Nominal Calculi for Security and Mobility

Andy Gordon, *Microsoft Research*

FOSAD, Bertinoro, September 2000

Parts I and II

Goals of the Course

- Explain the fundamentals of **nominal calculi** and programming with names
  - Nominal (6): **of names:** relating to or consisting of a name or names (Encarta World English Dictionary)
- Explain various specification techniques, including equations and correspondence assertions
- Explain various verification techniques, especially type-checking

Non-Goals of the Course

- Describe implementations

  They include: Pict, Jocaml, Funnel, XLANG

- Describe formalisms and proofs in full detail

  See papers

- Describe all applications of nominal calculi to security and mobility

  My selection reflects a personal bias!

Overall...

- I want to get you up to speed on recent advances on applying nominal calculi to security and mobility
- 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!

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

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 in COM, and encryption keys.
  - Non Ex: integers, pointers in C, or a path to a file.

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

Three Nominal Calculi

I: The π-calculus (today)

programming with names

II: The spi calculus (Thursday)

programming with cryptography

III: The ambient calculus (Friday)

programming with mobile containers

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
- A simple type system for guaranteeing locality and secrecy properties

Syntax and Semantics

The structure and interpretation
of π-calculus processes

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

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

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.

Driver code

Make new virtual printer

Virtual printer at x

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

All the data items are channel names.

All interactions are channel inputs or outputs.

Syntax of the π-Calculus

x,y,z	names
P,Q,R ::=	processes
out $x(y_1,\ldots,y_n)$	output tuple on x
inp $x(z_1,\ldots, z_n); P$	input tuple off x
new$(x); P$	new name in scope P
$P \mid Q$	composition
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

Semantics of the π-Calculus

- We define process behaviour in the “chemical” style (Berry and Boudol 1990).
- The semantics divides into a reduction relation $P \rightarrow P'$, describing the evolution of P into P', and an equivalence relation $P \equiv P'$
- Think of $\rightarrow$ as internal, nondeterministic computation, and $\equiv$ as re-arrangement

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:

  $P \to Q \Rightarrow P \mid R \to Q \mid R$

Replication

- Replication **repeat** P behaves like the parallel composition of unboundedly many replicas of P
- It obeys the rule:

  repeat P ≡ P | **repeat** P

- There are no reduction rules for **repeat** P
  - We cannot reduce within **repeat** P but must first expand into the form P | **repeat** P
- Replication has a simple semantics, and can encode recursion and repetition

Stop

- An inactive process that does nothing

  stop

- Sometimes, it is garbage to be collected:

  P | **stop** ≡ P

  new(x); **stop** ≡ **stop**

  repeat stop ≡ **stop**

- It has no reduction rules

Restriction

- Restriction creates a new, unforgeable, unique channel name x with scope P

$$\textbf{new}(x);\ P$$

- It may be re-arranged:

$$\textbf{new}(x);\ \textbf{new}(y);\ P \equiv \textbf{new}(y);\ \textbf{new}(x);\ P$$

$\textbf{new}(x);\ (P \mid Q) \equiv P \mid \textbf{new}(x);\ Q$ if x not free in P

Scope extrusion

- It obeys the reduction rule:

$$P \rightarrow Q \Rightarrow \textbf{new}(x);\ P \rightarrow \textbf{new}(x);\ Q$$

Channel Output

- Channel output represents a tuple $(y_1,\ldots,y_n)$ sent on a channel x

  $\mathbf{out}\ x(y_1,\ldots,y_n)$

- An abbreviation for asynchronous output:

  $\mathbf{out}\ x(y_1,\ldots,y_n);\ P \triangleq \mathbf{out}\ x(y_1,\ldots,y_n) \mid P$

  Means: send a tuple asynchronously, then do P

  Some versions of the π-calculus feature a synchronous, blocking output as primitive.

Channel Input

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

  inp $x(z_1,\ldots,z_n); P$

  The names $z_1,\ldots,z_n$ have scope P

- Input and output reduce together:

  out $x(y_1,\ldots,y_n)$ | **inp** $x(z_1,\ldots,z_n); P \rightarrow P\{z_1\leftarrow y_1,\ldots,z_n\leftarrow y_n\}$

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

The Semantics on One Page

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

$P \rightarrow Q \Rightarrow \mathbf{new}(x);P \rightarrow \mathbf{new}(x);Q$

$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$

$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \ \Rightarrow P' \rightarrow Q'$

$P \mid \mathbf{stop} \equiv P$

$P \mid Q \equiv Q \mid P$

$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

$\mathbf{repeat}\ \mathbf{stop} \equiv \mathbf{stop}$

$\mathbf{repeat}\ P \equiv P \mid \mathbf{repeat}\ P$

$\mathbf{new}(x);\mathbf{stop} \equiv \mathbf{stop}$

$\mathbf{new}(x);\mathbf{new}(y);P \equiv \mathbf{new}(y);\mathbf{new}(x);P$

$\mathbf{new}(x);(P \mid Q) \equiv P \mid \mathbf{new}(x);Q$ if $x \notin fn(P)$

$P \equiv P$

$P \equiv Q \Rightarrow Q \equiv P$

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

$P \equiv Q \Rightarrow \mathbf{new}(x);P \equiv \mathbf{new}(x);Q$

$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$

$P \equiv Q \Rightarrow \mathbf{repeat}\ P \equiv \mathbf{repeat}\ Q$

$P \equiv Q \Rightarrow \mathbf{inp}\ x(z_1,\ldots,z_n);P \equiv \mathbf{inp}\ x(z_1,\ldots,z_n);Q$

Ex: Exchanging Global Names

- Consider a fragment of our example:
  out start(v) | **out** v(job) |
  inp start(x); **inp** x(y); **out** p(y)
- We may re-arrange the process:
  ≡ **out** v(job) |
  out start(v) | **inp** start(x); **inp** x(y); **out** p(y)
- Apply a reduction:
  → **out** v(job) | **inp** v(y); **out** p(y)
- And again:
  → **out** p(job)

