Security protocols

Spi-calculus

The Spi-calculus and Related Calculi

- The spi-calculus is a successor of the lambda-calculus for reasoning about programs and processes.
- The lambda-calculus (Church, 1930s).
  - a calculus of functions (in the sense of functional programs)
  - originally invented by logicians as a foundation of mathematics (1930s)
  - later used in computer science as the basis of seminal work on programming language semantics (early 1970s)
  - now the core of functional programming languages (e.g., Haskell, OCaml, Clean, Lisp)

The Spi-calculus and Related Calculi

- The pi-calculus (Milner/Parrow/Walker, late 1980s).
  - a calculus for modeling concurrent processes
  - the feature that distinguishes the pi-calculus from related process calculi (e.g., CSP, CCS): dynamic channel creation
- The spi-calculus (Abadi/Gordon, 1997).
  - an extension of the pi-calculus with cryptographic primitives
  - dynamic creation of new names allows us to model the generation of unguessable entities like nonces and keys

Basic Idea

- a simple syntax
- an operational semantics to model program execution

  Think of these calculi as little programming languages:
  the operational semantics is like an interpreter.

Names and Variables

- Names
  - n, m, l, k, c, d, e
- Variables
  - x, y, z
- We assume we have two disjoint infinite sets of **names** and **variables**.
- Names represent constants.
- Variables are placeholders for names. They can be substituted by any name.
- By convention, we use the meta-variables n, m, l, k, c, d, e to range over names and x, y, z to range over variables.

Messages

- Messages
  - M, N, L, K
- Msg ::=   n ‖ x

- The syntax class of **messages** contains variables and names.
- *In the pure pi-calculus, there are no other messages*.
- By convention, we use the meta-variables M, N, L, K to range over messages.

Processes

- Proc ::= stop ‖ P ; Q ‖ P | Q ‖ P
- We use P, Q, R to range over **processes**.
- Inactivity. **stop** (“other notation”: **0**)
  - we usually omit the stop process, so if P is not ended by a stop, we assume P is a shorthand for P; stop
- Sequential concatenation of processes: ";"
- Parallel composition P | Q executes proc P and Q in parallel.
  - Operator precedence. The semicolon binds more tightly than the parallel bar: P1 ; P2 | P3 is (P1 ; P2) | P3
- Replication.
  - The process !P behaves like P | P | P | · · · .
  - !P | Q is (!P) | Q

Output and Input

- Proc ::= ... ∥ out N M; P %other notation: $\overline{N}\langle M\rangle.P$

  ∥ inp N x; P %other notation: N(x).P
- Output. A process that sends message M onto channel c and then stops:
  - out c M; stop
- A process that sends M onto c, then M onto d, and then stops:
  - out c M; out d M; stop
- Input. The process inp N x; P
  - waits for a message to arrive on channel N,
  - takes it out of the channel and binds it to variable x,
  - and then continues with process P.

Output and Input examples

- A process that redirects a message from channel c to channel d:
  - inp c x; out d x; % we omit the stop process
- A process that duplicates a message on channel c:
  - inp c x; out c x; out c x;
- Input and output synchronize
  - out c M; | inp c x; out c x; out c x;

    The first process sends M onto c.

    The second process **waits** for input on c.

    These two processes do interact!

    Process interaction is governed by the operational semantics.

The Pi-calculus: Bound Variables

- The variable x in   inp N x; P   is a **binder** whose scope is P.
- In this situation, occurrences of x in P are called **bound**.
- The identities of bound variables are inessential. Intuitively, there is no difference between the following processes:
  - inp c x; out d x;
  - inp c y; out d y;

    To capture this intuition we identify processes that only differ by renaming of bound variables.

The Pi-calculus: Free Variables

- A variable occurrence in **free**, if it is neither a binder nor a bound occurrence.
  - out c x;
  - inp c y; out d x;

  In both processes the occurrence of x is free.

  - inp c x; out d x;

  Neither of the two occurrences of x in this process is free. The first occurrence is a binder, and the second is bound.

  - out c x; | inp c x; out d x;

  The first occurrence of x is free but the second and third occurrences are not.

