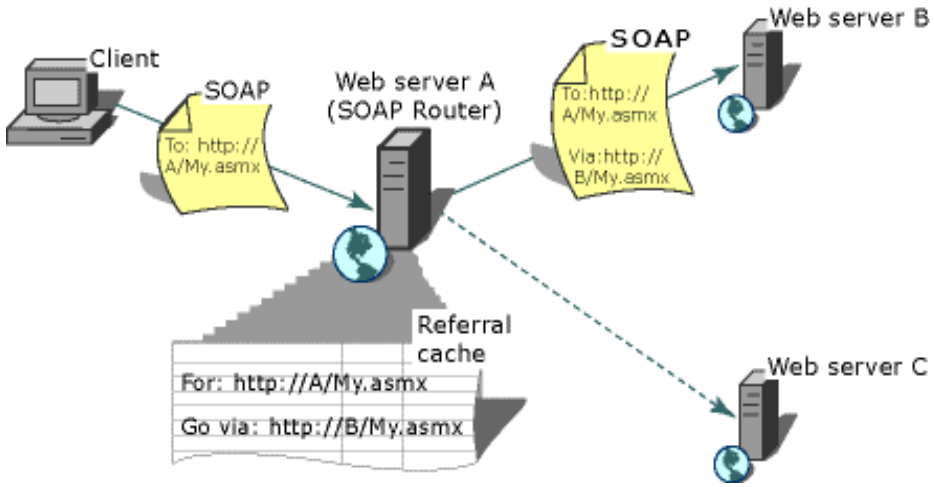
An Encrypted Request



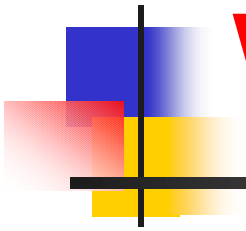
Message Filtering in WSE



Aside: WS-Routing



- Client requests a service from server A, which in fact routes it on to server B
- Routing determined by a *referral cache* at A
- If B needs to be taken offline, we can update the cache to point at C, transparently to the client
- Aside to an aside: DIME enables non-XML, binary attachments



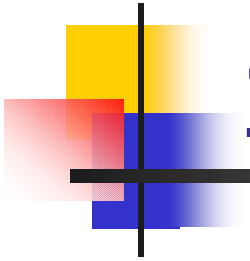
Demo: Signing with WS-Security

To try this at home, you need:

Windows XP Pro (not Home Edition)

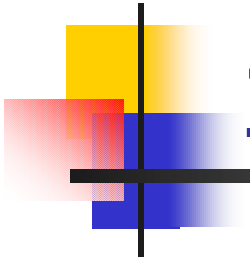
Visual Studio .NET or Visual Studio .NET 2003

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



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
- 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: type-theoretic techniques for verification
  - Lecture 3: application to SOAP-level security protocols



1: Resources

- Cardelli and Marais websites, for service combinators, and WebL
  - http://www.luca.demon.co.uk/
  - http://www.hannesmarais.com/index.html
- Standards tracks and whitepaper
  - http://www.w3.org/2002/ws/
  - http://www.oasis-open.org
  - http://msdn.microsoft.com/ws-security/
- Abiteboul of INRIA and Leymann of IBM, on web services:
  - http://www-rocq.inria.fr/~abitebou/PRESENTATION/WebServices-EDBT02.pdf
  - http://www.btw2003.de/proceedings/proceedings.en.shtml
- General introduction to computer security (one of many...)
  - http://www.cl.cam.ac.uk/Teaching/2001/IntroSecurity
- 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

End of
Part 1

Secure Global Computing with XML Web Services: Theory and Practice

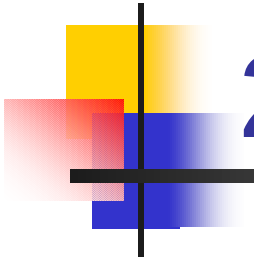
Andy Gordon

Microsoft Research

EEF Global Computing Summer School

Edinburgh, Scotland, July 7-11, 2003

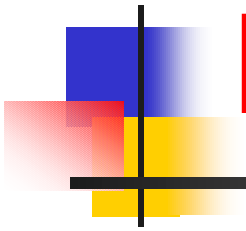
Lecture 2, Tuesday 16:30-18:00



2: Types for Crypto Protocols

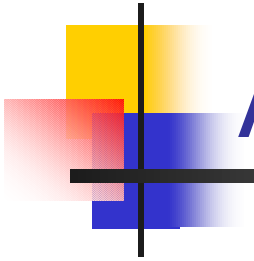
- Basic ideas of crypto protocols
- How the Dolev-Yao threat model can be rephrased within a process calculus
  - Attacker can modify, decrypt, replay network traffic, but cannot simply guess keys (even by brute force)
- How secrecy properties can be proved by typing
  - Continuation of long literature on security levels
- How authenticity properties can be proved by typing
  - Continuation of long literature on effect systems

Basic Ideas of Crypto Protocols



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

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



A Little History

- 1978: Needham and Schroeder invent authentication protocols; descendants eg SSL now widely deployed
- 1981: attack found on symmetric key protocol (!)
- 1987: Burrows, Abadi, and Needham invent a formal logic of authentication ("BAN logic")
- 1995: attack found on asymmetric key protocol (!!)
- 1997: Several formal methods for "BAN problem"
  - Tradeoff between accuracy and approximation: theorem-proving versus model-checking
- Today: Many formal methods now developed
  - Several deliver both accuracy and automation
    - Examples: Cohen's TAPS, Blanchet's Proverif

Ex I: Sending Messages

Message 1	B $\rightarrow$ A:	NB
Message 2	A $\rightarrow$ B:	$\{msg,NB\}_{KAB}$

- A and B share a secret key KAB
- A wishes to send a series of instructions to B
- We wish to secure against tampering and replays, and authenticate A's identity to B
- Encryption prevents tampering, and helps guarantee authenticity
- A *nonce challenge* prevents *replay attacks*

Ex II: Server-Based Login

- 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 II: 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	$S \rightarrow B$:	$\{NB\}_{KBS}$

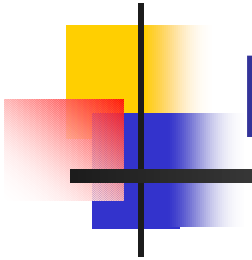
- Message 5 meant to prove to B that A is currently running the protocol
- But it doesn't mention A, so by manipulating parallel sessions, an attacker C may login as A

Attacking Ex II

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. $S \rightarrow B$: $\{...\}_{KBS}$

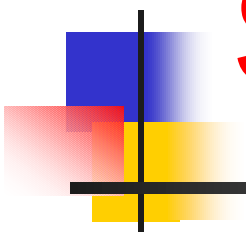
1. $C \rightarrow B$: C
2. $B \rightarrow C$: NB_C
3. $C \rightarrow B$: $\{NB_A\}_{KCS}$
4. $B \rightarrow S$: $B,\{C,\{NB_A\}_{KCS}\}_{KBS}$
5. $S \rightarrow B$: $\{NB_A\}_{KBS}$

- 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.



Discussion

- We include the additional names in Ex II to prevent an *impersonation attack*
- Infamously, crypto protocols are vulnerable to attack, without breaking the underlying crypto algorithms
- Ongoing need for protocol verification: new technologies force invention of new protocols, and change security assumptions
- The message notation is abstract, but not completely precise; e.g., authentication goals left implicit
- So, how might we specify properties formally...