Ex: Exchanging Local Names

- Next, we freshly generate a private v:

  new(v);(**out** start(v) | **out** v(job)) |
  inp start(x); **inp** x(y); **out** p(y)

- To allow reduction, we enlarge v's scope:

  ≡ **new**(v);(**out** v(job) | **out** start(v) |
  inp start(x); **inp** x(y); **out** p(y))

- Apply reductions:

  → **new**(v);(**out** v(job) | **inp** v(y); **out** p(y))
  → **new**(v); **out** p(job)

- And garbage collect:

  ≡ **out** p(job)

Exercises

1. Derive: if $x \notin fn(P)$ then $\mathbf{new}(x);P \equiv P$.
2. Find a derivation showing that the full printer example can reduce to a state where job has been sent on p.
   - Need to manipulate restrictions and replications
   - See notes for a solution (but don’t cheat!)

Lessons so far…

- The π-calculus is a basic model of computation based on interaction between concurrent processes
- Its semantics consists of relations $P \to P'$ (evolution) and $P \equiv P'$ (re-arrangement)
- The operator **new**(x);P tracks the mobile scope of dynamically created names

  Hence, we can tell who can know and who cannot know a particular name

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

Ex 1: Synchronised Exchange

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

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

```
system ≜  (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?

Correspondence Assertions

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

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

- Ex: “dispense coffee” $\hookrightarrow$ “insert coin”
- Ex: “A gets receipt for m” $\hookrightarrow$ “B gets m”

These assertions are simple safety properties

- Rule out replays, confused identities, etc.

Adding Correspondences to π

Programmers may write **begin**$(x_1,\ldots,x_n)$ and **end**$(x_1,\ldots,x_n)$ annotations in our π-calculus

These annotations implicitly define correspondence assertions of the form:

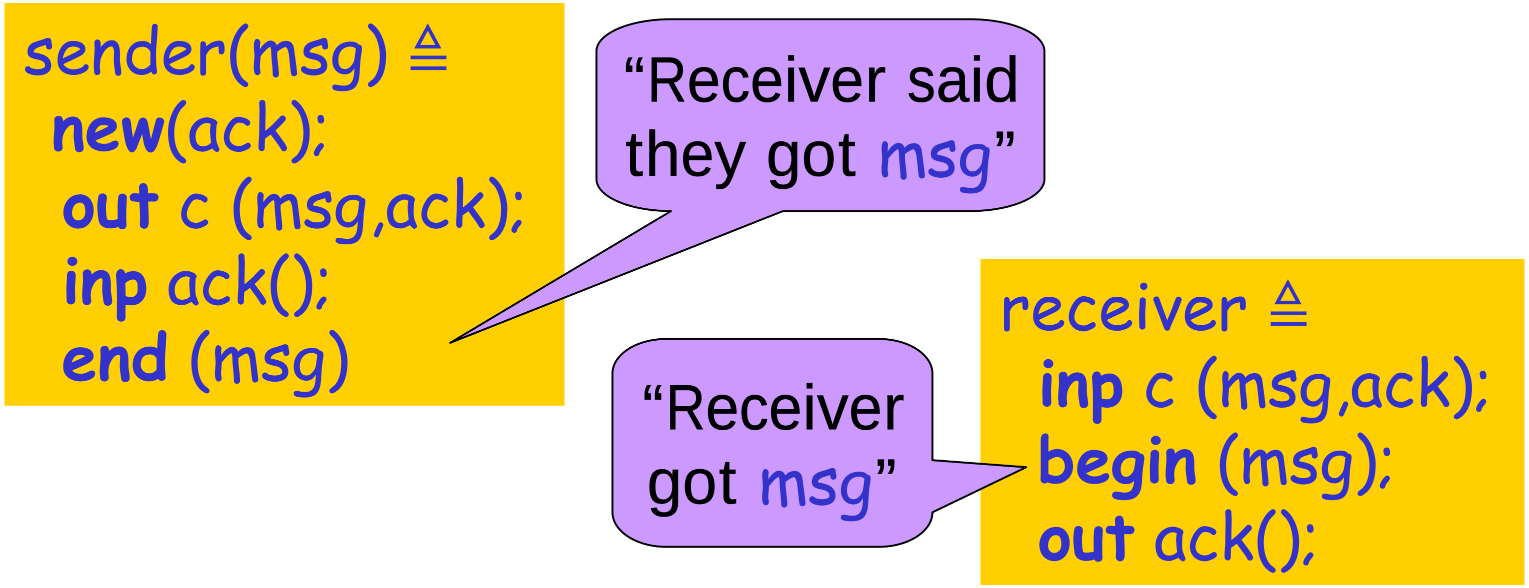
end$(x_1,\ldots, x_n) \hookrightarrow$ **begin**$(x_1,\ldots, x_n)$

that is, the count of **end**$(x_1,\ldots, x_n)$'s never exceeds the count of **begin**$(x_1,\ldots, x_n)$'s

no requirement that the begin and end events be properly bracketed

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

Adding Assertions



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

“Receiver said they got *msg*”

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

“Receiver got *msg*”

- This code makes the assertion:
  - $\mathbf{end}(msg) \hookrightarrow \mathbf{begin}(msg)$
  - that is, the count of “Receiver said they got *msg*” never exceeds the count of “Receiver got *msg*”

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

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

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_i pinged”); to indicate that it has been pinged

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

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_j,query)

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

A Semantics for Assertions

- Our existing semantics $P \to Q$ is very simple and elegant, but has no notion of "event" or "event history", just "internal evolution"
- Instead, we define two new relations:

$P \xrightarrow{\alpha} Q$ means P may evolve in one step α into Q

  - We call α an **event**
  - Events include $\mathbf{begin}(x_1,\ldots,x_n)$ and $\mathbf{end}(x_1,\ldots,x_n)$

$P \xrightarrow{t} Q$ means P may evolve in many steps $t=\alpha_1\ldots\alpha_n$ into Q (we call t a **trace**)

Events and Traces

α,β ::=	events
begin($x_1,\ldots,x_n$)	beginning $x_1,\ldots,x_n$
end($x_1,\ldots,x_n$)	ending $x_1,\ldots,x_n$
gen(x)	generate new x
τ	internal step

s,t ::=	trace
$\alpha_1 \ldots \alpha_n$	finite event sequence

A Trace of Ex 1

(**new**(m);Sender(m)) | Receiver

−**gen**(m)→ Sender(m) | Receiver

−**gen**(ack)→ (**out** c (m,ack); **inp** ack(); **end** (m)) | Receiver

−τ→ (**inp** ack(); **end** (m)) | (**begin** (m); **out** ack())

−**begin**(m)→ (**inp** ack(); **end** (m)) | **out** ack())

−τ→ **end** (m)

−**end**(m)→ **stop**

Safety

- Let a trace t be a **correspondence** iff $\mathrm{ends}(t) \leq \mathrm{begins}(t)$.

