

A Calculus for Cryptographic Protocols

Andy Gordon, *Microsoft Research*

Summer School on
Foundations of Internet Security

June 14-23, 2002, Duzniki Zdrój, Poland

1

Aims of the Course

- I want to help get you up to speed on recent advances on applying language theory to problems in computer security
- Security specific models have proved very effective...
- ...but I want to emphasise the benefits of general computational models
- And get you interested in making new advances yourselves!

2

Perspective

- Technology much less than half the battle; many, many process issues
- Certainty out of reach; need cost effective tools to improve reliability
- Still, many interesting language-related security issues within reach of formal analysis
- Quest for situations where without too much human effort we can find critical bugs

3

Acknowledgements

- I've drawn the material in this course from several books and articles
- I've attempted to credit all the authors whose work is directly reported
- Still, the context of all these works is a thriving research community
- In these lectures there is sadly no time to cover all the indirect influences or all the related work

4

What is a Cryptographic Protocol?

- It's not a crypto algorithm
 - Such as DES, RSA, RC4, etc
 - See Preneel & Rogaway's lectures
- It's a communication protocol using crypto algorithms to protect against eavesdropping and tampering
- Ex I is a demo showing password-based authentication between a windows-based client and a web server

5

```
private void button1_Click(object sender, System.EventArgs e)
{
    string principal = principalBox.Text;
    string password = passwordBox.Text;
    string request = requestBox.Text;
    localhost.Service1 ws = new localhost.Service1();
    int nonce = ws.GetNonce();
    Bytes plain = new Bytes(password, 16);
    Bytes cipher = Bytes.Concat(Bytes(nonce), new Bytes(request));
    Bytes cipher = plain.Encrypt(key);
    string reply = ws.MakeEncRequest(principal, cipher.Val);
}

[WebMethod(Description="Issue request, encrypted, freshened by the nonce.")
public string MakeEncRequest(string principal, byte[] cipher)
{
    if (keys.ContainsKey(principal))
        return "Denied: Never heard of principal '"+principal+"'";
    Bytes key = Bytes.KeyFromPrincipal(principal);
    Bytes plain = new Bytes(cipher).Decrypt(key);
    if (plain.Equals(" "))
        return "Denied: Decryption failed";
    int nonce = (int)plain.Char(0).AsUInt16();
    string request = plain.Char(1).AsString();
    if (nonce.ContainsKey(nonce))
        return "Denied: Never heard of nonce '"+nonce+"'";
    if ((bool)nonce.Equals(nonce))
        return "Denied: Nonce '"+nonce+"' already used";
    nonce.ContainsKey(nonce);
    return "Accepted: "+request+" with nonce '"+nonce+"'";
}

POST /WhipWebService/Service1.asmx HTTP/1.1 Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: http://tempuri.org/MakeEncRequest
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://schemas.xmlsoap.org/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
<soap:Body>
<MakeEncRequest xmlns="http://tempuri.org/"
soap:encoding="http://schemas.xmlsoap.org/soap/encoding">
<MakeEncRequest
xmlns="http://tempuri.org/"
soap:base64Binary="cipher">
<MakeEncRequest
xmlns="http://tempuri.org/"
soap:base64Binary="cipher">
</soap:Body>
</soap:Envelope>
```

Ex I: Discussion

- We needed to include the nonce challenge to prevent *replay attacks*
- Passwords are not good keys, because easily guessable via a *dictionary attack*
- Better: turn password into a strong key via an *authentication server*
- Is there a more abstract notation in which to express just the crucial details of the protocol?

7

Ex II: Server-Based Login

```

    graph LR
      A[PDA a knows kas] <--> B[Network b knows kbs]
      B <--> S[Server s knows kas and kbs]
  
```

- 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
- We use a standard message sequence notation

8

Ex II: Woo and Lam (1991)

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

9

Attacking Ex II

1. $c \rightarrow b: a$	1. $c \rightarrow b: c$
2. $b \rightarrow c: nb_a$	2. $b \rightarrow c: nb_c$
3. $c \rightarrow b: \{nb_a\}_{kcs}$	3. $c \rightarrow b: \{nb_a\}_{kcs}$
4. $b \rightarrow s: b, \{a, \{nb_a\}_{kcs}\}_{kbs}$	4. $b \rightarrow s: b, \{c, \{nb_a\}_{kcs}\}_{kbs}$
5. $s \rightarrow b: \{\dots\}_{kbs}$	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.

10

Ex II: Discussion

- We include the additional names to prevent an *impersonation attack*
- The message notation is abstract, but not completely precise; e.g., authentication goals left implicit
- Infamously, crypto protocols are vulnerable to attack, without breaking underlying crypto algorithms
- New technologies force invention of new protocols
- Many formal analyses exist; Meadows' course reviews the general area
- **Our approach:** express goals by adding explicit events; check using a type and effect system; Walker's course covers related type theory

11

A Fundamental Abstraction

- 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 useful, informal abstraction for distributed systems
 - Ex: heap references in type-safe languages, GUIDs (128 bit identifiers), and encryption keys.
 - Non Ex: integers, pointers in C, or a path to a file.

12

Formalizing Pure Names

- A **nominal calculus** includes a set of pure names and allows the generation of fresh, unguessable names.
- Ex:
 - the π -calculus (Milner, Parrow, and Walker 1989)
 - the join calculus (Fournet and Gonthier 1996)
 - the spi calculus (Abadi and Gordon 1997)
 - the ambient calculus (Cardelli and Gordon 1998)
- Non Ex:
 - CSP (Hoare 1977), CCS (Milner 1980): channels named, but neither generated nor communicated

13

Outline of the Course

- I: The π -calculus (today)
 - programming with names (review)
 - verifying correspondences with types and effects
- II: The spi-calculus (tomorrow)
 - programming with cryptography
 - verifying equational specs with bisimulations
 - types and effects for cryptography

14

Part I: The π -Calculus

- In this part:
- Examples, syntax, and semantics of the untyped π -calculus
 - Use of Woo and Lam's correspondence assertions to specify authenticity properties
 - A dependent type system for type-checking correspondence assertions

15

Syntax and Semantics

The structure and interpretation of π -calculus processes

R. Milner, J. Parrow, and D. Walker invented the π -calculus

16

Basic Ideas

- The π -calculus is a parsimonious formalism intended to describe the essential semantics of concurrent systems.
- A running π -program is an assembly of concurrent processes, communicating on named channels.
- Applications: semantics, specifications, and verifications of concurrent programs and protocols; various implementations

17

Example in the π -Calculus

Client: start virtual printer v ; use it:

```
new(v): (out start(v) | out v(job))
```

Server: handles real printer; makes virtual printers.

```
new(p): (... p ... | repeat (inp start(x);  
                           repeat (inp x(y); out p(y)))
```

Driver code Make new virtual printer Virtual printer at x

All the data items are channel names.