Bound Variables are Private

- Consider
  - out c x; stop | out d x; stop
  - the x on the left of the | is **the same x** we see on the right hand side
  - these are two instances of the **same** variable
  - so, on channel d and channel c we'll see the same message
- BUT
  - inp c x ; out c x; stop | inp c x ; out d x; stop
  - the x on the left of the | is **not the same x** we see on the right hand side
- Indeed the latter is equivalent to
  - inp c y ; out c y; stop | inp c x ; out d x; stop

Renaming bound variables is a good practice

Closed Processes and Closed Messages

- **A process P is closed** if If no variable occurrence in P is free
  - inp c x; out d x;

  This process is closed.

  - out c x; stop
  - inp c y; out d x; stop
  - out c x; | inp c x; out d x;

  None of these processes is closed because each contains a free occurrence of x.

  > Note (for future reference) that closed processes may contain (free) names.

- **A message M is closed** if it does not contain variables.
  - In the pi-calculus the only closed messages are names ...
  - ... but in the spi-calculus there will be more closed messages.

The Pi-calculus: new, bound and free names

- Proc ::= ……. ‖ new n ; P
- Name generation: generates a name n (whose scope is P) and continues with P.
  - In pi-calculus, name generation is typically used to model dynamic creation of channels.
  - In spi-calculus, name generation is typically used to model dynamic generation of cryptographic keys and nonces.
- Other presentations of pi sometimes use different syntax:

  $(\nu n)P$

- In new n; P, the occurrence of n is a **binder** whose scope is P
- The definition of bound and free names is analogous to our definition of bound and free variables.

Bound Names are Private

  - out c n; stop | out d n; stop
  - the n on the left of the | is **the same n** we see on the right hand side
  - this message is guessable by any process in which n is free
- BUT
  - new n; out c n; stop | new n; out d n; stop
  - the n on the left of the | is **not the same n** we see on the right hand side - on channel d and channel c we'll see two different, unguessable messages
- Indeed the latter is equivalent to
  - new n; out c n; stop | new m; out d m; stop
- **Renaming bound names with new names is a good practice, and it can always be done**

Pi-calculus: new stuff, example

- Suppose client A wants to receive from server B a very large message M, but they only share a low-bandwidth channel.
- A can "create" a high-bandwidth channel, and send this channel to B, who then sends M to A via this fast channel.
- Formalized in pi:
  - PA == new high; out low high; inp high y;
  - PB == inp low x; out x M;
  - P == PA | PB

More examples

- in c x ; (out c x | stop)
  - x is bound, it is the "same variable"
  - the process reads x from x and then either stops or outputs x on c again
- in c x ; stop | out c x ; stop
  - the first x is bound, the second is not, they are not the same variable.
  - you can better rename the bound variable and write
  - in c y ; stop | out c x ; stop
  - this gives a better idea of what the process does
- Good practice: rename bound variables and bound names to avoid conflicts.

Substitutions

- If M is a closed message, we define

  {M/x}P

  as the process that is obtained by replacing all free occurrences of x in P by M.
- Example.
  - {n/x}(out c x; out c x;) = out c n; out c n;
  - {n/x}(out c x; | inp c x; out d x;) = out c n; | inp c x; out d x;
    - Note that the binder x and the bound occurrence of x do not get replaced by n.
- There is one catch: the **bound name convention** (next slide)

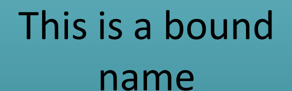
Bound Name Convention

- Consider this Example
  - {c/x}(new e; out d e; out d x;) == new e; out d e; out d c
  - On the lhs e is bound (and unguessable), c and d are free
  - The same applies to the rhs,
  - so on channel d we see two outputs: the first one is unguessable, the second is guessable and different from the first one.
- Now, renaming the bound names should in principle make no difference, but here there is a risk. Consider the following
  - {c/x}(new c; out d c; out d x;)

  If we apply the subsitution “without thinking” we get:
  - {c/x}(new c; out d c; out d x;) =??= new c; out d c; out d c

Bound Name Convention (2)

- We were looking at
  - {c/x}(new c; out d c; out d x;) =??= new c; out d c; out d c
- What is now wrong here?
  - operationally:
    - the process on the lhs outputs two different names on channel d.
    - the process on the rhs outputs the same (unguessable) name twice on d
  - what is wrong is that the leftmost "c" is free, while all other occurrences of "c" are bound.
  - {c/x}(new c; out d c; out d x;) =??= new c; out d c; out d c



Bound Name Convention (3)