Multisets of **begin** and end **events** in t

Ex: $t_1 = \mathbf{begin}(x), \mathbf{begin}(y), \mathbf{end}(x), \mathbf{end}(y)$

Ex: $t_2 = \mathbf{end}(x), \mathbf{end}(y), \mathbf{begin}(x), \mathbf{begin}(y)$

- A process P is **safe** iff for all t, Q, if $P \xrightarrow{t} Q$ then t is a correspondence.
  - Requires all “intermediate” traces to be correspondences.

Robust Safety

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

- Though safe, our example is not robustly safe

  Let P ≜ (**new**(m);Sender(m)) | Receiver

  Take O ≜ **inp** c (m,ack); **out** ack() and P|O exhibits the trace: **gen**(m),**gen**(ack),τ,τ,**end**(m)

- To achieve robust safety, the channel c must be private, as in **new**(c);P

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**(x);P and **end**(x), and illustrated by example

Safety by Typing

A type and effect system for the π-calculus

By typing, we can prove correspondence assertions

Joint work with A. Jeffrey

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+"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)

A Type and Effect System

- **Idea**: statically infer judgments

Types for names

$$E \vdash P : [xs_1, \ldots, xs_n]$$

the **effect** of P

meaning that multiset $[xs_1, \ldots, xs_n]$ 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.

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

- The process $\mathbf{end}(x_1,\ldots,x_n)$ performs an unmatched **end**-event:

$$E \vdash \mathbf{end}(x_1,\ldots,x_n) : [(x_1,\ldots,x_n)]$$

- The process $\mathbf{begin}(x_1,\ldots,x_n);P$ matches a single **end**-event:

If $E \vdash P : [xs_1, \ldots, xs_n]$
then $E \vdash \mathbf{begin}(x_1,\ldots,x_n);P : [xs_1, \ldots, xs_n] - [(x_1,\ldots,x_n)]$

Multiset subtraction

- Ex: we can tell $\mathbf{begin}(x);\mathbf{end}(x)$ is safe:

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

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:

  $E \vdash$ **stop** $: []$

- Ex: an unsafe process,

  (**begin**(x);**stop**) | **end**(x)) : $[(x)]$

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),…]
- 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 $E \vdash \textbf{repeat}\ P : []$

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 $x \notin fn(e)$
  then $E \vdash \mathbf{new}(x{:}T);P : e$

Effects of I/O (First Try)

- An output has no effect.

  If $E \vdash x : \mathbf{Ch}(T_1,\dots,T_n)$ and $E \vdash y_i : T_i$ for $i \in 1..n$
  then $E \vdash \mathbf{out}\ x\ (y_1,\dots,y_n) : []$

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

  If $E \vdash x : \mathbf{Ch}(T_1,\dots,T_n)$ and $E, z_1{:}T_1,\dots,z_n{:}T_n \vdash P : e$
  and no $z_i \in e$ then $E \vdash \mathbf{inp}\ x\ (z_1{:}T_1,\dots,z_n{:}T_n);P : e$

- Ex: $\mathbf{inp}\ x(z{:}T);\ \mathbf{end}\ (x,z)$ is not well-typed

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?

Adding Effects to Channels

- We annotate channel types with effects
- Ex: a nullary channel, with effect [(x)]

  z : **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

Dependent Effects

- Consider the nondeterministic process:

  begin(x_1); **out** z (x_1) |
  begin(x_2); **out** z (x_2) |
  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 : **Ch**$(x{:}T)[(x)]$

Effect of Output (Again)

- An output unleashes the channel's effect, given the actual data output:

  If $E \vdash x : \mathbf{Ch}(z{:}T)e_x$ and $E \vdash y : T$
  then $E \vdash \mathbf{out}\ x(y) : e_x\{z \leftarrow y\}$

- Ex:

  Given $z : \mathbf{Ch}(x{:}T)[(x)]$, $\mathbf{out}\ z\ (x_1) : [(x_1)]$
  and so $\mathbf{begin}(x_1);\ \mathbf{out}\ z\ (x_1) : []$
  and also $\mathbf{begin}(x_2);\ \mathbf{out}\ z\ (x_2) : []$.

- Generalizes to polyadic output.

Effect of Input (Again)

- An input hides the channel's effect:

  If $E \vdash x : \mathbf{Ch}(z{:}T)e_x$ and $E,z{:}T \vdash P : e$
  and $z \notin e - e_x$ then $E \vdash \mathbf{inp}\ x(z{:}T);P : e - e_x$
- Ex:

  $\mathbf{inp}\ z(x);\ \mathbf{end}(x) : []$ given $z : \mathbf{Ch}(x{:}T)[(x)]$
- Hence,

  $\mathbf{begin}(x_1);\ \mathbf{out}\ z\ (x_1)\ |$
  $\mathbf{begin}(x_2);\ \mathbf{out}\ z\ (x_2)\ |$
  $\mathbf{inp}\ z(x);\ \mathbf{end}(x)$

  has the empty effect.

Typing Ex 1