All interactions are channel inputs or outputs.

18

Syntax of the π -Calculus

x, y, z	names
$P, Q, R ::=$	processes
$\text{out } x(y_1, \dots, y_n)$	output tuple on x
$\text{inp } x(z_1, \dots, z_n); P$	input tuple off x
$\text{new}(x); P$	new name in scope P
$P \mid Q$	composition
$\text{repeat } P$	replication
stop	inactivity

Names x, y, z are the only data
Processes P, Q, R are the only computations
Beware: non-standard syntax

Communication is All!

- Operational semantics is a transition relation
 $P \rightarrow Q$
meaning that P evolves to Q
- Every transition derives from an I/O interaction:
 $\text{out } x(y_1, \dots, y_n) \mid \text{inp } x(z_1, \dots, z_n); P \rightarrow P\{z_1 \leftarrow y_1, \dots, z_n \leftarrow y_n\}$
where $P\{z \leftarrow y\}$ is the outcome of substituting y for each free occurrence of z in P

20

Correspondence Assertions

Using the π -calculus to specify authenticity properties of protocols.
T. Woo and S. Lam invented correspondence assertions.
Joint work with A. Jeffrey

21

Ex 1: Synchronised Exchange

```
sender(msg)  $\triangleq$ 
new(ack);
out c (msg, ack) |
inp ack();
```

```
receiver  $\triangleq$ 
inp c (msg, ack);
out ack();
```

```
system  $\triangleq$  (new(msg); sender(msg)) | receiver
```

After receiving an acknowledgement on the private channel ack , the sender believes the receiver has obtained the message msg .

How can this be formalized?

22

Correspondence Assertions

To specify authenticity properties, Woo and Lam propose **correspondence assertions**

Let $e \leftrightarrow b$ mean that the count of e events never exceeds the count of b events

Ex: "server accepts m from c " \leftrightarrow " c sent m "

Ex: "sender gets m ack" \leftrightarrow "receiver sent m ack"

These assertions are simple safety properties

Rule out replays, impersonations, etc.

23

Adding Correspondences to π

Programmers may write **begin** L and **end** L annotations in our π -calculus

These annotations implicitly define correspondence assertions of the form:

end $L \leftrightarrow$ **begin** L

that is, every **end** L is "paid for" by a distinct, preceding **begin** L

no requirement that the **begin** and **end** events be properly bracketed

The programmer may think of these assertions as verified at runtime (like **assert** in C)

24

Adding Assertions

```

sender(msg) ≜
new(ack);
out c (msg,ack) |
inp ack();
end (msg)

```

"Sender gets m ack"

```

receiver ≜
inp c (msg,ack);
begin (msg);
out ack();

```

"Receiver sent m ack"

- This code makes the assertion:
 - $end(msg) \leftrightarrow begin(msg)$
 - that is, the count of "Sender gets m ack" never exceeds the count of "Receiver got m ack"

25

Ex 2: Hostname Lookup

- We consider n hosts named h_1, \dots, h_n
- Host h_i listens for pings on channel $ping_i$; it replies to each ping it receives
- A single name server maps from hostnames h_i to ping channels $ping_i$
- After receiving a ping reply, a client may conclude it has talked to the correct server
 - We formalize this as a correspondence assertion

26

The Name Server

```

NameServer(query, h_1, ..., h_n, ping_1, ..., ping_n) ≜
repeat
inp query(h, res);
if h=h_1 then out res(ping_1); else
...
if h=h_n then out res(ping_n);

```

Returns the ping channel $ping_i$ when sent the hostname h_i

27

Ping Server on each Host

There is a process $PingServer(h_i, ping_i)$ running on each host h_i

```

PingServer(h, ping) ≜
repeat
inp ping(ack);
begin("h pinged");
out ack();

```

"h pinged" ≜ h

Before sending each acknowledgment, it runs $begin("h, pinged")$; to indicate that it has been pinged

28

A Client Process

```

PingClient(h, query) ≜
new(res);
out query(h, res);
inp res(ping);
new(ack);
out ping(ack);
inp ack();
end("h pinged")

```

Get ping address ping for hostname h

Ping the server at ping

If we get an acknowledgement, we believe we've been in touch with h

29

The Whole Example

```

system ≜
NameServer(query, h_1, ..., h_n, ping_1, ..., ping_n) |
pingServer(h_1, ping_1) | ... | pingServer(h_n, ping_n) |
pingClient(h_i, query)

```

- The **begin** and **end** annotations implicitly define a correspondence assertion:
 - the count of "h_i pinged" by $PingClient(h_i, query)$ never exceeds the count of "h_i pinged" by $PingServer(h_i, ping_i)$
- Easily generalises to multiple clients

30

Safety and Robust Safety

The point of writing correspondence assertions is to catch programs that are not safe:

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

Moreover, later we want programs to be safe in the presence of any hostile opponent, modelled by an arbitrary program:

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

31

Summary

- Woo and Lam used correspondence assertions to specify authenticity properties of crypto protocols
- Correspondence assertions are not just applicable to crypto protocols
- We added these to the π -calculus by incorporating **begin** $L;P$ and **end** $L;P$, and illustrated by example

32

Break

33

Safety by Typing

A λ -calculus effect system for the π -

Typing, we can prove correspondence assertions

Joint work with A. Jeffrey

34

Origins of Type Theory

- Cambridge 1901: Russell uncovers a paradox in Frege's system of arithmetic
- Later, 1908, he proposes his Theory of Types to patch the bug



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



Whoops!

35

Motivation for Type Systems

- A type system allows dynamic invariants (e.g., upper bounds on the values assumed by a variable) to be checked before execution (at compile- or load-time)
- Historically, types arose in programming languages to help prevent accidental programming errors, e.g., $1,0 + \text{"Fred"}$
- Also, types can guarantee properties that prevent malicious errors:
 - Denning's information flow constraints (Volpano and Smith)
 - Memory safety for mobile code (Stamos, bytecode verifiers, proof carrying code)

36

A Type and Effect System

- **Idea:** statically infer judgments

Types for names $E \vdash P : [L_1, \dots, L_n]$ the effect of P

meaning that multiset $[L_1, \dots, L_n]$ is a bound on the tuples that P may **end** but not **begin**.

- Hence, if we can infer $P : []$, we know any **end** in P has at least one matching **begin**, and so P is safe.
- We warm up by describing an effect system for straight-line code.