- To compute {c/x}P, do the following
  - If c is bound in P, then rename all bound instances of c in P with a new name (not occurring elsewhere)
  - then replace all free instances of x in P with c

- In the spy calculus you also deal with more complex messages, but the reasoning is the same:
  - to compute {M/x}P, for each name c in M, if c is bound in P, then rename all bound instances of c in P with a new name (not occurring elsewhere)
  - then replace all free instances of x in P with M

The Pi-calculus: structural congruence

- We want to
  - ignore the order of processes in a parallel composition $P_1 \mid \cdots \mid P_n$
- So, we define an equivalence on processes:
  - $P \mid Q \equiv Q \mid P$
  - $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
  - $P \mid \text{stop} \equiv P$
  - $!P \equiv P \mid !P$
- These equivalence laws may be applied **anywhere inside process expressions**.
- The resulting equivalence relation on the set of processes is called **structural congruence**.

Structural congruence with new names

- structural congruence with new names
  - $P \mid \text{new } n \,;\, Q \equiv \text{new } n \,;\, (P \mid Q)$ (provided that n is not free in P)

  scope extrusion rule: it allows to expand
  the scope of the new n

The Pi-calculus: operational semantics (1)

- Basis: the **step relation** on closed processes
  - P -> Q

  Intuitively, it says that P can evolve into Q in one step.
- The transitive closure of the step relation:
  - P ->* Q

  Intuitively, P can evolve into Q in zero or more steps

Technically, P ->* Q is defined to hold whenever

- $P \equiv P_0 \to P_1 \to \ldots \to P_n \equiv Q$ for some $n \geq 0$

  so, it is defined modulo structural congruence

The Pi-calculus: operational semantics (2)

- Two administrative rules.

  [A1] If P ≡ P’, P’ -> Q’ and Q’ ≡ Q then P -> Q

  [A2] If P -> P’, then P | Q -> P’ | Q

  - Rule [A1] allows us to replace processes by congruent processes in between steps.

- step rule for new names
  - If P -> Q then new n; P -> new n; Q

The Pi-calculus: operational semantics (3)

- The step I/O rule.
  - out c M; P | inp c x; Q -> P | {M/x}Q

- Rule [A2] allows us to apply the I/O-rule also when the I/O part is in parallel with some other process (but not when it is preceded by another process) .
- The congruence P | new n ; Q ≡ new n ; (P | Q) [provided that n is not free in P] allows to apply the I/O rule “under” a "new" operator.

	out c n; | inp c x; out c x; out c x;
->	{n/x}(out c x; out c x;)
=	out c n; out c n;

The Pi-calculus: operational semantics (4)

- **Non-determinism** (very important).
- Let P == out c n; | inp c x; out d x; | inp c x; out c x;
  - P -> out d n; | inp c x; out c x;
  - P -> inp c x; out d x; | out c n; -> out d n;

  The same process P can evolve in two different ways with two different final results.
- The operational semantics is non-deterministic.

The Pi-calculus: operational semantics (5)

- A non-example.
  - inp c x; (out d n; | inp d x; stop) -/-> inp c x; stop
- There is no administrative rule that allows to apply the I/O rule under an input prefix.
  - This is intentionally forbidden.
- Text under in- or output prefixes is regarded as static program text that only becomes executable after the prefix IS REMOVED.

Conditional

- Proc ::= .... ‖ if M = N then P

- **Conditional**. The process
  - continues with P if M = N,
  - and stops otherwise.
- Operational semantics:
  - if M = M then P -> P
- Operator precedence. Conditional binds more tightly than the parallel bar:
  - if N = M then P | Q is (if N = M then P) | Q

A communication protocol in Pi (1)

- Take the following simple protocol narration

A -> B : "How are you?"

B -> A : "Fine, thanks!"

- How do we translate this into PI-calculus? (next slide)

A communication protocol in Pi (2)

```
PA ==   out c "How are you?"; inp d y; stop
PB ==   inp c x;
        if x = "How are you?" then
        out d "Fine, thanks!";
P == PA | PB
```

- execution

P == out c "How are you?"; inp d y; stop | inp c x; if x = "How are you?" then out d "Fine, thanks!";

-> inp d y; stop | if "How are you?" == "How are you?“ then out d "Fine, thanks!";

-> inp d y; stop | out d "Fine, thanks!";