```
Msg ≜ Ch()[]
Ack(msg) ≜ Ch()[(msg)]
Req ≜ Ch(msg:Msg,ack:Ack(msg))[]
```

The channel c has type Req

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

```
receiver ≜
  inp c (msg:Msg,ack:Ack(msg));
  begin (msg);
  out ack();
```

```
c:Req ⊢ (new(msg:Msg)sender(msg:Msg)) | receiver : []
```

Typing Ex 2

Host ≜ **Ch**()[]
Ack(h) ≜ **Ch**()[“h pinged”]
Ping(h) ≜ **Ch**(ack:Ack(h))[]
Res(h) ≜ **Ch**(ping:Ping(h))[]
Query ≜ **Ch**(h:Host,res:Res(h))[]

NameServer(query,h_1,…,h_n,$ping_1$,…,$ping_n$) |
pingServer (h_1,$ping_1$) |… | pingServer (h_n,$ping_n$) |
pingClient(h_j,query) : []

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

A Problem

```
NameServer(q:Query,$h_1$:Host,…,$p_1$:Ping($h_1$),…) ≜
    repeat
        inp q(h:Host, res:Res(h));
        if h=$h_1$ then out res($ping_1$); else
        …
        if h=$h_n$ then        res($ping_n$);
```

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 $E \vdash$ **if** $x{=}y$ **then** P **else** $Q : e \vee e'$

$e \vee e'$ is the least effect including e and e'

- 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

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 $E \vdash$ **if** $x=y$ **then** P **else** $Q : e \vee e'$

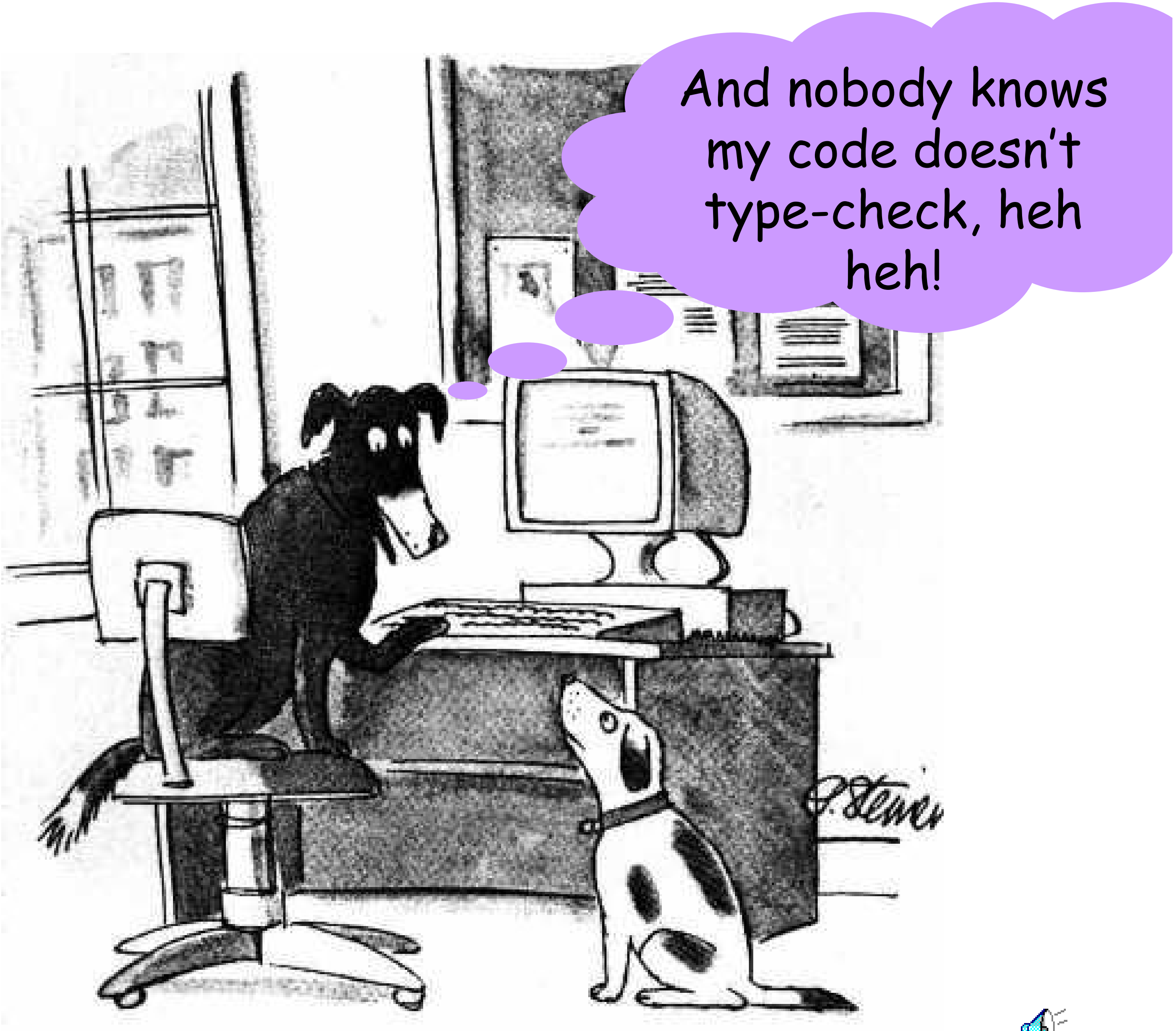
- The operation $E\{x \leftarrow y\}$ deletes the definition of x, and turns all uses of x into y
- Ex: we can now type-check:

  if $h=h_1$ **then out** $res(ping_1)$; **else** …

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.



"On the Internet, nobody knows you're a dog."

Typing the Opponent

- Want to prove *robust* safety, that P|O is safe for any (untyped) **end**-free opponent O.
- We might, somehow, prove something about the type erasure of P, but this gets messy.
- Instead, we adopt an old trick: a universal type **Un** for “typing” essentially untyped data.
- Hence, we represent the opponent as a typed process whose variables are of type **Un**.

Rules for Type **Un**

(Proc **Un** Input)

$$\frac{E \vdash x : \mathbf{Un} \qquad E, z_1{:}\mathbf{Un},\ldots,z_n{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{inp}\ x(z_1{:}\mathbf{Un},\ldots,z_n{:}\mathbf{Un}).P}$$

(Proc **Un** Output)

$$\frac{E \vdash x : \mathbf{Un} \qquad E \vdash y_1 : \mathbf{Un} \qquad E \vdash y_n : \mathbf{Un}}{E \vdash \mathbf{out}\ x(y_1,\ldots,y_n)}$$

- Untrusted data of type **Un** is unregulated, except it cannot be confused with trusted data of type $\mathbf{Ch}(T_1,\ldots,T_n)e$.

Robust Safety by Typing

Theorem (Robust Safety)
If $x_1{:}\mathsf{Un},\dots,x_n{:}\mathsf{Un} \vdash P : []$ then P is robustly safe.

- Simply by constructing a type derivation, we proved our two examples are robustly safe.