37

Effects of **begin** and **end**

- The process **end** L performs an unmatched **end**-event:

$E \vdash \text{end } L : [L]$

- The process **begin** L;P matches a single **end**-event:

If $E \vdash P : [L_1, \dots, L_n]$
then $E \vdash \text{begin } L;P : [L_1, \dots, L_n] - [L]$

Multiset subtraction

- Ex: we can tell **begin**(x);**end**(x) is safe:

$\text{begin}(x); \text{end}(x) : [(x)] - [(x)] = []$

38

Effects of Parallel and Stop

- The effect of $P \mid P'$ is the multiset union of the effects of P and P':

If $E \vdash P : e$ and $E \vdash P' : e'$ then $E \vdash P \mid P' : e+e'$

- The effect of **stop** is the empty multiset:

$E \vdash \text{stop} : []$

- Ex: an unsafe process,

$(\text{begin}(x); \text{stop}) \mid \text{end}(x) : [(x)]$

39

Effect of Replication

- The effect of **repeat** P is the effect of P multiplied unboundedly.

- On the face of it, **repeat end**(x) would have an effect $[(x), (x), (x), \dots]$

- But an unbounded effect cannot ever be matched by **begin**(x), so is unsafe.

- Hence, we require the effect of a replicated process to be empty.

If $E \vdash P : []$ then $E \vdash \text{repeat } P : []$

40

Effect of Restriction

- Restriction does not change effect of its body

- Need to avoid names going out of scope

Consider **begin**(x); **new**(x:T); **end**(x).

Unsafe, as the two x's are in different scopes

Same as **begin**(x); **new**(x':T); **end**(x').

If the restricted name occurs in the effect, it can never be matched, so the restriction is unsafe.

- Hence, we adopt the rule:

If $E, x:T \vdash P : e$ and $x \notin \text{fn}(e)$
then $E \vdash \text{new}(x:T); P : e$

41

Effects of I/O (First Try)

- An output has no effect.

If $E \vdash x : \text{Ch}(T)$ and $E \vdash z : T$
then $E \vdash \text{out } x z : []$

- Like restriction, an input does not change the effect of its body, but we must avoid scope violations.

If $E \vdash x : \text{Ch}(T)$ and $E, y:T \vdash P : e$
and $y \notin \text{fn}(e)$ then $E \vdash \text{inp } x (y:T); P : e$

- Ex: **inp** x(z:T); **end** (x,z) is not well-typed

42

Beyond Straight-Line Code?

```
begin(x);
new(z);
out z () |
(inp z(): end(x))
```

Safe, but cannot
be given effect []

```
new(z);
(begin(x); out z ()) |
(inp z(): end(x))
```

- We can now type straight-line code
- But our system is rather incomplete.
- What about the interdependencies induced by I/O?

43

Adding Effects to Channels

- We annotate channel types with effects
- Ex: a nullary channel, with effect $[(x)]$
 $z : \text{Ch}()[(x)]$
- Intuition: the effect of a channel represents unmatched `end`-events unleashed by output
An input can mask the effect: `inp z(): end(x) : []`
But an output must incur the effect: `out z () : [(x)]`
Have: `(begin(x); out z()) | (inp z(); end(x)) : []`
Sound, because an input needs an output to fire

44

Dependent Effects

- Consider the nondeterministic process:

```
begin(x1); out z (x1) |
begin(x2); out z (x2) |
inp z(x); end(x)
```

- We cannot tell whether the channel's effect should be $[(x_1)]$ or $[(x_2)]$
- So we allow channel effects to depend on the actual names communicated
- In this example, $z : \text{Ch}(x:T)[(x)]$

45

Effect of Output (Again)

- An output unleashes the channel's effect, given the actual data output:
If $E \vdash x : \text{Ch}(y:T)e_x$ and $E \vdash z : T$
then $E \vdash \text{out } x(z) : e_x\{y \leftarrow z\}$
- Ex:
Given $z : \text{Ch}(x:T)[(x)]$, `out z (x1) : [(x1)]`
and so `begin(x1); out z (x1) : []`
and also `begin(x2); out z (x2) : []`.
- Generalizes to polyadic output.

46

Effect of Input (Again)

- An input hides the channel's effect:
If $E \vdash x : \text{Ch}(y:T)e_x$ and $E, y:T \vdash P : e$
and $y \notin \text{fn}(e - e_x)$ then $E \vdash \text{inp } x(y:T); P : e - e_x$
- Ex:
`inp z(x); end(x) : []` given $z : \text{Ch}(x:T)[(x)]$
- Hence,
`begin(x1); out z (x1) |`
`begin(x2); out z (x2) |`
`inp z(x); end(x)`
has the empty effect.

47

Typing Ex 1

```
Msg  $\triangleq$  Name
Ack(msg)  $\triangleq$  Ch()[(msg)]
Req  $\triangleq$  Ch(msg:Msg,ack:Ack(msg))[]
```

The channel `c`
has type `Req`

```
sender(msg:Msg)  $\triangleq$ 
new(ack:Ack(msg));
out c (msg,ack);
inp ack();
end (msg)
```

```
receiver  $\triangleq$ 
inp c (msg:Msg,ack:Ack(msg));
begin (msg);
out ack();
```

$c:\text{Req} \vdash (\text{new}(msg:\text{Msg});\text{sender}(msg:\text{Msg})) \mid \text{receiver} : []$

48

Typing Ex 2

```
Host  $\triangleq$  Name
Ack(h)  $\triangleq$  Ch()["h pinged"]
Ping(h)  $\triangleq$  Ch(ack:Ack(h))[]
Res(h)  $\triangleq$  Ch(ping:Ping(h))[]
Query  $\triangleq$  Ch(h:Host,res:Res(h))[]
```

```
NameServer(query,h1,...,hn,ping1,...,pingn) |
pingServer(h1,ping1) |... | pingServer(hn,pingn) |
pingClient(h1,query) : []
```

- All type-checks fine, apart from the conditional in the NameServer code...

49

A Problem

```
NameServer(q:Query,h1:Host,...,p1:Ping(h1),...)  $\triangleq$ 
repeat
inp q(h:Host, res:Res(h));
if h=h1 then out res(ping1); else
...
if h=hn then out res(pingn);
```

Whoops!
We have $res:Ch(ping:Ping(h))[]$
but $p_1:Ping(h_1)$ Type error!

Hmm, in the then branch
we know that $h=h_1$...