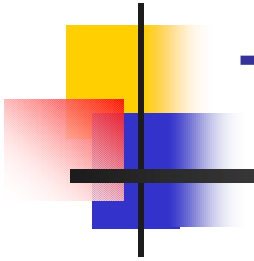


Formally Specifying Security Properties

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 of Abadi and Gordon (CCS'97, Concur'97)

Related process calculi include the sjoin-calculus, the applied π-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:

decrypt $\{L\}_N$ **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 $\{L\}_N$
- Abstraction introduced by Dolev and Yao (1983)

Specifying Authenticity

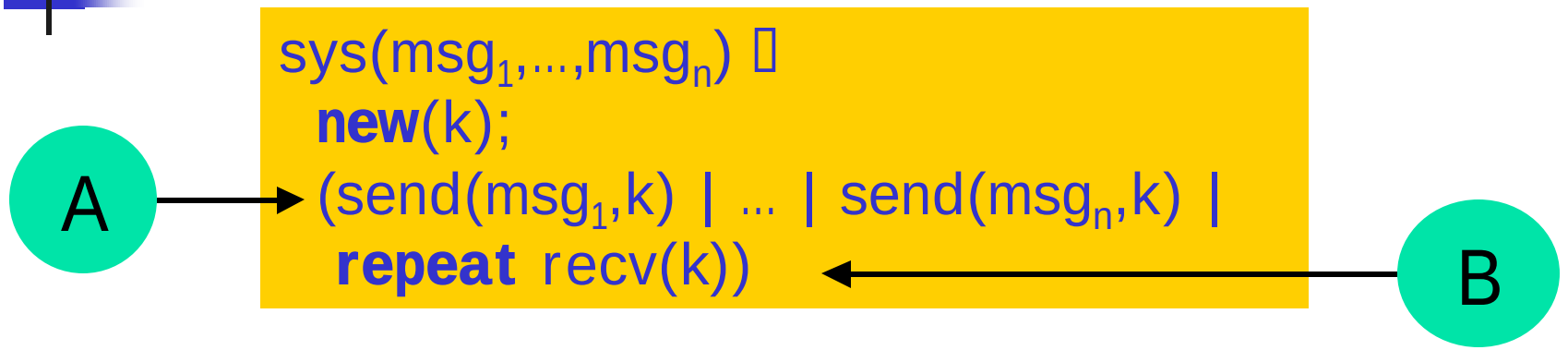
Event 1	A begins	(A sending msg to B)
Message 1	B $\rightarrow$ A:	NB
Message 2	A $\rightarrow$ B:	$\{msg,NB\}_{KAB}$
Event 1'	B ends	(A sending msg to B)

Each end-assertion to have distinct, preceding begin-assertion with same label

Attacks (replays, impersonations) show up as violations of these assertions

Named *correspondence assertions* by Woo and Lam, but also *injective agreements* by Lowe.

Authenticity Specified in Spi



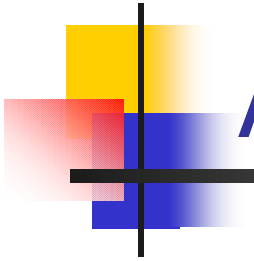
send(msg,k)
inp net(no);
begin “Sender sent msg”;
out net ({msg,no}$_k$);

recv(k)
new(no); **out** net(no); **inp** net(u);
decrypt u **is** {msg,no’}$_k$;
check no’ **is** no; **end** “Sender sent msg”;

Secrecy Specified in Spi

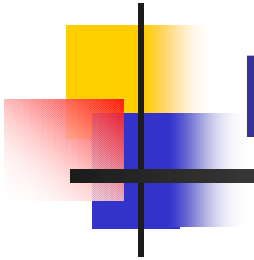
For all (msg_{L1}, msg_{R1}), ..., (msg_{Ln}, msg_{Rn}),
$sys(msg_{L1},\ldots,msg_{Ln}) \approx sys(msg_{R1},\ldots,msg_{Rn})$

- No opponent O should be able to distinguish runs carrying different messages.
- We interpret $P \approx Q$ 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...



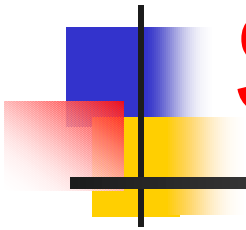
A Small Information Leak

- Consider sys(msg_1,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 $\{msg_1,no_1\}_k$ and $\{msg_2,no_2\}_k$
- But the opponent may feed its own nonce no twice to A, cause A to emit $\{msg_1,no\}_k$ and $\{msg_2,no\}_k$, and hence can tell whether $msg_1=msg_2$
- To fix this, A sends $\{msg,no,co\}_k$ for some fresh *confounder* co instead of simply $\{msg,no\}_k$



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...



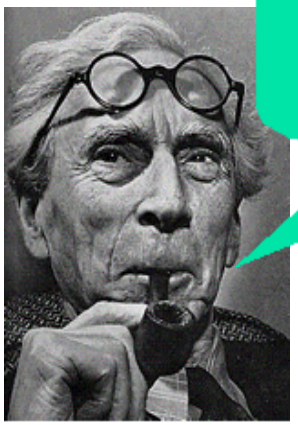
Secrecy Types

Introduction to “Secrecy by Typing in Security Protocols” by Abadi (JACM 1999)

Draws on earlier work on security levels to control information flow (Denning,...), and on types for π (Milner, Pierce and Sangiorgi,...)

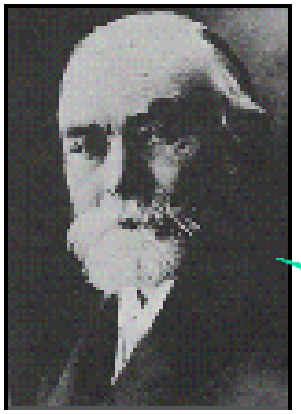
Origins of Type Theory

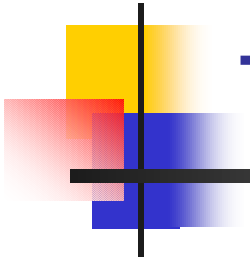
- Cambridge 1901: Russell uncovers a paradox in Frege's system of arithmetic



Your system admits $S=\{x \mid x \notin x\}$.
But is $S \in S$?

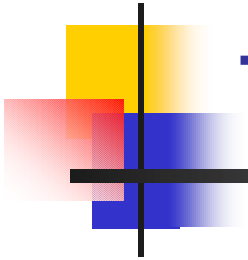
- Later, 1908, he proposes his Theory of Types to patch the bug





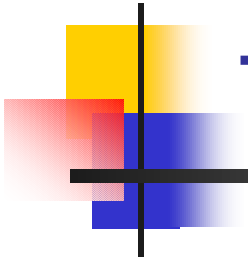
Three Security Levels

- There are three types T for data
  - **Un** for public data known to the opponent
  - **Secret** for private data assumed not to leak to the opponent
  - **Top** for arbitrary data, either **Un** or **Secret**
- Judgment E ⊢ M : T means message M has type T
- Judgment E ⊢ P **ok** means process P well-typed



Typing Public Channels

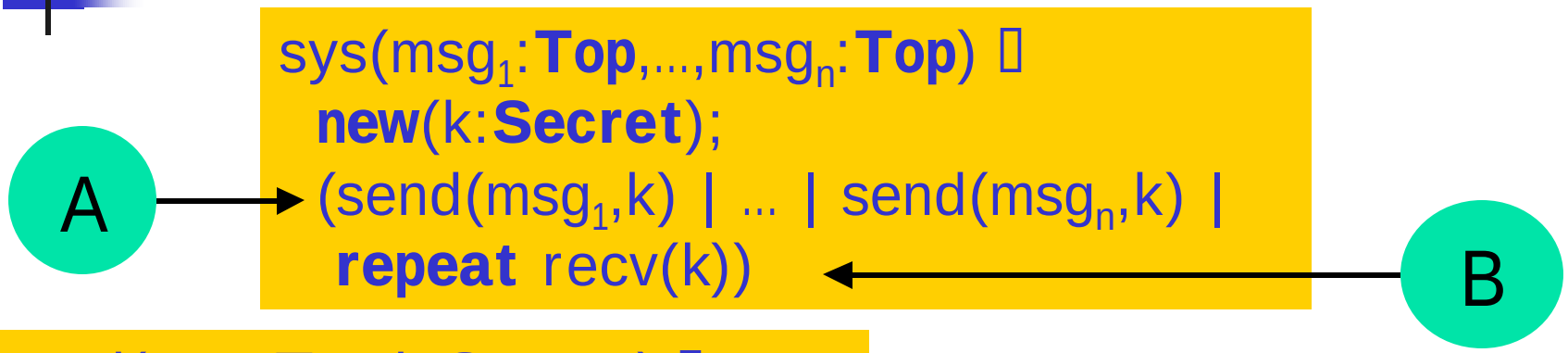
- The rules for communication on public channels
  - allow communication of public **Un** data
  - but prevent communication of **Secret** or **Top** data
- Informally:
  - If M:**Un** and N:**Un** then **out** M N **ok**
  - If M:**Un** and x:**Un** ⊢ P **ok** then **inp** M (x:**Un**);P **ok**