- A reasonable limitation is that names shared with opponent have types $x_1{:}\mathsf{Un},\dots,x_n{:}\mathsf{Un}$.
- Typing is considerably simpler than direct proof, and can prove infinite state properties.

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
- The **Un** type for type-checking opponents

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
  - Groups guarantee privacy (omitted)
  - Types with effects prove authenticity
  - The **Un** device for "typing" opponents

See elsewhere for a more traditional view, emphasising bisimulation and semantics.

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

Basic Ideas of Spi

Expressing cryptography in a nominal calculus

Joint work with M. Abadi

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

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

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

Syntax of Spi Processes

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

Operational Semantics

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

“if M is $\{x\}_N$ for some x, run P”

Decryption evolves according to the rule:

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

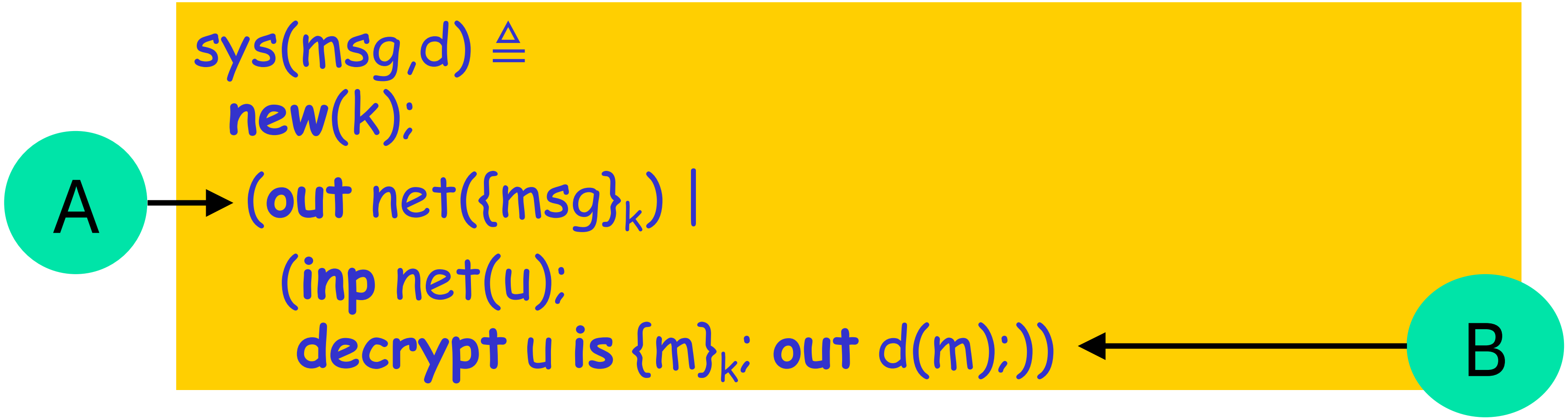
- Decryption requires having the key N
- Decryption with the wrong key gets stuck
- There is no other way to decrypt

Equations and Spi

Specifying and verifying crypto protocols using equations

Joint work with M. Abadi

Ex: A Simple Exchange



sys(msg,d) ≜
  new(k);
  (**out** net($\{msg\}_k$) |
    (**inp** net(u);
     decrypt u **is** $\{m\}_k$; **out** d(m);))

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

Specifying Security Properties

We are only interested in safety properties.

We use equations for simplicity.

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

$Sys'(msg,d) \triangleq \mathbf{new}(k);\ (\mathbf{out}\ net(\{msg\}_k)\ |$
$\quad (\mathbf{inp}\ net(u);\ \mathbf{decrypt}\ u\ \mathbf{is}\ \{m\}_k;\ \mathbf{out}\ d(msg);))$

Secrecy. For all msg_L, msg_R,
$\mathbf{new}(d);sys(msg_L,d) \simeq$
$\mathbf{new}(d);sys(msg_R,d)$

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

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!

Defining Equivalence

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

Technically, we use a testing equivalence $P \simeq Q$ (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.

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"
  - $\textbf{new}(k);\ \textbf{out}\ c\ \{M\}_k \simeq \textbf{new}(k);\ \textbf{out}\ c\ \{M'\}_k$
  - The outcome of a test cannot depend on data encrypted under an unknown key

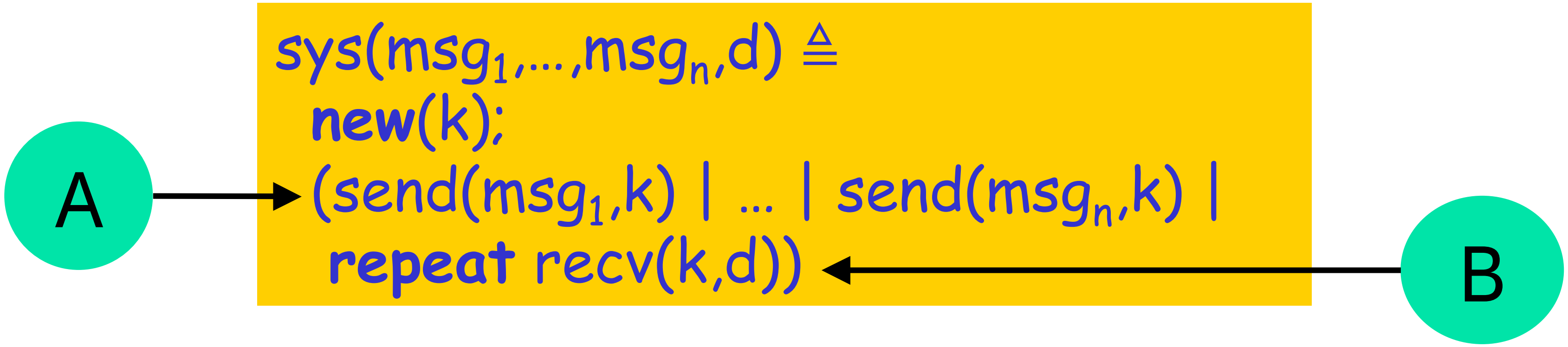
The Opponent

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.

Ex: Multiple Exchanges