-> stop

Example of infinite duplication

- PA | PB
  - A sends one "How are you?"
  - B replies with "Fine, thanks!". And that's it.
- !PA | !PB
  - A keeps sending "How are you?" and waiting for replies.
  - B keeps waiting for "How are you?" and replying with "Fine, thanks!".
- Why is that?
  - !PA | !PB ≡ PA | !PA | PB | !PB ≡ PA | PB | !PA | !PB
    ->* stop | !PA | !PB ≡ !PA | !PB

Example

- PA = new high; out low high; inp high y;
- PB = inp low x; out x M;

- A run
  - PB | PA == inp low x; out x M; | new high; out low high; inp high y;

  (apply scope extrusion)
  - ≡ new high; (inp low x; out x M; | out low high; inp high y;)
  - -> new high; (out high M; | inp high y;)
  - -> new high; stop

Pi-calculus

- Is finished.

- Now we move to the SPI calculus, which extends the Pi with constructs which can model cryptographic primitives.

The Spi-calculus: Message Constructors

$M ::= \; ...$	$\parallel (M_1 , . . . , M_n)$	(tuples)
	$\parallel \{M\}K$	(symmetric encryption)
	$\parallel \#(M)$	(hashing)
	$\parallel \{\vert M \vert\}K$	(asymmetric encryption)
	$\parallel Enc(K)$	(encryption key of asymmetric key K)
	$\parallel Dec(K)$	(decryption key of asymmetric key K)

- For asymmetric keys, the decryption key is different from the encryption key
- sometimes we write K^{-1} to indicate the decryption key of K
- If $M_1 , . . . , M_n$ are closed message, we define $\{M_1 , . . . ,M_n /x_1, . . . , x_n \}P$: as the process that is obtained by simultaneously replacing all free occurrences of $x_1 , . . . , x_n$ in P by $M_1 , . . . , M_n$ (renaming bound names in P to avoid capture of free names).

The Spi-calculus: Message Destructors

P ::= . . . ‖ split M is (x1, . . . , xn); P (xi is binder with scope P)

‖ decrypt M is {x}K ; P (x is binder with scope P)

‖ decrypt M is {|x|}K^{-1} ; P (x is binder with scope P)

- split M is (x1, . . . , xn); P

  If M is an n-tuple, its components get bound to x1, . . . , xn and P gets executed. If M is not an n-tuple, the process stops.

  split (M1, . . . ,Mn) is (x1, . . . , xn); P -> {M1, . . . ,Mn/x1, . . . , xn}P

- decrypt M is {x}K ; P

  If M is a ciphertext encrypted with symmetric key K, then the plaintext gets bound to x and P gets executed. Otherwise, the process stops. For instance in the case in which that M is encrypted with a key K1 different from K (simple syntactic check).

  decrypt {M}K is {x}K ;P -> {M/x}P

- decrypt M is {|x|}K^{-1} ; P

  If M is a ciphertext encrypted with the inverse of decryption key K, then the plaintext gets bound to x and P gets executed. Otherwise, the process stops.

  decrypt {|M|}Enc(K) is {|x|}Dec(K); P -> {M/x}P

If the keys are different

- A == new k, s; out c {s}k
- B == new k; inp c x; decrypt x is {y}k; …
- Notice that in this case
  - c is a shared name
  - k is a local name for each one of the process
  - B is not supposed to be able to decrypt the message (he doesn't have k)
- A|B == new k, s; out c {s}k | new k; inp c x; decrypt x is {y}k; …

  To get rid of the "new" we have to rename at least one of the k's

  == new k1, s; out c {s}k1 | new k2; inp c x; decrypt x is {y}k2;

  == new k1, k2, s (out c {s}k1 | inp c x; decrypt x is {y}k2; …)

  … and indeed B will never decrypt it.

Example: a security protocol

A -> B : (M, A)

B -> A : N

A -> B : {|#(M, B, N)|}sA

- A signs the last message (it encrypts it with her secret key)
- Initiator:

  what does the initiator know? His identity A, the responders identity B and her own signing key sA. (and the channel upon which all communication takes place)

```
Init(net, A, B, sA) ==   new m;
                         out net (m,A);
                         inp net x;
                         out net {|#(m, B, x)|}sA;
```

Example: a security protocol

A -> B : (M, A)