Typing Secret Keys

- The rules for cryptography using secret keys
  - preserve types while sending data on public channels
  - force inclusion of suitable confounders
- Informally:
  - If M_1:**Secret** and M_2:**Top** and M_3:**Un** and N:**Secret** and n a fresh confounder then $\{M_1,M_2,M_3,n\}_N$:**Un**
  - If M:**Un** and N:**Secret** and x_1:**Secret**,x_2:**Top**,x_3:**Un**,x_4:**Top** ⊢ P **ok** then **decrypt** M **as** $\{x_1$:**Secret**,x_2:**Top**,x_3:**Un**,x_4:**Top**$\}_N$;P **ok**

Secrecy for Ex I



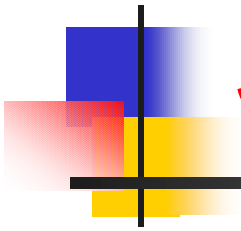

```
send(msg:Top,k:Secret) 
  inp net(no:Un);
  new(co:Secret);
 out net ({_,msg,no,co}k);
```

```
recv(k:Secret) 
  new(no:Un); out net(no); inp net(u:Un);
  decrypt u is {_,msg:Top,no':Un,co'}k;
  check no' is no;
```

Secrecy by Typing

Theorem (Abadi)
If $E=x_1$:**Un**,…,x_n:**Un** and E,y:**Top** $\vdash P\{y\}$ **ok** and $E \vdash M,M'$:**Top** then $P\{M\} \approx P\{M'\}$.

- Beware, does not say no flow from **Secret** to **Un**
- Instead, provided public channels and secret keys have types **Un** and **Secret**, respectively, then **Top** data, e.g., the protocol payload, not observable
- In our example, net:**Un**,msg_1:**Top** $\vdash$ sys(msg_1) **ok**, so we obtain desired secrecy property (also n-ary case)
- Proof relies on *Opponent Typability*; any O can be typed if all its variables of type **Un**



Authenticity Types for Symmetric-Key Crypto

All the work on authenticity types is by Gordon and Jeffrey.

Work on types for correspondence assertions and symmetric-key crypto at MFPS'01, CSFW'01.

Draws on earlier work on effect systems for resource control in functional languages.

Specifying Authenticity

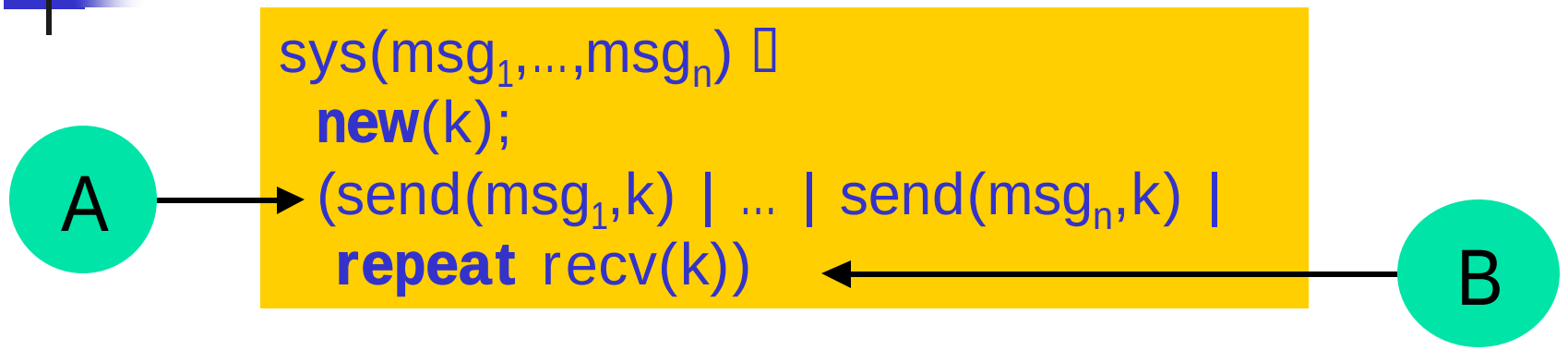
Event 1	A begins	(A sending msg to B)
Message 1	B $\rightarrow$ A:	NB
Message 2	A $\rightarrow$ B:	$\{msg,NB\}_{KAB}$
Event 1'	B ends	(A sending msg to B)

Each end-assertion to have distinct, preceding begin-assertion with same label

Attacks (replays, impersonations) show up as violations of these assertions

Named *correspondence assertions* by Woo and Lam, but also *injective agreements* by Lowe.

Authenticity Specified in Spi

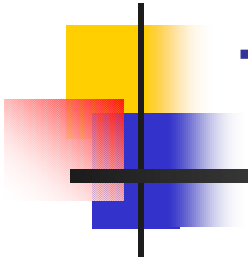


send(msg,k)
inp net(no);
begin “Sender sent msg”;
out net ({msg,no}$_k$);

recv(k)
new(no); **out** net(no); **inp** net(u);
decrypt u **is** {msg,no’}$_k$;
check no’ **is** no; **end** “Sender sent msg”;

The Two Main Judgments

- Judgment $E \vdash M : T$ means message M has type T
- Judgment $E \vdash P : [L_1,\ldots,L_n]$ means process P has *effect* $[L_1,\ldots,L_n]$, 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.

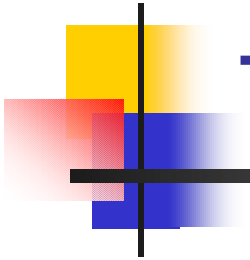


Typing Symmetric Crypto

- We drop **Secret** and **Top** and add some new types, including pairing and tagged unions
- Messages of type **Un** are data known to the untyped opponent, as before
- Messages of type **Key**(T) are names used as symmetric keys for encrypting type T

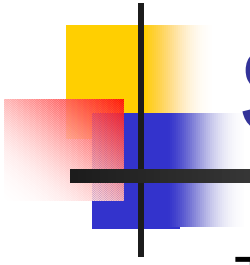
  If M:T and N:**Key**(T) then $\{M\}_N$:**Un**

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



Typing a Nonce Handshake

- Messages of type **Nonce**[L_1,…,L_n] prove begin-events labelled L_1,…,L_n have previously occurred.
- Example nonce lifecycle:
  1. Receiver publishes new N:**Un**
  2. Sender receives N:**Un**, asserts **begin** L, casts nonce into **Nonce**[L], returns within ciphertext
  3. Receiver decrypts message, checks just once for presence of N:**Nonce**[L], then asserts **end** L
- Effect in the **Nonce**[L] type allows transfer
- Each **cast** is a cost; each **check** is a benefit



Semantics of **cast**

The process **cast** x **to** (y:**Nonce** e);P evolves into the process P{y←x}

- Only way to make name of type **Nonce** e
- The name is a proof events in e have happened

It "costs" the effect e:

- If E ⊢ x : **Un** and E, y:**Nonce** e ⊢ P : e'
  then E ⊢ **cast** x **to** (y:**Nonce** e);P : e+e'
- Only kind of type-cast in the system

Semantics of **check**

Process **check** x **is** y;P evolves into process P if x≡y; but otherwise gets stuck.