Effect of If (First Try)

- The obvious rule for if:

```
If  $E \vdash x : T$  and  $E \vdash y : T$   
and  $E \vdash P : e$  and  $E \vdash Q : e'$   
then  $E \vdash \text{if } x=y \text{ then } P \text{ else } Q : eve'$ 
```

- Ex: the following has effect $[(x),(x).(y)]$

```
if  $x=y$  then end(x) | end(x)  
else end(x) | end(y)
```

- This rule is sound, but incomplete in the sense it cannot type our server example

eve' is the
least effect
including e
and e'

51

Effect of If

- Instead of the basic rule, we adopt:

```
If  $E \vdash x : T$  and  $E \vdash y : T$   
and  $E\{x \leftarrow y\} \vdash P\{x \leftarrow y\} : e\{x \leftarrow y\}$  and  $E \vdash Q : e'$   
then  $E \vdash \text{if } x=y \text{ then } P \text{ else } Q : eve'$ 
```

- The operation $E\{x \leftarrow y\}$ deletes the definition of x , and turns all uses of x into y

- Ex: we can now type-check:
if $h=h_1$ then out res(ping₁); else ...

52

Safety by Typing

Theorem (Safety)

If $E \vdash P : []$ then P is safe.

- Hence, to prove an authenticity property expressed as a correspondence assertion, all one need do is construct a typing derivation.

53

Summary of the Effect System

We exploited several ideas:

- Woo and Lam's correspondence assertions
- A type and effect system
- Dependent types for channels
- Special rule for checking conditionals

54

Summary of Part I

A rather idiosyncratic view of the π -calculus, emphasising:

- Examples of concurrent programming
- Type systems for preventing errors, including security related errors:
 - Simple types prevent channel mismatches
 - Types with effects prove authenticity

A more traditional view emphasises bisimulation and semantics.

55

End of Part I

56

A Calculus for Cryptographic Protocols

Andy Gordon, *Microsoft Research*

Summer School on
Foundations of Internet Security

June 14-23, 2002, Duszynki Zdrój, Poland

57

Ex I: Replays and Nonces

```

Whigmaleerie Client
adg@microsoft.com
secret
Transfer €100 to Prof Pacholski
Submit
Creating proxy object...
Contacting service http://localhost/WhigWebService/Service1.asmx for nonce...got 25
plain=<00,00,00,04,00,00,00,19,54,00,72,00,61,00,6E,00,73,00,66,00,65,00,72,00,20,00,AC,20,31,00,30,00,30,00,20,00,74,00,6F,00,20,00,50,00,72,00,6F,00,66,00,20,00,50,00,61,00,63,00,68,00,6F,00,6C,00,73,00,6B,00,63,00> (70 bytes)
key=<E7,31,31,C7,ED,68,AE,AB,E3,D1,78,2D,D7,7C,0B,22> (16 bytes)
Sending service http://localhost/WhigWebService/Service1.asmx
principal=adg@microsoft.com
cipher=<6D,94,B1,6A,DE,51,3F,78,39,18,C2,23,3C,D3,9C,E4,02,C6,E9,39,8F,46,60,60,EE,C4,81,80,04,23,8E,19,52,1D,2B,65,3E,55,A7,9E,3D,8D,14,05,38,96,8A,8E,5B,0A,0B,3D,54,8B,F5,F3,87,87,4E,9E,36,65,68,4D,93,A9,FD,12,4A,A2,08,D5,8E,71,78,5E,06,E9,81,84> (80 bytes)...got "Accepted: Transfer €100 to Prof Pacholski with nonce=25"
    
```

Ex II: Impersonation and Ids

Event 1	a begins	"a proving presence to b"
Message 1	a → b:	a
Message 2	b → a:	nb
Message 3	a → b:	{b, nb} _{key}
Message 4	b → s:	b, {a, {b, nb} _{key} } _{key}
Message 5	s → b:	{a, nb} _{key}
Event 2	b ends	"a proving presence to b"

- Impersonation avoided by including extra identities
- Authentication goals made precise by including a correspondence assertion

59

Part II: The Spi Calculus

In this part:

- Spi calculus = π -calculus plus crypto
- Programming crypto protocols in spi
- Two styles of specification and verification
 - By equations and bisimulation
 - By correspondence assertions and typing

60

Basic Ideas of Spi

Expressing cryptography in a nominal calculus

Joint work with M. Abadi

61

Beginnings

Crypto protocols are communication protocols that use crypto to achieve security goals

The basic crypto algorithms (e.g., DES, RSA) may be vulnerable, e.g., if keys too short

But even assuming perfect building blocks, crypto protocols are notoriously error prone

Bugs turn up decades after invention

Plausible application of the π -calculus:

Encode protocols as processes

Analyse processes to find bugs, prove properties

62

Spi = π + Cryptography

The names of the π -calculus abstractly represent the random numbers of crypto protocols (keys, nonces, ...)

- Restriction models key or nonce generation

We can express some forms of encryption using processes in the π -calculus

- We tried various encodings

Instead, spi includes primitives for cryptography

63

Syntax of Spi Terms

$M, N ::=$	terms
x	name, variable
(M, N)	pair
$\{M\}_N$	ciphertext

Since the π -calculus can express pairing, only symmetric-key ciphers are new

We can include other crypto operations such as hashing or public-key ciphers

64

Syntax of Spi Processes

$P, Q, R ::=$	processes
$\text{out } M \ N$	output N on M
$\text{inp } M(x); P$	input x off M
$\text{new}(x); P$	new name
$P \mid Q$	composition
$\text{repeat } P$	replication
stop	inactivity
$\text{split } M \text{ is } (x, y); P$	pair splitting
$\text{decrypt } M \text{ is } \{x\}_N; P$	decryption
$\text{check } M \text{ is } N; P$	name equality

65

Operational Semantics

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

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

Decryption evolves according to the rule:

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

- Decryption requires having the key N
- Decryption with the wrong key gets stuck
- There is no other way to decrypt

66

Equations and Spi

Specifying and verifying crypto protocols using equations

Joint work with M. Abadi

67

Ex: A Simple Exchange

```
sys(msg,d) ≜
new(k);
```

```

A → (out net({msg}_k) |
      (inp net(u);
       decrypt u is {m}_k; out d(m);)) ← B
```

The process *sys* represents a protocol where:

- A sends *msg* to B encrypted under *k*, over the public channel *net*
- Then B outputs the decryption of its input on another channel *d*

The protocol will get stuck (safely) if anyone captures or replaces A's message

68

Specifying Security Properties

We are only interested in safety properties.

We use equations for simplicity.

For authenticity, we build a necessarily correct, "magical" implementation:

```
Sys'(msg,d) ≜ new(k); (out net({msg}_k) |
                      (inp net(u); decrypt u is {m}_k; out d(msg);))
```

Secrecy. For all $msg_L, msg_R,$
 $new(d):sys(msg_L,d) \approx$
 $new(d):sys(msg_R,d)$

Authenticity. For all $msg,$
 $sys(msg,d) \approx sys'(msg,d)$

69

Formality in Context...