```
$sys(msg_1,…,msg_n,d) \triangleq$
  new(k);
  (send($msg_1$,k) | … | send($msg_n$,k) |
   repeat recv(k,d))
```

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.

Secrecy Specified in Spi

Secrecy.

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

$\textbf{new}(d);sys(msg_{L1},\ldots,msg_{Ln},d) \simeq$

$\textbf{new}(d);sys(msg_{R1},\ldots,msg_{Rn},d)$

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

Authenticity Specified in Spi

Authenticity.
For all $p_1, \ldots, p_n$, there is Q such that $fn(Q) \subseteq \{p_1,\ldots,p_n,net\}$ and for all names $msg_1, \ldots, msg_n$:

$$\begin{aligned}&sys(msg_1,\ldots,msg_n,d) \simeq\\ &\quad \mathbf{new}(p_1,\ldots,p_n);\\ &\quad\; (Q \mid \mathbf{inp}\ p_1(x).\mathbf{out}\ d(msg_1) \mid \ldots \mid \mathbf{inp}\ p_n(x).\mathbf{out}\ d(msg_n))\end{aligned}$$

By construction, the right-hand process:

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

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

An Insecure Implementation

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

Satisfies neither secrecy nor authenticity.

Can you see why?

A Secure Implementation

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

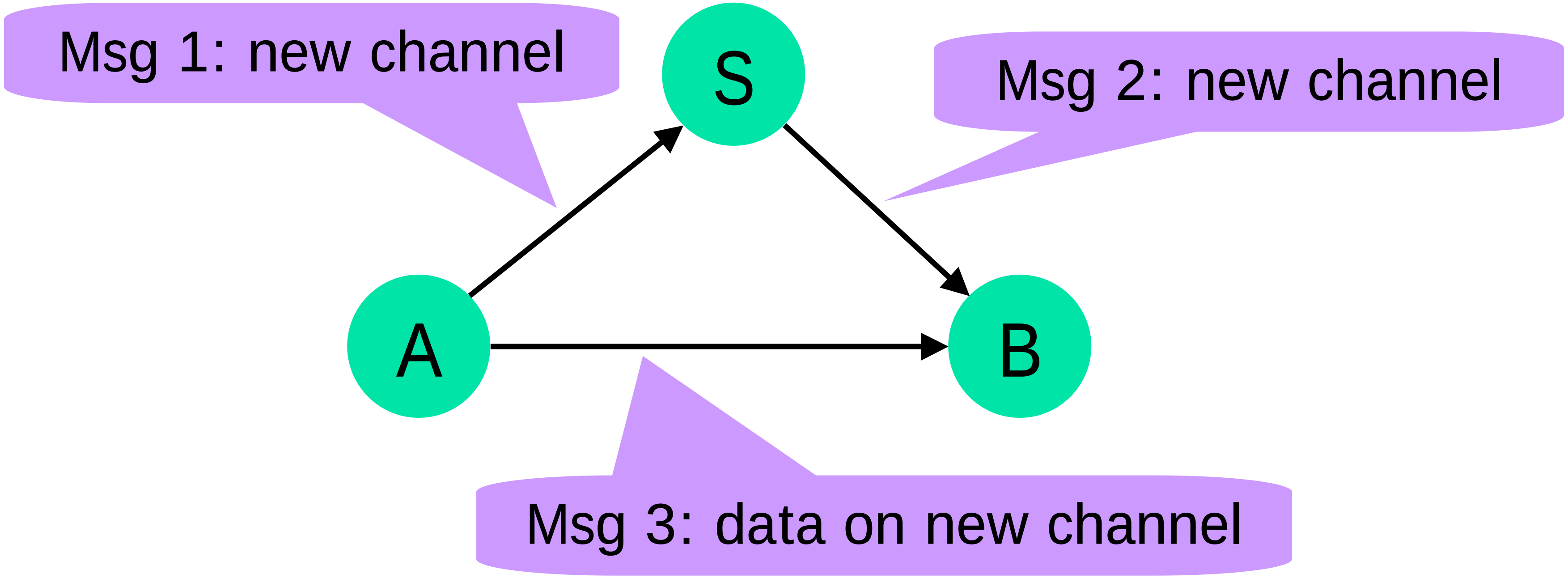
ca is a confounder, nb a nonce: random numbers;

ca is needed for secrecy, nb for authenticity

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