It "pays for" the effect e in P:

If E ⊢ x : **Nonce** e and E ⊢ y : **Un** and E ⊢ P : e'
then E ⊢ **check** x **is** y; P : e'–e

For each **new**(y:**Un**);P, we require that the name y be used in a **check** at most once

Enforced by adding a new kind of effect; details omitted

Typing Ex I

Msg ⊑ **Un**
MyNonce(m) ⊑ **Nonce** [“Sender sent m”]
MyKey ⊑ **Key** (m:Msg, MyNonce(m))

```
send(msg:Msg,k:MyKey):[] ⊑
  inp net(u:Un);
  begin “Sender sent msg”;
  cast u to (no:MyNonce(msg));
  out net ({msg,no}k);
```

```
recv(k:MyKey):[] ⊑
  new(no:Un); out net(no); inp net(u:Un);
  decrypt u is {msg:Msg,no’:MyNonce(msg)}k;
  check no’ is no; end “Sender sent msg”;
```

Authenticity by Typing

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

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

Theorem (Robust Safety)
If $x_1,\ldots,x_n$:**Un** ⊢ P : [] then P is robustly safe.

For Ex I, we can check the following:

$net,msg_1,\ldots,msg_n$:**Un** ⊢ $sys(msg_1,\ldots,msg_n)$: []

Typing Ex II

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

PrincipalKey(p) □ **Key**(Cipher3(p) + Cipher4(p) + Cipher5(p))

Cipher3(A) □ (B:**Un**, NB:**Nonce**["A proving presence to B"])

Cipher4(B) □ (A:**Un**, cipher:**Un**) --seems redundant

Cipher5(B) □ (A:**Un**, NB:**Nonce**["A proving presence to B"])

Typing Ex II, again

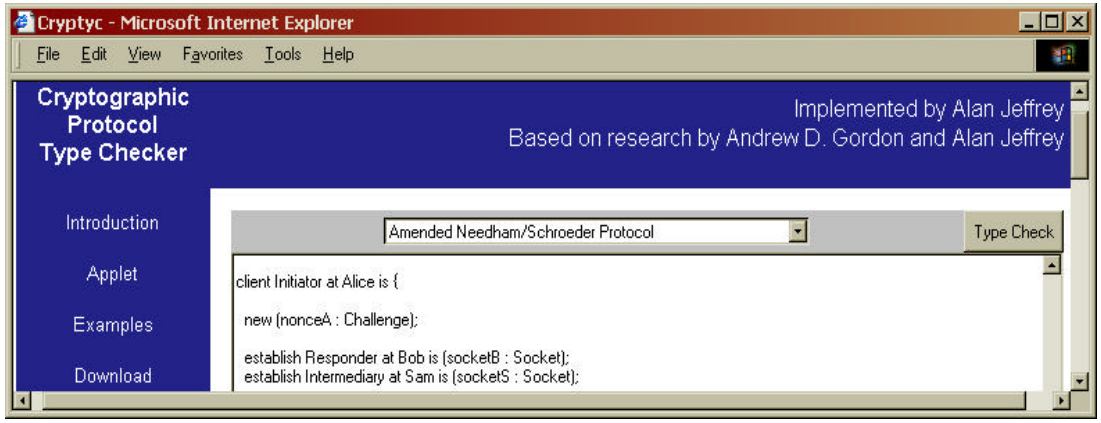
Assertion 1	A begins	"A proving presence to B"
Message 1	A $\rightarrow$ B:	A
Message 2	B $\rightarrow$ A:	NB
Message 3	A $\rightarrow$ B:	$\{tag3(B,NB)\}_{KAS}$
Message 4	B $\rightarrow$ S:	$A,\{tag3(B,NB)\}_{KAS}$
Message 5	S $\rightarrow$ B:	$\{tag5(A,NB)\}_{KBS}$
Assertion 2	B ends	"A proving presence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

PrincipalKey(p) □ **Key**(Cipher3(p) + Cipher5(p))

Cipher3(A) □ (B:**Un**, NB:**Nonce**["A proving presence to B"])

Cipher5(B) □ (A:**Un**, NB:**Nonce**["A proving presence to B"])

Implementation



- Checked a standard suite of symmetric key protocols
- Re-discovered known bugs, found redundancies
- See MSR-TR-2001-49, http://cryptyc.cs.depaul.edu

Authenticity Types for Asymmetric-Key Crypto

Recent work (G&J CSFW'02) on authenticity types for asymmetric-key crypto, including public-key encryption and digital signatures

Secrecy types for asymmetric crypto, Abadi and Blanchet (FOSSACS'01), is the closest related work

Abstracting Asymmetric Crypto

Terms **Enc** k and **Dec** k extract the two parts of a asymmetric key-pair, the name k

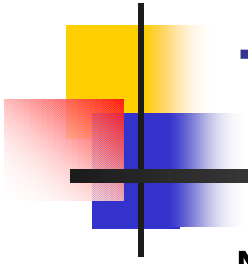
Term $\{|M|\}_N$ is M encrypted with key N

Process **decrypt** L **is** $\{|x|\}_N$;P attempts to decrypt L with key N

$$\textbf{decrypt}\ \{|M|\}_{\textbf{Enc}\ k}\ \textbf{is}\ \{|x|\}_{\textbf{Dec}\ k};P \rightarrow P\{x \leftarrow M\}$$

Fairly standard model; has known limitations

Same operational semantics models both public-key crypto and digital signature applications



Typing Asymmetric Crypto

Names of type **KeyPair**(T) represent a key-pair for transforming T data.

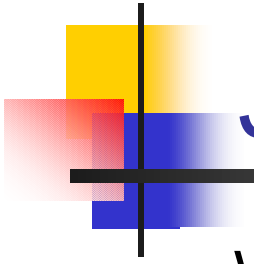
Terms of type **EncKey**(T) and **DecKey**(T) are encryption and decryption keys, respectively.

If p:**KeyPair**(T) then **Enc** p:**EncKey**(T).

If p:**KeyPair**(T) then **Dec** p:**DecKey**(T).

If M:T and N:**EncKey**(T) then $\{M\}_N$:**Un**.

If M:**Un** and N:**DecKey**(T) and x:T ⊢ P well-typed, then so is **decrypt** M **as** $\{x\}_N$;P.



Jargon: Public versus Tainted

We (re-) introduce a subtype order T<:U

If M:T and T<:U then M:U

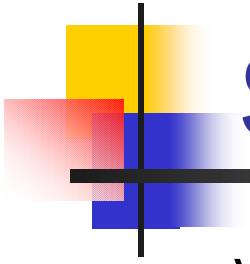
Hence, we characterize data that may flow to or from the opponent:

Let a type T be **public** iff T<:**Un**

Let a type T be **tainted** iff **Un**<:T

Ex: **Un** is both public and tainted

Ex: **Top** is tainted but not public

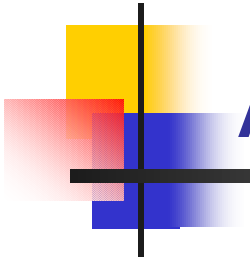


Subtyping Asymmetric Keys

Variance rules reminiscent of types for input and output channels (see Pierce and Sangiorgi)

- If T<:U then **EncKey**(U)<:**EncKey**(T) (contravariant)
- If T<:U then **DecKey**(T)<:**DecKey**(U) (covariant)
- **KeyPair**(T) neither co- nor contravariant.

For both Opponent Typability and to allow publication of keys for **Un**, both **EncKey**(**Un**) and **DecKey**(**Un**) are tainted and public.



Analysing our Abstraction

Can prove the following:

(PK) **EncKey**(T) public iff T tainted

(DS) **DecKey**(T) public iff T public

So how can we apply key-pairs of type **KeyPair**(T)

If (PK) but not (DS): public-key crypto

If (DS) but not (PK): digital signature

If both (PK) and (DS): have T<:>**Un**, beware!