Like other formalisms, spi abstracts protocols:

- e.g., ignoring key and message lengths

So an implementation may not enjoy all the security properties provable at the spi level.

- Similarly, for flaws found at the spi level

In security applications, as in others, formal methods need to be joined with engineering rules-of-thumb and commonsense!

70

Defining Equivalence

Two processes are equivalent if no environment (opponent) can distinguish them.

Technically, we use a testing equivalence $P \approx Q$ (R. Morris; R. de Nicola and M. Hennessy).

- 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 are equivalent iff they pass the same tests.

71

Testing Equivalence

- Allows equational reasoning
- Is implied by other equivalences
 - Bisimulation focuses on the process in isolation
 - We often prove testing equivalence via bisimulation
- Reveals curious properties of spi, such as the "perfect encryption equation"

```
new(k); out c {M}_k ≈ new(k); out c {M}'_k
```

The outcome of a test cannot depend on data encrypted under an unknown key

72

The Opponent

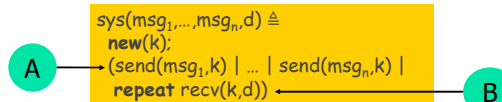
Our use of testing equivalence implicitly defines the opponent as an arbitrary spi program:

- it can try to create confusion through concurrent sessions,
- it can initiate sessions,
- it can replay messages,
- it can make up random numbers,
- but it cannot get too lucky, because of scoping.

Most approaches have more limited models.

73

Ex: Multiple Exchanges (Ex II)



Purpose: send multiset of messages from A to B:
The process sys represents a protocol where:

- There are n senders $send$,
- a replicated receiver $recv$ capable of receiving arbitrarily many messages,
- and both the senders and receivers share key k .

74

Secrecy Specified in Spi

Secrecy.
For all $(msg_{L1}, msg_{R1}), \dots, (msg_{Ln}, msg_{Rn}),$
 $new(d); sys(msg_{L1}, \dots, msg_{Ln}, d) \approx$
 $new(d); sys(msg_{R1}, \dots, msg_{Rn}, d)$

No observer (opponent) should be able to distinguish runs carrying different messages.

75

Authenticity Specified in Spi

Authenticity.
For all p_1, \dots, p_n , there is Q such that $fn(Q) \subseteq \{p_1, \dots, p_n, net\}$ and for all names $msg_1, \dots, msg_n:$
 $sys(msg_1, \dots, msg_n, d) \approx$
 $new(p_1, \dots, p_n);$
 $(Q \mid \mathit{inp}\ p_1(x).out\ d(msg_1) \mid \dots \mid \mathit{inp}\ p_n(x).out\ d(msg_n))$

By construction, the right-hand process:

- Only ever delivers the names msg_1, \dots, msg_n on d .
- Delivers msg no more times than it occurs in the multiset msg_1, \dots, msg_n .

By the equation, the same holds of $sys(msg_1, \dots, msg_n, d)$.

76

An Insecure Implementation

$send(msg, k) \triangleq out\ net(\{msg\}_k);$
 $recv(k, d) \triangleq \mathit{inp}\ net(u); \mathit{decrypt}\ u\ is\ \{msg\}_k; out\ d(msg)$

Satisfies neither secrecy nor authenticity.

Can you see why?

77

A Secure Implementation

Message 1	$b \rightarrow a:$	nb
Message 2	$a \rightarrow b:$	$\{ca, nb, msg\}_{kab}$

ca is a confounder, nb a nonce: random numbers;

ca is needed for secrecy, nb for authenticity

$send(msg, k) \triangleq$
 $\mathit{inp}\ net(u);$
 $new(ca);$
 $out\ net(\{ca, u, msg\}_k);$

$recv(k, d) \triangleq$
 $new(nb);$
 $out\ net(nb);$
 $\mathit{inp}\ net(u);$
 $\mathit{decrypt}\ u\ is\ \{ca, nb', msg\}_k;$
 $check\ nb' \ is\ nb;$
 $out\ d(msg)$

78

Ex: Wide Mouth Frog

The new channel is a fresh session key.
To prevent replays, we use nonce challenges.

79

A Crypto Implementation

Goal: authenticate a and kab to b

Message 1	a → s:	a
Message 2	s → a:	ns
Message 3	a → s:	a, {a,a,b,kab,ns} _{kas}
Message 4	s → b:	*
Message 5	b → s:	nb
Message 6	s → b:	{a,s,b,kab,nb} _{kab}
Message 7	a → b:	a, {msg} _{kab}

80

WMF Expressed in Spi

We consider n clients plus a server and m instances (sender, receiver, message):
 $I_i = (a_i, b_i, msg_i), \dots, I_m = (a_m, b_m, msg_m)$

```

sys(I1, ..., In, d) ≜
  new(k1s): ... new(kns):
  new(ks1): ... new(ksn):
  (send(I1) | ... | send(In) |
  repeat server |
  repeat recv(I1, d) | ... | repeat recv(In, d))
  
```

Allows opponent to interact with and initiate arbitrarily many concurrent sessions.

81

WMF Specified in Spi

Secrecy.
 If $a_{Lk} = a_{Rk}$ and $b_{Lk} = b_{Rk}$ for all $k \in 1..m$ then
 $new(d): sys(I_{L1}, \dots, I_{Ln}, d) \approx$
 $new(d): sys(I_{R1}, \dots, I_{Rn}, d)$

Authenticity.
 $sys(I_1, \dots, I_n, d) \approx sys'(I_1, \dots, I_n, d)$
 where $sys'(I_1, \dots, I_n, d)$ is a suitable "magical" specification, as before

Proved via a bisimulation relation defined by a rather complex and ad hoc invariant.

82

Lessons so far...

- The spi calculus is rather abstract
 - Can ignore details, especially details of encryption
- The spi calculus is rather accurate
 - Can describe exact conditions for sending messages
 - More precise than informal notations and some formal notations, e.g., the BAN logic
- Implicit opponent falls out of testing equivalence
- Direct proofs of equational specs can be very time consuming, though, can we do better?

83

Break

84

Typing and Spi

Type-checking correspondence assertions for crypto protocols

Joint work with A. Jeffrey

85

A New Protocol Analysis

First, code up the protocol in spi.

Second, specify protocol guarantees via Woo and Lam's correspondence assertions.

Third, figure out types for the keys, nonces, and data of the protocol.

Fourth, check that the program (including the assertions) is well-typed.

A type soundness theorem implies that the protocol guarantees are satisfied.