```
recv(k,d) ≜
  new(nb);
  out net(nb);
  inp net(u);
  decrypt u is $\{co,nb',msg\}_k$;
  check nb' is nb;
  out d(msg)
```

Ex: Wide Mouth Frog


Msg 1: new channel

Msg 2: new channel

Msg 3: data on new channel

The new channel is a fresh session key.

To prevent replays, we use nonce challenges.

A Crypto Implementation

Goal: authenticate a and kab to b

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

WMF Expressed in Spi

We consider n clients plus a server and m instances (sender, receiver, message):

$I_1=(a_1,b_1,msg_1)$, …, $I_m=(a_m,b_m,msg_m)$

```
$sys(I_1,\ldots, I_n,d) \triangleq$
  new(k1S); … new(knS);
  new(kS1); … new(kSn);
  (send($I_1$) | … | send($I_n$) |
    repeat server |
    repeat recv(1,d) | … | repeat recv(n,d))
```

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

WMF Specified in Spi

Secrecy.

If $a_{Lk}=a_{Rk}$ and $b_{Lk}=b_{Rk}$ for all $k\in 1..m$ then

$\textbf{new}(d); sys(I_{L1},\ldots, I_{Ln},d) \simeq$

$\textbf{new}(d); sys(I_{R1},\ldots, I_{Rn},d)$

Authenticity.

$sys(I_1,\ldots, I_n,d) \simeq sys'(I_1,\ldots, I_n,d)$

where $sys'(I_1,\ldots, I_n,d)$ is a suitable "magical" specification, as before

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

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., BAN

Implicit opponent falls out of testing equivalence

Direct proofs of equational specs can be very time consuming, though, can we do better?

Some Spi Developments...

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; Gordon and Jeffrey)

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

Interlude: The Budget Calculus

Understanding correspondences as financial prudence

Based on a convivial conversation with Josh and Moti

The Budget Calculus

An analogy between processes and financial plans may help explain our effect system:

- **begin**'s are like earnings; **end**'s are like spending.
- The effect of a process is like a spending budget.
  - A budget is a bound on how much you plan to spend beyond what you earn or receive yourself.
- An effect transfer is like a gift between different departments in the same organisation.
  - If you receive a gift, you can spend it, assuming someone else has already earned it.

The Widget Department

```
WidgetDept ≜
  workHard; paySalary;
  if feelingGenerous then payR&D else payBonus
workHard ≜ earn €1000
paySalary ≜ spend €500
payR&D ≜ issue memo
payBonus ≜ spend €500
```

According to its plan, this dept starts with nothing, earns some money, and then spends it, depending on how things turn out.

The Research Department

```
ResearchDept ≜
  thinkDeepThoughts; hopeForBertinoro
thinkDeepThoughts ≜ …          --costs 0, earns 0
hopeForBertinoro ≜ await memo; goToBertinoro
goToBertinoro ≜ spend €500
```

This department starts with nothing, breaks even by default, but spends some money if it has a friendly sponsor in another department.

Effects are like Budgets

Budgets are upper bounds on spending:

spend €1000 : €1000
earn €1000; **spend** €500 : €0
if … **then spend** €100 **else spend** €200 : €200

Budget memos allow gifts:

memo: **Memo** €100
earn €100; **issue** memo : €0
await memo; **spend** €100 : €0

Breaking Even like Safety

WidgetDept | ResearchDept : €0

Showing a plan has a €0 budget implies it will at least break even.

Overall, all its spending is justified by earnings, though there may be surplus earnings.

Analogously, showing a process has an [] effect implies it is safe.

Overall, all its end's are justified by earlier begin's, though there may be surplus begin's.

Typing and Spi

Type-checking correspondence assertions for crypto protocols

Joint work with A. Jeffrey

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)

Authenticity Re-formulated

Authenticity.
The process $sys(msg_1,\ldots,msg_n,d)$ 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.

If we could check robust safety by typing, we'd have a cost effective verification method...

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, as in the π-calculus.

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

In the end, the payoff is a guarantee of robust safety by typing, as in the π-calculus.

The Untrusted Type **Un**

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

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

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

x_1:**Un**, ..., x_n:**Un** $\vdash O'$ and $O = erase(O')$

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

The Channel Type **Ch** T

Terms of type **Ch** T are names used as channels for communicating type T

- If M:**Ch** T and N:T then **out** M N well-typed
- If M:**Ch** T and x:T ⊢ P well-typed, then so is **inp** M(x:T);P

The Key Type $\mathbf{Key}\ T$

Terms of type $\mathbf{Key}\ T$ are names used as symmetric keys for encrypting type T

If $M{:}T$ and $N{:}\mathbf{Key}\ T$ then $\{M\}_N{:}\mathbf{Un}$.

If $M{:}\mathbf{Un}$ and $N{:}\mathbf{Key}\ T$ and $x{:}T \vdash P$ well-typed, then so is $\mathbf{decrypt}\ M\ \mathbf{as}\ \{x{:}T\}_{N};P$

The Pair Type $(x{:}T,\ U)$

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

- If $M{:}T$ and $N{:}U\{x \leftarrow M\}$ then $(M,N){:}\ (x{:}T,\ U)$.
- If $M{:}\ (x{:}T,\ U)$ and $x{:}T, y{:}U \vdash P$ well-typed, then so is $\mathbf{split}\ M\ \mathbf{is}\ (x{:}T, y{:}U);P$.

This is a standard dependent record type; in $(x{:}T,\ U\{x\})$ name x is bound with scope $U\{x\}$.

- If x is not free in $U\{x\}$ we get ordinary pairs.

The Sum Type $T+U$

Terms of type $T+U$ are tagged variants, either of type T or of type U.

- If $M:T$ then $\mathbf{inl}(M):T+U$.
- If $M:U$ then $\mathbf{inr}(M):T+U$
- If $M:T+U$ and both $x:T \vdash P$ and $y:U \vdash Q$ well-typed, then so is $\mathbf{case}\ M\ \mathbf{is}\ \mathbf{inl}(x:T);P\ \mathbf{is}\ \mathbf{inr}(y:U);Q$

Motivation: type of a key for encrypting plaintexts of two different types: $\mathbf{Key}(T+U)$

What Do We Have So Far?

Net ≜ **Un**
Msg ≜ **Un**
MyNonce ≜ **Un**
MyKey ≜ **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)
- mixing keys and channels (cannot encrypt with Msg)

But Can We Check Assertions?

Much as before, let the judgment

$$E \vdash P : [M_1,\ldots,M_n]$$

mean the multiset $[M_1,\ldots,M_n]$ is a bound on the terms that P may **end** but not **begin**.

If $M{:}T$ then **end** $M : [M]$

If $M{:}T$ and $P{:}e$ then **begin** $M;P : e - [M]$

Fine for straight-line code, but need to allow inter-process messages to transfer effects.

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)

Cannot Transfer via **Un**

Suppose, somehow, each untrusted type **Un** has an effect *e*, and we:

- allow an untrusted input to mask the effect *e*, but
- require an untrusted output to incur the effect *e*.

Unsound. Although have 1-1 correspondence, we need to type opponents using **Un.**

So cannot enforce requirement on **Un** outputs.

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.

Can Transfer 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)

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
- Implicitly checks x is a name

It incurs the effect e:

- If $E \vdash x : $ **Un** and $E, y{:}$**Nonce** $e \vdash P : e'$ then $E \vdash$ **cast** x **to** (y:**Nonce** e);P : e+e′
- Only kind of 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 masks the effect *e*:

- 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 **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

Summary of Spi Types

T, U ::=	types
Un	untrusted data
Ch T	channel
(x:T,U)	dependent pair
T+U	tagged variant
Key T	symmetric key
Nonce e	nonce, witnessing e

Ex: Multiple Messages Again