If neither, model degenerates to symmetric key

Ex III: Authentication by Certs

Server A authenticates to client B via certificate from CA

Assertion 1	A begins	(A sending msg to B)
Message 1	B $\rightarrow$ A:	NB
Message 2	A $\rightarrow$ B:	$\{A,KA\}_{KCA^{-1}},\{msg,B,NB\}_{KA^{-1}}$
Assertion 2	B ends	(A sending msg to B)

pKCA, pKA -- key-pairs
KCA $\in$ **Dec** pKCA -- CA's verification key (known to B)
$KCA^{-1} \in$ **Enc** pKCA -- CA's private signing key
KA $\in$ **Dec** pKA -- A's verification key (initially unknown)
$KA^{-1} \in$ **Enc** pKA -- A's private signing key

Ex III: Types for the Key-Pairs

AuthMsg(A) ≡
  (msg:T, B:**Un**, N:**Nonce**[(A sending msg to B)]
pKCA: **KeyPair** (A:**Un**, KA: **DecKey**(AuthMsg(A)))
pKA: **KeyPair** (AuthMsg(A))
KCA^{-1}: **EncKey** (A:**Un**, KA: **DecKey**(AuthMsg(A)))

As before, a name of type **Nonce**[(A sending msg to B)] bears witness to a distinct preceding begin-event labelled [(A sending msg to B)]

(DS) applies to both key-pairs, since AuthMsg(A) and **DecKey**(AuthMsg(A)) are public (assuming T public)

So verification keys public, signing keys private

Type-checking verifies that A can authenticate to B

Authenticity Types: Two Refinements

With symmetric-key protocols, nonces can be public.

With public-key, nonces may need to be private.

Hence, we need new nonce types and new effects.

Public Nonces Insufficient

Assertion 1	A begins	(A sending msg to B)
Message 1	B $\rightarrow$ A:	NB
Message 2	A $\rightarrow$ B:	$\{A,msg,NB\}_{KB}$
Assertion 2	B ends	(A sending msg to B)

Encryption with B's public key

Replaced symmetric encryption with asymmetric

B has now no reason to believe message 2 from A

Unsafe, and indeed fails to type-check

(DS) rather than (PK) holds

since payload type is public but untainted

Ex IV: Encrypt Outgoing Nonce

Assertion 1	A begins	(A sending msg to B)
Message 1	B $\rightarrow$ A:	$\{NB\}_{KA}$
Message 2	A $\rightarrow$ B:	$\{A,msg,NB\}_{KB}$
Assertion 2	B ends	(A sending msg to B)

Now, B reasons that since only A can obtain NB from $\{NB\}_{KA}$, A must have sent Message 2.

This protocol is safe.

To type-check it, we need new secret but tainted types for the nonce challenge and response.

Typing Private, Tainted Nonces

AuthMsg(P)
  msg1(N: **PrivChall**[]) +
  msg2(msg:**Top**, Q:**Un**,
        N:**PrivResp**[(Q sending msg to P)])
KA: **EncKey**(AuthMsg(A))
KB: **EncKey**(AuthMsg(B))

Names of type **PrivChall**[] are private but tainted challenges

Names of type **PrivResp**[L] are private but tainted responses, witness to a distinct begin-event L

With these typings, can verify the protocol

For (PK), taint AuthMsg(P), by assuming msg:**Top**

Adding Trust Effects

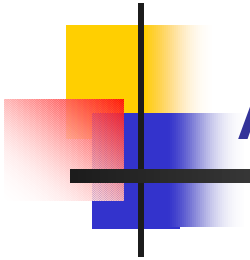
```
AuthMsg(P) 
  msg1(N: PrivChall[]) +
  msg2(msg:Top, Q:Un,
          N:PrivResp[(Q sending msg to P), msg:T]
KA: EncKey (AuthMsg(A))
KB: EncKey (AuthMsg(B))
```

The effect msg:T asserts the existing name msg has type T

Before checking the nonce, B knows only that msg:**Top**

If the nonce-check fails, B knows nothing more about msg

If it succeeds, B can downcast msg to type T



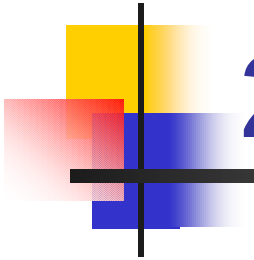
Assessment of Authenticity Types

Benefits

- Familiar program/type-check/debug cycle
- Little human effort per protocol
- No bound on size of opponent or protocol
- Types are as intuitive as BAN formulas
- Directly check implementations

Limitations

- No automatic discovery of attacks
- Type inference problem still open
- Usual Dolev-Yao perfect encryption assumptions
- Incompletenesses, like any type system



2: Conclusions

The Dolev-Yao threat model of crypto protocols may be formalized within process calculi

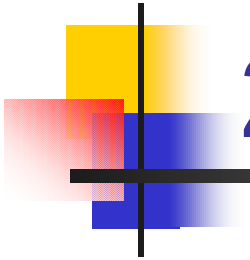
Secrecy and authenticity properties may be formalized using behavioural equivalences and events

Suitable type systems may establish both, even in the presence of an untyped attacker

There are many other formal methods; see our papers for a discussion

DePaul/MSRC Cryptyc Project working to develop type-checked protocol implementations

DePaul funded from NSF Trusted Computing program



2: Resources

- Cryptyc
  - http://cryptyc.cs.depaul.edu
  - http://research.microsoft.com/~adg/cryptyc.htm
- Blanchet's Proverif verification tool
  - http://www.mpi-sb.mpg.de/~blanchet/

End of
Part 2

Secure Global Computing with XML Web Services: Theory and Practice

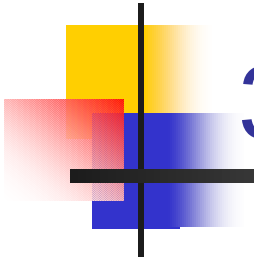
Andy Gordon

Microsoft Research

EEF Global Computing Summer School

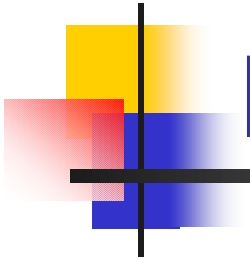
Edinburgh, Scotland, July 7-11, 2003

Lecture 3, Wednesday 9:00-10:30



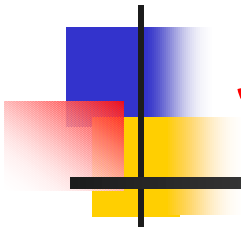
3: Analysing web services security

- Validating a security abstraction by typing
- XML with symbolic crypto
- Username signing example
- X509 signing example
- All part of our Samoa project...



MSRC Samoa Project

- **Goal:** exploit advances in the analysis of security protocols in the practical setting of XML web services.
- Outcomes so far:
  - An implementation of declarative security attributes for web services
    - A. Gordon and R. Pucella, *Validating a web service security abstraction by typing*. In **2002 ACM Workshop on XML Security**, Nov 2002
  - Design of a logic-based approach to checking SOAP-based protocols
    - K. Bhargavan, C. Fournet, A. Gordon, *A semantics for web services authentication*, in preparation
  - Actionable feedback to internal group
- **http://Securing.WS**



Part I: A Web Service Security Abstraction

With Riccardo Pucella

An informal design, and pre-WS-Security implementation

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 Security Abstraction

```
class BankingServiceClass {
  string callerid;

  [WebMethod] [SecurityLevel(Level=Auth)]
  public int Balance (int account)

  [WebMethod] [SecurityLevel(Level=AuthEnc)]
  public string Statement (int account) }
```

- Each web method has one of three security levels
  - None, Auth or AuthEnc
- Akin to SRC Secure Network Objects, for example
- Enough to support various authorisation mechanisms

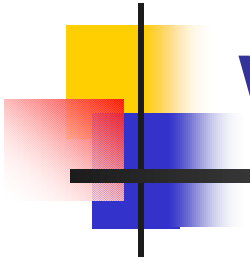
A SOAP-Level Implementation

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

- We assume key KAB shared between A and B
  - We also 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 types:
  - We formalise the abstraction as an object calculus
  - We specify its semantics by translation into spi
  - Since the translation preserves typings, attacks representable in spi are impossible
  - Verification of formal model, not running code
    - Still, verified implementations are coming within reach