86

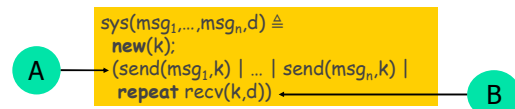
Step One

Code up the protocol as a spi calculus program

Exactly as before...

87

Ex: Multiple Exchanges



Purpose: send multiset of messages from A to B:

The process *sys* represents a protocol where:

- There are *n* senders *send*,
- a replicated receiver *recv* capable of receiving arbitrarily many messages,
- and both the senders and receivers share key *k*.

88

Step Two

Specify protocol guarantees via Woo and Lam's correspondence assertions

To do so, we enrich spi with **begin** *L* and **end** *L* assertions

89

Woo and Lam for Spi

Adapting Woo and Lam, we specify authenticity by annotating the system with **begin** and **end** events that ought to be in correspondence:

```
"Sender sent m" ≜ (m)  
send(msg, k) ≜  
begin "Sender sent msg": out net {(msg)k};  
recv(k, d) ≜  
inp net(u); decrypt u is {msg}k;  
end "Sender sent msg": out d(msg)
```

90

Authenticity Re-formulated

Authenticity.

The process $\text{sys}(msg_1, \dots, msg_n, d)$ is robustly safe, i.e., safe given any **begin** and **end** free opponent.

For the same reason it failed previously, the insecure implementation fails this spec based on correspondence assertions.

We can annotate the secure implementation similarly.

No consensus on formulations of authenticity.

Still, correspondences are simple and precise.

In Lowe's terminology "injective agreement"

See Focardi, Gorrieri & Martinelli for comparison

91

Interlude

A type system for crypto primitives

Abadi's "Secrecy by Typing in Security Protocols" is one inspiration

92

Typing Assertions in Spi?

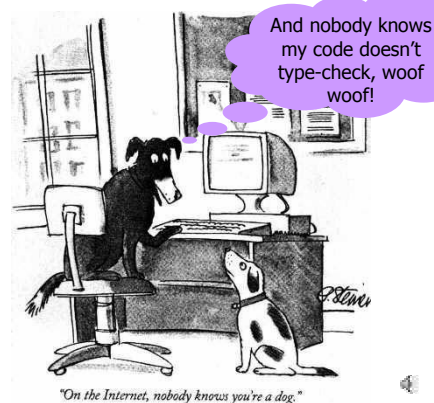
First, we introduce a typed spi calculus, whose rules can type I/O, data structures, and encryption.

Second, we extend with effects for tracking **end**-events.

A novel type for nonces transfers effects between senders and receivers.

In the end, the payoff is a guarantee of robust safety by typing.

93



The Untrusted Type Un

Terms of type Un represent untrusted channels and data structures read off the network

Rules include: if $M:\text{Un}$ and $N:\text{Un}$ then both $(M,N):\text{Un}$ and $\{M\}_N:\text{Un}$

For any untyped process O with free names x_1, \dots, x_n there is a typed process O' such that:

$x_1:\text{Un}, \dots, x_n:\text{Un} \vdash O'$ and $O = \text{erase}(O')$

So (due to Un) typed opponents (such as O') are as dangerous as untyped opponents.

95

Keys, Pairs

Terms of type $\text{Key } T$ 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 $x:T \vdash P$ well-typed, then so is $\text{decrypt } M \text{ as } \{x:T\}_N:P$

Terms of type $(x:T, U\{x\})$ are dependent records of type T and type $U\{x\}$.

96

What Do We Have So Far?

$Net \triangleq Un$
 $Msg \triangleq Un$
 $MyNonce \triangleq Un$
 $MyKey \triangleq Key (Msg, MyNonce)$

We've enough to confer types on all the data in the multiple message protocol.

Typing avoids:

- arity errors ($MyKey$ only encrypts pairs)
- key disclosure (cannot transmit $MyKey$ on net)
- confusing keys with other types, e.g., pairs

97

But Can We Check Assertions?

Let the judgment

$E \vdash P : [L_1, \dots, L_n]$

mean the multiset $[L_1, \dots, L_n]$ is a 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.

98

Transferring Effects?

In π , if a trusted channel $Ch T$ has effect e , we:
 allow an input to mask the effect e , but
 require an output to incur the effect e .

Transfer sound due to both 1-1 correspondence and the requirement on output.

But useless for crypto protocols, since messages between processes are:

- communicated on untrusted channels (e.g., $net:Un$)
- secured via trusted keys (e.g., $k:MyKey$)

99

Cannot Transfer via $Key T$

Suppose each key type $Key T$ has an effect e , and we:

- allow a decryption to mask the effect e , but
- require an encryption to incur the effect e .

Unsound. Although can enforce requirement on decryption, ciphertexts may be duplicated (replayed).

So cannot rely on 1-1 correspondence between encryption and decryption.

100

Transfer Effects via $Nonce e$

Nonces are published names, so created as Un .

Still, consider a type $Nonce e$, and we:

- allow checking a nonce to mask effect e , but
- require casting an Un name to $Nonce e$ to incur e .

Sound, because:

Typing constraints guarantee if a $Nonce e$ name exists then a **cast** has incurred the effect e

Linearity constraints on nonce checking ensure 1-1 correspondence (only check fresh name, once)

101

Typing a Nonce Handshake

1. Receiver publishes new $N:Un$
2. Sender receives $N:Un$, asserts **begin** L , casts nonce into $Nonce[L]$, returns within encrypted message
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 (justified by previous **cast**)

102

Semantics of **cast**

The process **cast** x to $(y:\mathbf{Nonce}\ e);P$ evolves into the process $P\{y\leftarrow x\}$

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 $E \vdash x : \mathbf{Un}$ and $E, y:\mathbf{Nonce}\ e \vdash P : e'$
then $E \vdash \mathbf{cast}\ x\ \mathbf{to}\ (y:\mathbf{Nonce}\ e);P : e+e'$

Only kind of cast in the system

103

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 $E \vdash x : \mathbf{Nonce}\ e$ and $E \vdash y : \mathbf{Un}$ and $E \vdash P : e'$ then
 $E \vdash \mathbf{check}\ x\ \mathbf{is}\ y; P : e'-e$

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

Enforced by adding a new kind of effect; details omitted

104

Step Three

Figure out types for the keys, nonces, and data of the protocol.

105

Ex: Multiple Messages Again

```
net : Un
Msg  $\triangleq$  Un
MyNonce(m)  $\triangleq$  Nonce ["Sender sent m"]
MyKey  $\triangleq$  Key (m:Msg, MyNonce(m))
```

```
send(msg:Msg,k:MyKey):[]  $\triangleq$ 
inp net(u:Un);
begin "Sender sent msg";
cast u to (no:MyNonce(msg));
out net ((msg,no),k);
```

106

Ex: Multiple Messages Cont.

```
recv(k:MyKey,d:Un):[]  $\triangleq$ 
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";
out d(msg);
```

(For clarity, we omit confounders.)