```
net : Un
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);
```

Ex: Multiple Messages Cont.

```
recv(k:MyKey,d:Un):[] ≜
  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.)

Robust Safety by Typing

Theorem (Robust Safety)
If $x_1,\ldots,x_n{:}\mathbf{Un} \vdash P : []$ then P is robustly safe.

In the multiple messages example:

$$net,msg_1,\ldots,msg_n,d{:}\mathbf{Un} \vdash sys(msg_1,\ldots,msg_n,d) : []$$

Hence, authenticity is a corollary of the theorem.

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

Typing the WMF

$p_1, \ldots, p_n$,s: Prin ≜ **Un** --n principals, one server

SKey = **Key** T --session keys, for some payload T

kp_is: PrincipalKey(p_i) --longterm key for each principal

PrincipalKey(p) ≜ **Key**(Cipher3(p) + Cipher6(p))

Cipher3(a) ≜ (b:Prin, kab:SKey, ns:**Nonce**["a sending b key kab"])

Cipher6(b) ≜ (a:Prin, kab:SKey, ns:**Nonce**["a sending b key 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.

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

Typing Woo and Lam

Event 1	a begins	"a proving presence to b"
Message 1	a → b:	a
Message 2	b → a:	nb
Message 3	a → b:	$\{tag3(b,nb)\}_{kas}$
Message 4	b → s:	$b,\{tag4(a, \{tag3(b,nb)\}_{kas})\}_{kbs}$
Message 5	s → b:	$\{tag5(a,nb)\}_{kbs}$
Event 2	b ends	"a proving presence to b"

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

Cipher3(a) ≜ (b:Prin, nb:**Nonce**["a proving presence to b"])

Cipher4(b) ≜ (a:Prin, cipher:**Un**) --seems redundant

Cipher5(b) ≜ (a:Prin, nb:**Nonce**["a proving presence to b"])

Typing Woo and Lam, again

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

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

Cipher3(a) ≜ (b:Prin, nb:**Nonce**[“a proving presence to b”])

Cipher5(b) ≜ (a:Prin, nb:**Nonce**[“a proving presence to b”])

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

Abadi and Needham’s version

Message 1	a → b:	a,b,na
Message 2	b → 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 → b:	$\{tag3a(a,b,kab,na)\}_{kas},\{tag3b(a,b,kab,nb)\}_{kbs}$
Event 3	b ends	“responder b key kab for a”
Message 4	b → a:	$\{tag3a(a,b,kab,na)\}_{kas}$
Event 4	a ends	“initiator a key kab for b”

PrincipalKey(p) ≜ **Key**(Cipher3a(p) + Cipher3b(p))

Cipher3a(a)≜(a′,b:Prin,kab:SKey,na:**Nonce**[“initiator a key kab for b”])

Cipher3b(b)≜(a,b′:Prin,kab:SKey,nb:**Nonce**[“responder b key kab for a”])

Typing Spi: Status

Abadi already proposes typing for guaranteeing secrecy properties in spi.

Our types can guarantee correspondences, with little human work, for unbounded opponents.

Typing WMF took 30 minutes not 3 months!

Lots of open questions...

- Need more typing rules for examples...
- Relation to other notions of authenticity?
- Can we deal with partially trusted opponents?
- Can we type other uses of nonces, other primitives?

Relation to Model Checking...

Finite model checking, since Lowe, is popular:

- User codes model, opponent, specification
- Automatic discovery of attacks, but limited opponent, so “false negatives”

Type checking spi shows promise:

- Additionally, user invents types
- Automatic type-checking, unlimited opponent, but “false positives”

...and to Deductive Reasoning

Deductive reasoning in specific (e.g. BAN) or generic (e.g. HOL) logics complementary:

- Typically, user needs to code model and specification, guide construction of the proof
- Unlimited opponent, so no “false negatives”

Recent tools require little user intervention:

- Song’s Athena (strand spaces)
- Heather and Schneider (rank functions)

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.

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