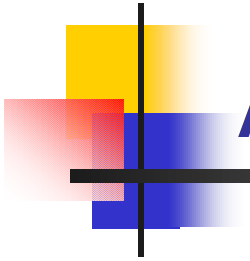
Back to our SOAP protocol

Specify authenticity via event **correspondences**

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

Verify via generic types for crypto keys and nonces

```
SharedKey(a,b)  Key(Union(
  req(w:Un, m:Un, t:Un, Nb:Nonce[end req(a,b,w,m,t)]),
  res(w:Un, n:Un, t:Un, Na:Nonce[end res(a,b,w,n,t)])))
```



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 “Featherweight Java”
    - 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

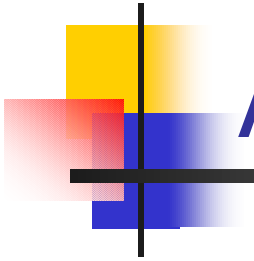
$c \in Class$	class name
$f \in Field$	field name
$l \in Meth$	method name
$p \in Prin$	principal name
$w \in WebService$	service name

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

A, B ::=	type
Id	principal
c	object
$sig ::= B(A_1\ x_1,\ldots,A_n\ x_n)$	signature

x,y,z	variable
u,v ::=	value
x	variable
null	null
new $c(v_1,\ldots,v_n)$	object
p	principal

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



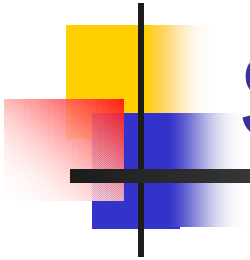
An Informal Semantics

- 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_{true} **else** a_{false}, evaluate $a_{u=v}$ as p
  - To evaluate v.f, when v=**new** c(v_1,..., v_n) and f is the i^{th} field of c, terminate with v_i
  - To evaluate v.l(u_1,...,u_n), when v=**new** c(v_1,..., v_n) and l in c has signature B(A_1 x_1,...,A_n x_n) and body b{this,x_1,...,x_n}, evaluate b{v,u_1,...,u_n} as p
  - To evaluate w:l(u_1,...,u_n), evaluate the method call **new** *class*(w)(p).l(u_1,...,u_n) as *owner*(w)

A Formal Semantics

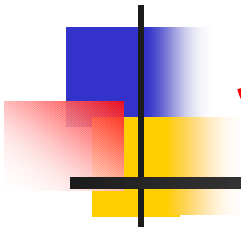
- We map type B to spi message type ⟦B⟧
- We map value v to spi message ⟦v⟧
- We map body b running as p to spi process ⟦b⟧pk where k is a continuation channel
- We represent SOAP envelopes as spi messages
- We represent security guarantees by embedding begin- and end-assertions
- These security guarantees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

Theorem (Type Preservation)
If E ⊢ b : B then ⟦E⟧, k:**Ch**(⟦B⟧) ⊢ ⟦b⟧pk : []



Summary of Part I

- Coding experiments in 2002, pre-WS-Security, showed we could implement an abstraction of SOAP-level security
- Formal model shows Cryptyc can verify basic design
  - A novel approach to verifying secure RPC
  - Exposes some limitations of the type theory
    - No good model of compromised insiders
- A criticism of this abstract approach to protocol verification is that it's not clear which details are safe to omit
  - Hence, we are developing a version of the pi calculus that directly embeds XML messages with crypto



Part II: XML with Symbolic Crypto

With K. Bhargavan and C. Fournet

We develop a symbolic Prolog-like notation for XML and predicates on XML

For example, we can write predicates defining envelopes, username tokens, and signatures

Hence, we will subsequently be able to specify security protocols that use such tokens

Our XML Model Part 1

XML Data Model: Standard Core

Label	::= `anyLegalXmlName`	element or attribute name
String : **string**	::= `any legal XML string`	XML string
Att : **att**	::= *Label*="*String*"	attribute
Atts : **atts**	::= *Att Atts* | ε	attribute sequence
Item : **item**	::= *Element* | *String*	item
Items : **items**	::= *Item Items* | ε	item sequence
Element	::= <*Label Atts*>*Items*</*Label*>	element

- Represents valid, parsed XML
- Sorts **string**, **att**, **atts**, **item**, **items**, plus some others
- Adapted from Siméon and Wadler's model (POPL'03)
- Resembles the W3C Infoset recommendation

Logical Predicates

Syntax of Formulas and Predicate Definitions:

$\Phi ::=$	formula
$V = T$	term comparison
$U \in V$	list membership
$p(\widetilde{V})$	predicate instance
Φ_1, Φ_2	conjunction
$p(\widetilde{x}) :- \Phi_1 \vee \cdots \vee \Phi_n$	definition of predicate p with $n > 0$

- A Horn logic over our many-sorted algebra
  - primitive formulas for equality and list membership, but no recursively-defined predicates
- Given certain implementability constraints, logic programs may be compiled into Abadi and Fournet's applied pi calculus
  - Much like spi, but parametric on the algebra of values

Ex I: A SOAP request

- The predicate *hasBody*(**item**, *b*) below means *b* is the body of envelope *env* (the wildcard _ matches anything):

$$hasBody(env : \mathbf{item}, b : \mathbf{item}) \text{ :-}$$
$$env = \texttt{<Envelope><Header>\_</>}b\texttt{</>},$$
$$b = \texttt{<Body \_>\_</>}.$$

- Next, a body requesting info on *OrderId*:

$$isGetOrder(b : \mathbf{item}, OrderId : \mathbf{string}) \text{ :-}$$
$$b = \texttt{<Body \_>}$$
$$\texttt{<GetOrder>}$$
$$\texttt{<orderId>}OrderId\texttt{</>}$$

- Overall:

$$soapGetOrder(env : \mathbf{item}, OrderId : \mathbf{string}) \text{ :-}$$
$$hasBody(env, b), isGetOrder(b, OrderId).$$

Ex II: A SOAP Response

- Body of the response concerning *OrderId* owned by user *u*

$$
\begin{aligned}
&\mathit{isGetOrderResponse}(b : \mathbf{item}, \mathit{OrderId} : \mathbf{string}, u : \mathbf{string}) \ :- \\
&\quad b = \texttt{<Body \_>} \\
&\qquad \texttt{<GetOrderResponse>} \\
&\qquad\quad \texttt{<orderId>}\mathit{OrderId}\texttt{</>} \\
&\qquad\quad \texttt{<date>\_</>} \\
&\qquad\quad \texttt{<userId>}u\texttt{</>} \\
&\qquad\quad \_
\end{aligned}
$$

- Overall:

$$
\begin{aligned}
&\mathit{soapGetOrderResponse}(\mathit{env} : \mathbf{item}, \mathit{OrderId} : \mathbf{string}, u : \mathbf{string}) \ :- \\
&\quad \mathit{hasBody}(\mathit{env}, b), \mathit{isGetOrderResponse}(b, \mathit{OrderId}, u).
\end{aligned}
$$

Need for Symbolic Crypto

- To specify interesting security properties, our predicates need to talk about cryptographically significant byte arrays encoded as strings within XML, i.e., security tokens
- For example, a UsernameToken may be added to a security header to identify the origin of a message
  - Nonce contains the Base64-encoding of random number
  - Password contains the Base64-encoding of a secure hash of the shared password, the nonce, and the timestamp
- To do this, we add new sort **bytes**, and extend **string**

```
<UsernameToken>
        <Username>adg</>
        <Password>Ouywn2V6ikNNtWYL29gI9R3CPBk=</>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
        <Created>2003-02-04T16:49:45Z</>
```