107

Step Four

Fourth, check that the program (including the assertions) is well-typed.

A type soundness theorem implies the protocol guarantees are satisfied.

108

Robust Safety by Typing

Theorem (Robust Safety)

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

In the multiple messages example, we can check by hand that the following is derivable:

$\text{net}, \text{msg}_1, \dots, \text{msg}_n, d: \text{Un} \vdash \text{sys}(\text{msg}_1, \dots, \text{msg}_n, d) : []$

Hence, authenticity is a corollary of the theorem.

109

Ex: A slight variant of WMF

Event 1	a begins	"a sending b key kab"
Message 1	$a \rightarrow s:$	a
Message 2	$s \rightarrow a:$	ns
Message 3	$a \rightarrow s:$	$a, \{\text{tag3}(b, \text{kab}, \text{ns})\}_{\text{kas}}$
Message 4	$s \rightarrow b:$	*
Message 5	$b \rightarrow s:$	nb
Message 6	$s \rightarrow b:$	$\{\text{tag6}(a, \text{kab}, \text{nb})\}_{\text{kbs}}$
Event 2	b ends	"a sending b key kab"
Message 7	$a \rightarrow b:$	$a, \{\text{msg}\}_{\text{kab}}$

110

Typing the WMF

$p_1, \dots, p_n, s: \text{Prin} \triangleq \text{Un}$ --n principals, one server
 $\text{SKey} \triangleq \text{Key } T$ --session keys, for some payload T
 $\text{kp}, s: \text{PrincipalKey}(p)$ --longterm key for each principal

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3}(p) + \text{Cipher6}(p))$
 $\text{Cipher3}(a) \triangleq (b: \text{Prin}, \text{kab}: \text{SKey}, \text{ns}: \text{Nonce}["a \text{ sending } b \text{ key } \text{kab}"])$
 $\text{Cipher6}(b) \triangleq (a: \text{Prin}, \text{kab}: \text{SKey}, \text{ns}: \text{Nonce}["a \text{ sending } b \text{ key } \text{kab}"])$

Slightly simplified compared to original.

Given these types, the system has empty effect (and names known to opponent have type Un). Hence, authenticity follows just by typing.

111

Ex: Woo and Lam

Cannot type the flawed original

Can type a version where the ciphertexts include principal identities

As discussed by Abadi and Needham (others?)

The typing implies one encryption is redundant

As suggested by Anderson and Needham

112

Typing 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	$a \rightarrow b:$	$\{\text{tag3}(b, \text{nb})\}_{\text{kas}}$
Message 4	$b \rightarrow s:$	$b, \{\text{tag4}(a, \{\text{tag3}(b, \text{nb})\}_{\text{kas}})\}_{\text{kbs}}$
Message 5	$s \rightarrow b:$	$\{\text{tag5}(a, \text{nb})\}_{\text{kbs}}$
Event 2	b ends	"a proving presence to b"

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3}(p) + \text{Cipher4}(p) + \text{Cipher5}(p))$
 $\text{Cipher3}(a) \triangleq (b: \text{Prin}, \text{nb}: \text{Nonce}["a \text{ proving presence to } b"])$
 $\text{Cipher4}(b) \triangleq (a: \text{Prin}, \text{cipher}: \text{Un})$ --seems redundant
 $\text{Cipher5}(b) \triangleq (a: \text{Prin}, \text{nb}: \text{Nonce}["a \text{ proving presence to } b"])$

113

Typing Woo and Lam, again

Event 1	a begins	"a proving presence to b"
Message 1	$a \rightarrow b:$	a
Message 2	$b \rightarrow a:$	nb
Message 3	$a \rightarrow b:$	$\{\text{tag3}(b, \text{nb})\}_{\text{kas}}$
Message 4	$b \rightarrow s:$	$a, \{\text{tag3}(b, \text{nb})\}_{\text{kas}}$
Message 5	$s \rightarrow b:$	$\{\text{tag5}(a, \text{nb})\}_{\text{kbs}}$
Event 2	b ends	"a proving presence to b"

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3}(p) + \text{Cipher5}(p))$
 $\text{Cipher3}(a) \triangleq (b: \text{Prin}, \text{nb}: \text{Nonce}["a \text{ proving presence to } b"])$
 $\text{Cipher5}(b) \triangleq (a: \text{Prin}, \text{nb}: \text{Nonce}["a \text{ proving presence to } b"])$

114

A Rule-of-Thumb

The Explicitness Principle: Robust security is about explicitness. A cryptographic protocol should make any necessary naming, typing and freshness information explicit in its messages; designers must also be explicit about their starting assumptions and goals, as well as any algorithm properties which could be used in an attack.

(from Anderson and Needham, *Programming Satan's Computer*, in LNCS 1000, 1995.)

Our type system shows promise as a formalisation of at least some of this principle.

115

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>

116

```

Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\adg>cd desktop\demo
C:\Documents and Settings\adg\Desktop\Demo>cryptyc 3-typed.cry
Cryptyc

3-typed.cry
Type error!
At line 40:
In server Receiver at B (socket : Public):
In end (A sent msg to C);
the effect (end (A sent msg to C)) is unjustified.
Effects can be justified either with a matching begin,
or with appropriate nonce checks.

C:\Documents and Settings\adg\Desktop\Demo>
    
```

Related Protocol Analyses

		Little human effort	Auto attack discovery	Intuitive explanation of passes	Precise semantics	Unbounded principal size
BAN	Belief logic					
FDR	Model checker					
Isabelle	Interactive thm prover					
Athena	Automatic thm prover					
Cryptyc	Type checker					

- Beware: some columns rather subjective
- Lots of work omitted; see technical report

118

Assessment

- Benefits
 - Familiar program/type-check/debug cycle
 - Little human effort per protocol
 - No bound on size of opponent or protocol
 - Naturally handles control flow
 - Types are meaningful documentation
- Limitations
 - Certain correct idioms do not fit our type discipline
 - No automatic discovery of attacks
 - No insider attacks
 - Usual Dolev-Yao perfect encryption assumptions

119

Related Work on Spi

Improved techniques for equational reasoning (Abadi and Gordon; Boreale, De Nicola, and Pugliesi; Abadi and Fournet)

Reachability analysis (Amadio; Abadi and Fiore)

Authentication schema (Focardi, Gorrieri, and Martinelli)

Type systems (Abadi and Blanchet; Gordon and Jeffrey)

Flow analyses (Bodei, Degano, Nielson, and Nielson)

Logic programming (Abadi and Blanchet; Blanchet)

120

Summary of Part II

- The spi calculus allows programming and specification of crypto protocols
- We borrow many ideas from the π -calculus
- We specify both secrecy and authenticity
- Testing equivalence crisply specifies secrecy
- Woo and Lam's correspondence assertions are good for authenticity
- Type-checking spi programs is a cost effective method for checking some authenticity properties

121

Ideas for Research...

- Alan and I are actively developing types for authenticity
 - Lots of issues remain such as key compromise, further transforms, timestamps, ...
- The MSR Vault project checks low level code using an effect system
 - Would be valuable to give a direct formalization of Vault's semantics

122

Overall...

- Showed specific applications of process algebraic techniques to security issues
- This is an exciting time...loads of security problems amenable to formal analyses
- In this area, there is a premium on finding attacks, but proving limited guarantees has great value too
- Don't forget practical perspective
 - Schneier *Secrets and Lies* (2000)
 - Anderson *Security Engineering* (2001)

123

End of course

124

Ex: Otway and Rees

- Cannot type the (correct) original
 - A "false positive" because we have no rules for the way in which nonces used
- Can type a more efficient version given by Abadi and Needham
- The typing implies a further simplification

125

Abadi and Needham's version

Message 1	$a \rightarrow b:$	a, b, na
Message 2	$b \rightarrow s:$	a, b, na, nb
Event 1	s begins	"initiator a key kab for b"
Event 2	s begins	"responder b key kab for a"
Message 3	$s \rightarrow b:$	$\{\text{tag3a}(a, b, kab, na)\}_{k_{ab}}, \{\text{tag3b}(a, b, kab, nb)\}_{k_{ba}}$
Event 3	b ends	"responder b key kab for a"
Message 4	$b \rightarrow a:$	$\{\text{tag3a}(a, b, kab, na)\}_{k_{ab}}$
Event 4	a ends	"initiator a key kab for b"

$\text{PrincipalKey}(p) \triangleq \text{Key}(\text{Cipher3a}(p) + \text{Cipher3b}(p))$

$\text{Cipher3a}(a) \triangleq (a', b': \text{Prin}, kab: \text{SKey}, na: \text{Nonce}["\text{initiator a key } kab \text{ for b}"])$

$\text{Cipher3b}(b) \triangleq (a, b': \text{Prin}, kab: \text{SKey}, nb: \text{Nonce}["\text{responder b key } kab \text{ for a}"])$

126

Pi Basics

127

Replication

- Replication $\text{repeat } P$ behaves like the parallel composition of unboundedly many replicas of P
- It obeys the rule:
 $\text{repeat } P \equiv P \mid \text{repeat } P$
- There are no reduction rules for $\text{repeat } P$
 - We cannot reduce within $\text{repeat } P$ but must first expand into the form $P \mid \text{repeat } P$
- Replication has a simple semantics, and can encode recursion and repetition

128

Parallel Composition

- Parallel composition is a binary operator:
 $P \mid Q$
- Processes P and Q may interact together, or with their environment, or on their own.
- It is associative and commutative:
 $P \mid Q \equiv Q \mid P$
 $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
- It obeys the reduction rule:
 $P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$

129

Stop

- An inactive process that does nothing
 stop
- Sometimes, it is garbage to be collected:
 $P \mid \text{stop} \equiv P$
 $\text{new}(x); \text{stop} \equiv \text{stop}$
 $\text{repeat } \text{stop} \equiv \text{stop}$
- It has no reduction rules

130

Restriction

- Restriction creates a new, unforgeable, unique channel name x with scope P
 $\text{new}(x); P$
- It may be re-arranged:
 $\text{new}(x); \text{new}(y); P \equiv \text{new}(y); \text{new}(x); P$
 $\text{new}(x); (P \mid Q) \equiv P \mid \text{new}(x); Q$ if x not free in P
- It obeys the reduction rule: **Scope extrusion**
 $P \rightarrow Q \Rightarrow \text{new}(x); P \rightarrow \text{new}(x); Q$

131

Channel Output

- Channel output represents a tuple (y_1, \dots, y_n) sent on a channel x
 $\text{out } x(y_1, \dots, y_n)$
- An abbreviation for asynchronous output:
 $\text{out } x(y_1, \dots, y_n); P \triangleq \text{out } x(y_1, \dots, y_n) \mid P$
Means: send a tuple asynchronously, then do P
- Some versions of the π -calculus feature a synchronous, blocking output as primitive.

132

Channel Input

- Channel input blocks awaiting a tuple (z_1, \dots, z_n) sent on a channel x , then does P

$\text{inp } x(z_1, \dots, z_n); P$

The names z_1, \dots, z_n have scope P

- Input and output reduce together:

$\text{out } x(y_1, \dots, y_n) \mid \text{inp } x(z_1, \dots, z_n); P \rightarrow P\{z_1 \leftarrow y_1, \dots, z_n \leftarrow y_n\}$

where $P\{z \leftarrow y\}$ is the outcome of substituting y for each free occurrence of z in P

133

The Semantics on One Page

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

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

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

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

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

134

Ex: Exchanging Global Names

- Consider a fragment of our example:
 $\text{out start}(v) \mid \text{out } v(\text{job}) \mid$
 $\text{inp start}(x); \text{inp } x(y); \text{out } p(y)$
- We may re-arrange the process:
 $\equiv \text{out } v(\text{job}) \mid$
 $\text{out start}(v) \mid \text{inp start}(x); \text{inp } x(y); \text{out } p(y)$
- Apply a reduction:
 $\rightarrow \text{out } v(\text{job}) \mid \text{inp } v(y); \text{out } p(y)$
- And again:
 $\rightarrow \text{out } p(\text{job})$

135

Ex: Exchanging Local Names

- Next, we freshly generate a private v :
 $\text{new}(v); (\text{out start}(v) \mid \text{out } v(\text{job})) \mid$
 $\text{inp start}(x); \text{inp } x(y); \text{out } p(y)$
- To allow reduction, we enlarge v 's scope:
 $\equiv \text{new}(v); (\text{out } v(\text{job}) \mid \text{out start}(v) \mid$
 $\text{inp start}(x); \text{inp } x(y); \text{out } p(y))$
- Apply reductions:
 $\rightarrow \text{new}(v); (\text{out } v(\text{job}) \mid \text{inp } v(y); \text{out } p(y))$
 $\rightarrow \text{new}(v); \text{out } p(\text{job})$
- And garbage collect:
 $\equiv \text{out } p(\text{job})$

136