Our XML Model Part 2

XML Data Model: Byte Arrays, Symbolic Cryptographic Digests

Bytes : **bytes** ::=	byte array (not itself XML)
s	abstract name, such as a key or nonce
concat$(Bytes_1, Bytes_2)$	array concatenation
c14n$(Item)$	canonical bytes of an item
utf8$(String)$	UTF8 representation of strings
sha1$(Bytes)$	cryptographic digest
p-sha1$(String_{pwd}, Bytes_{salt})$	key from salted password
hmac-sha1$(Bytes_{key}, Bytes_{subj})$	keyed hash
String : **string** ::=	XML string
s	abstract name, such as a password
base64$(Bytes)$	Base64-encoding of byte array
principal$(String_{pwd})$	function from password to principal

Inverses, Equations

Equivalence of Terms of the Data Model: $U = V$

$$\mathbf{hd}(Att\ Atts) = Att \qquad \mathbf{tl}(Att\ Atts) = Atts$$
$$\mathbf{hd}(Item\ Items) = Item \qquad \mathbf{tl}(Item\ Items) = Items$$
$$\mathbf{i\text{-}base64}(\mathbf{base64}(Bytes)) = Bytes$$
$$\mathbf{i\text{-}utf8}(\mathbf{utf8}(String)) = String$$
$$\mathbf{i\text{-}c14n}(\mathbf{c14n}(Item)) = Item$$
$$\mathbf{fst}(\mathbf{concat}(Bytes_1, Bytes_2)) = Bytes_1$$
$$\mathbf{snd}(\mathbf{concat}(Bytes_1, Bytes_2)) = Bytes_2$$

Plus destructors for elements and attributes

- Following Dolev and Yao, model includes no inverses for cryptographic “one-way” functions
  - **sha1**, **p-sha1**, **hmac-sha1**
- Nor for the function modelling user/password databases
  - **principal**

Ex: User Token with Digest

```
<UsernameToken>
        <Username>adg</>
        <Password>Ouywn2V6ikNNtWYL29gl9R3CPBk=</>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
        <Created>2003-02-04T16:49:45Z</>
```

$$
\begin{aligned}
&\mathit{isDigestUserToken}(\mathit{tok} : \mathbf{item}, u : \mathbf{string}, \mathit{pwd} : \mathbf{string}, n : \mathbf{bytes}, t : \mathbf{string}) \ :- \\
&\quad \mathit{tok} = \texttt{<UsernameToken \_>} \\
&\qquad\qquad \texttt{<Username>}u\texttt{</>} \\
&\qquad\qquad \texttt{<Password>}\mathbf{base64}(d)\texttt{</>} \\
&\qquad\qquad \texttt{<Nonce>}\mathbf{base64}(n)\texttt{</>} \\
&\qquad\qquad \texttt{<Created>}t\texttt{</>}, \\
&\quad u = \mathbf{principal}(\mathit{pwd}), \\
&\quad d = \mathbf{sha1}(\mathbf{concat}(n, \mathbf{concat}(\mathbf{utf8}(t), \mathbf{utf8}(\mathit{pwd})))).
\end{aligned}
$$

Ex: A Security Header

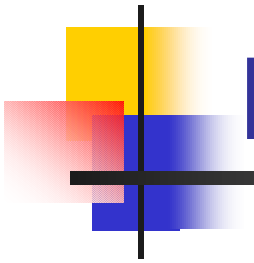
- User tokens occur within an envelope's security header:

$$
\begin{aligned}
&\mathit{hasSecurityHeader}(\mathit{env} : \mathbf{item}, \mathit{toks} : \mathbf{items}) \ :- \\
&\quad \mathit{env} = \texttt{<Envelope><Header>}\mathit{headers}\texttt{</>_</>}, \\
&\quad \mathit{header} \in \mathit{headers}, \\
&\quad \mathit{header} = \texttt{<Security>}\mathit{toks}\texttt{</>}.
\end{aligned}
$$

- Hence, we can stipulate that an envelope contains a user token with a password digest:

$$
\begin{aligned}
&\mathit{hasUserTokenDigest}(e : \mathbf{item}, u, \mathit{pwd} : \mathbf{string}, \\
&\qquad\qquad\qquad\qquad n : \mathbf{bytes}, t : \mathbf{string}, b : \mathbf{item}) \ :- \\
&\quad \mathit{hasSecurityHeader}(e, \mathit{toks}), \\
&\quad \mathit{utok} \in \mathit{toks}, \\
&\quad \mathit{isDigestUserToken}(\mathit{utok}, u, \mathit{pwd}, n, t), \\
&\quad \mathit{hasBody}(e, b).
\end{aligned}
$$

Discussion: Password Digests



- We are assuming underlying transport is unencrypted
- What does $hasUserTokenDigest(e,u,pwd,n,t,b)$ tell us?
- Not very much in itself:
  - That u recently possessed the password pwd
  - Assuming that t is recent, that S doesn't itself generate such tokens, and that only S and u possess pwd
  - Nothing in $\mathbf{sha1}(\mathbf{concat}(n,\mathbf{concat}(\mathbf{utf8}(t),\mathbf{utf8}(pwd))))$ specific to body b, so cannot infer u meant to send b to S
  - Worse, since n and t are public, the digest invites a dictionary attack to recover the pwd (outside our model)
- Explains why, to authenticate the message, we need to add a signature of the body based on a key derived from pwd

Signing Key from User Token

- Following stipulates the key k derived from the password, timestamp, and nonce; unlike the digest, k must be secret

$$
\begin{aligned}
&\mathit{isUserTokenKey}(\mathit{tok} : \textbf{item}, u, \mathit{pwd} : \textbf{string},\\
&\qquad\qquad n : \textbf{bytes}, t : \textbf{string}, k : \textbf{bytes}) \ :-\\
&\quad \mathit{tok} = \texttt{<UsernameToken \_>}\\
&\qquad\quad \texttt{<Username>}u\texttt{</>}\ \_\\
&\qquad\quad \texttt{<Nonce>}\textbf{base64}(n)\texttt{</>}\\
&\qquad\quad \texttt{<Created>}t\texttt{</>}\\
&\quad u = \textbf{principal}(\mathit{pwd}),\\
&\quad k = \textbf{p-sha1}(\mathit{pwd}, \textbf{concat}(n, \textbf{utf8}(t))).
\end{aligned}
$$

Formalizing 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

$$ref(t : \mathbf{item}, r : \mathbf{item}) \text{ :-}$$
$$r = \texttt{<Reference \_>}$$
$$\texttt{\_ \_ <DigestValue>}\mathbf{base64}(\mathbf{sha1}(\mathbf{c14n}(t)))\texttt{</>}.$$

- 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="#..."/>
```

$$isSigVal(sv : \mathbf{bytes}, si : \mathbf{item}, k : \mathbf{bytes}, a : \mathbf{string}) \text{ :-}$$
$$a = \texttt{hmac-sha1}, sv = \mathbf{hmac\text{-}sha1}(k, \mathbf{c14n}(si)).$$

$$isSigInfo(si : \mathbf{item}, a : \mathbf{string}, x_1, \ldots, x_n : \mathbf{item}) \text{ :-}$$
$$si = \texttt{<SignedInfo>\_<SignatureMethod Algorithm="}a\texttt{"></>}r_1 \ \ldots \ r_n\texttt{</>},$$
$$ref(x_1, r_1), \ldots, ref(x_n, r_n).$$

$$isSignature(sig : \mathbf{item}, a : \mathbf{string}, k : \mathbf{bytes}, x_1, \ldots, x_n : \mathbf{item}) \text{ :-}$$
$$sig = \texttt{<Signature>}si\ \texttt{<SignatureValue>}\mathbf{base64}(sv)\texttt{</>\_</>},$$
$$isSigInfo(si, a, x_1, \ldots, x_n), isSigVal(sv, si, k, a).$$

Username Signed Message

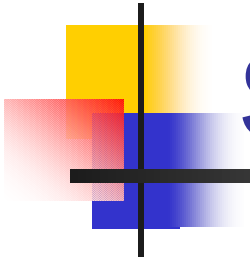
- We need one final predicate:

$$
\begin{aligned}
&\mathit{hasUserSignedBody}(e : \textbf{item}, u : \textbf{string}, \mathit{pwd} : \textbf{string},\\
&\qquad\qquad n : \textbf{bytes}, t : \textbf{string}, b : \textbf{item}) \ :-\\
&\quad \mathit{hasBody}(e, b),\\
&\quad \mathit{hasSecurityHeader}(e, \mathit{toks}),\\
&\quad \mathit{utok} \in \mathit{toks},\\
&\quad \mathit{isUserTokenKey}(\mathit{utok}, u, \mathit{pwd}, n, t, k),\\
&\quad \mathit{sig} \in \mathit{toks},\\
&\quad \mathit{isSignature}(\mathit{sig}, \texttt{hmac-sha1}, k, b).
\end{aligned}
$$

WS-Security Protocol 1

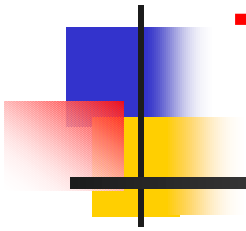
Event 1	I logs	$begin(A,n,t,orderid)$
Message 1	I $\rightarrow$ S	e_1 where $hasUserSignedBody(e_1,A,p,n,t,b)$ and $isGetOrder(b,orderid)$
Event 1'	S logs	$end(A,n,t,orderid)$
Message 2	S $\rightarrow$ I	e_2 where $soapGetOrderResponse\ (e_2, orderid,A)$

- Authentication formalized as a correspondence; authorization decision not formalized
- We describe this protocol as a process Q, and take the opponent O to be any arbitrary process in parallel
  - Details omitted; this is much as in Lecture 2
- **Theorem:** $Q \mid O$ is safe, that is, in every run, every end-event corresponds to a preceding begin-event
- Proofs use a combination of process calculus techniques, and are compositional



Summary of Part II

- We propose a method for analysing SOAP-level protocols
  - Identify the principals taking part in the protocol
  - Describe the exchange of messages, and the events that are to be in correspondence
  - Formalize security checks using predicates in our XML model with symbolic crypto
  - Define applied pi processes to represent behaviour of principals, by interpreting predicates as processes
  - Prove robust safety
    - In our paper, we use standard process calculus techniques
    - We are also exploring automatic techniques



Part III: X509 Security Tokens

We can handle protocols based on public-key signatures, and also protocols relying on SOAP intermediaries

Our XML Model Part 3

Additions for Public-Key Certificates and Signatures:

Bytes : **bytes** ::=	byte array
pk$(Bytes_{priv})$	public key derivation
x509$(Bytes_{priv}, String_u, String_a, Bytes_k)$	X509 cert
check-x509$(Bytes_{cert}, Bytes_{pub})$	X509 cert verification
x509-key$(Bytes_{cert})$	public key in X509 cert
rsa-sha1$(Bytes_{orig}, Bytes_{priv})$	public key signature
check-rsa-sha1$(Bytes_{orig}, Bytes_{sig}, Bytes_{pub})$	public key verification
String : **string** ::=	XML string
x509-user$(Bytes_{cert})$	name in X509 cert
x509-alg$(Bytes_{cert})$	algorithm in X509 cert

Our XML Model Part 3

Additional Equations for Public-Key Cryptography:

$$\textbf{check-x509}(\textbf{x509}(T_{priv}, T_u, T_a, T_k), \textbf{pk}(T_{priv})) = \textbf{pk}(T_{priv})$$
$$\textbf{x509-user}(\textbf{x509}(T_{priv}, T_u, T_a, T_k)) = T_u$$
$$\textbf{x509-alg}(\textbf{x509}(T_{priv}, T_u, T_a, T_k)) = T_a$$
$$\textbf{x509-key}(\textbf{x509}(T_{priv}, T_u, T_a, T_k)) = T_k$$
$$\textbf{check-rsa-sha1}(T_{orig}, \textbf{rsa-sha1}(T_{orig}, T_{priv}), \textbf{pk}(T_{priv})) = \textbf{pk}(T_{priv})$$

- Now, we can define well-formed X509 security tokens:

$$isX509Token(tok : item, k_r : bytes, u : string, a : string, k : bytes) :-$$
$$tok = \texttt{<BinarySecurityToken ValueType="X509v3" \_>}\textbf{base64}(xcert)\texttt{</>},$$
$$\textbf{check-x509}(xcert, k_r) = k_r,$$
$$u = \textbf{x509-user}(xcert), a = \textbf{x509-alg}(xcert), k = \textbf{x509-key}(xcert).$$

X509 Signed Message

- We add a new clause to the definition of *isSigVal* and define a new top-level predicate

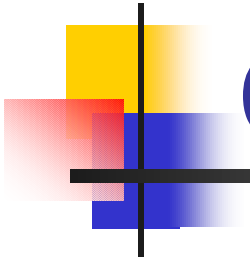
$$isSigVal(sv : bytes, si : item, p : bytes, a : string) :-$$
$$a = \texttt{rsa-sha1}, \textbf{check-rsa-sha1}(\textbf{c14n}(si), sv, p) = p.$$

$$hasX509SignedBody(e : \textbf{item}, k_r : \textbf{bytes}, u, ac, to : \textbf{string},$$
$$b, ea, et, ei : \textbf{item}) :-$$
$$hasBody(e, b),$$
$$hasPathHeader(e, ac, to, id, ea, et, ei),$$
$$hasSecurityHeader(e, toks),$$
$$xtok \in toks,$$
$$isX509Token(xtok, k_r, u, \texttt{rsa-sha1}, k),$$
$$sig \in toks,$$
$$isSignature(sig, \texttt{rsa-sha1}, k, b, ea, et, ei).$$

WS-Security Protocol 2

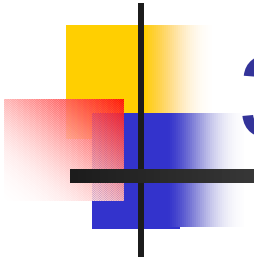
Event 1	I logs	$begin(A,n,t,orderid)$
Message 1	I $\rightarrow$ S	e_1 where $isGetOrder(b,orderid)$ and $hasX509SignedBody(e_1,k_r,A,W,S,b,ea,et,ei)$
Event 1'	S logs	$end(A,n,t,orderid)$
Message 2	S $\rightarrow$ I	e_2 where $soapGetOrderResponse\ (e_2, orderid,u)$

- As before, we describe this protocol as a process Q, and take the opponent O to be any arbitrary process in parallel
- **Theorem:** $Q \mid O$ is safe, that is, in every run, every end-event corresponds to a preceding begin-event



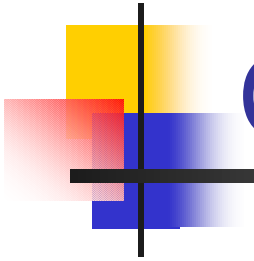
Contributions of the Paper

- Details and theorems for examples:
  - Username/password signatures
  - X509 signatures
  - Firewall-based authentication
- Application of three standard security principles:
  - Use explicit syntax for cryptographic transforms
  - Identify explicit goals (not just secrecy...)
  - Separate verification from discovery of evidence
- Some advice: beware weak passwords, verify all headers have been signed, use short frag URIs



3: Conclusions, Futures

- Successfully bridged gap between theoretical pi threat model and XML used in WS security protocols
  - Began with abstract view of SOAP, but was advantageous to work with direct XML moel
  - Put effort into real samples, eg, MS Pet Shop
  - Found attacks within threat model
  - Proved wire-level theorems about protocols
- Next step, automated analyses within our new symbolic model of XML security protocols
  - Many potential users of WS-Security
    - BPEL4WS, OGSA, ...
  - Not many “best practices” just yet
    - Lots of standard syntax, little standard semantics



Overall Summary

- Unlike other initiatives, safe bet that web services will be widely deployed
  - If Grid happens, it will be WS-based
  - If Semantic Web happens, ditto!
- Moreover, web service security engineering presents additional challenges
- Subtlety and lack of standardized semantics for WS-Security great opportunity to exploit successful development of formal methods for crypto

End of
Part 3