B -> A : N

A -> B : {|#(M, B, N)|}sA

- Responder:
  - what does he know?
  - The network interface net, his identity B, the initiator's identity A and her public key pA.

```
Resp(net, B, A, pA)  ==  inp net x; split x is (m, a);
                         if a = A then
                         new n; out net n;
                         inp net y; decrypt y is {|z|}pA;
                         if z = #(m, B, n) then stop
// now B knows that m is from A
```

Putting things together (1)

- We have defined two processes Init() and Resp()
- We need to define a process that “calls” both of them
- This process will have to
  - Give Init() and Resp() enough information to operate
  - Give the intruder too enough information to be able to operate
- We are going to call this process “system”

Putting things together (2)

```
A -> B : (M,A)
B -> A : N
A -> B : {|#(M,B,N)|}sA
```

- system(net) == new A; new B; new kA;

```
( out net (A, B, Dec(kA)) |       //this data is public
    !Init(net, A, B, Enc(kA)) |
    !Resp(net, B, A, Dec(kA)) )
```

net is the only free name of the system.
Free names model public data.

The replications:
each agent can run several sessions in sequence or in parallel.

Execution trace

Sys ::= !Init(net, A, B, Enc(kA)) | !Resp(net, B, A, Dec(kA)))

- system(net) == new A; new B; new kA; (out net (A, B, Dec(kA)) | Sys)

-> new m; out net (m,A); inp net x; out net {|#(m, B, x)|}sA |
Resp(net, B, A, Dec(kA)) | Sys

-> out net (m,A); inp net x; out net {|#(m, B, x)|}sA |
inp net x; split x is (m, a); if a = A then new n; out net n;
inp net y; decrypt y is {|z|}pA; if z = #(m, B, n) then stop | Sys

-> inp net x; out net {|#(m, B, x)|}sA |
split (m, A) is (m, a); if a = A then new n; out net n;
inp net y; decrypt y is {|z|}pA; if z = #(m, B, n) then stop | Sys

Execution trace

-> inp net x; out net {|#(m, B, x)|}sA |
split (m, A) is (m, a); if a = A then new n; out net n;
inp net y; decrypt y is {|z|}pA; if z = #(m, B, n) then stop | Sys

-> inp net x; out net {|#(m, B, x)|}sA | out net n;
inp net y; decrypt y is {|z|}pA; if z = #(m, B, n) then stop | Sys

-> out net {|#(m, B, n)|}sA | inp net y; decrypt y is {|z|}pA;
if z = #(m, B, n) then stop | Sys

-> decrypt {|#(m, B, n)|}sA is {|z|}pA;
if z = #(m, B, n) then stop | Sys

-> Sys

Putting things together (3)

A -> B : (M,A)
B -> A : N
A -> B : {|#(M,B,N)|}sA

- if we want both agents to take both roles:

```
system(net) ==     new A; new B; new kA; new kB
                   out net (A, B, Dec(kA), Dec(kB)) |
                   !Init(net, A, B, Enc(kA)) |
                   !Resp(net, B, A, Dec(kA)) |
                   !Init(net, B, A, Enc(kB)) |
                   !Resp(net, A, B, Dec(kB)) |
```

- more agents ?

Attack

A -> B : (M,A)
B -> A : N
A -> B : {|#(M,B)|}sA
(remove N)

1. A -> B : (M,A)
2. B -> A : N
3. A -> I : {|#(M,B)|}sA
4. I -> B : {|#(M,B)|}sA
5. I -> B : (M,A)
6. B -> I : N
7. I -> B : {|#(M,B)|}sA

```
Intruder(net, A, M) == inp net signed;
                       (  out net signed | out net (M,A) |
                          inp net r;  out net signed; )
```

Execution trace with Intruder

Pub ::= out net Dec(kA)

Sys ::= !Pub | !Init | !Resp   Int ::= !Intruder(net, A, B, M)

System(net, A, B, M) | !Intruder(net, A, B, M)

== (new kA; Sys) | Int

(1) -> new kA; (out net (M, A); inp net x; out net {|#(M, B)|}sA)
| inp net x; split x is (m, a); if a = A then new n; out net n;
inp net y; decrypt y is {|z|}pA; if z = #(m, B) then stop |Sys)| Int

-> (inp net x; out net {|#(M, B)|}sA) |
split (M, A) is (m, a); if a = A then new n; out net n; inp net y;
decrypt y is {|z|}pA; if z = #(m, B) then stop |Sys) | Int

Execution trace with Intruder

-> (inp net x; out net {|#(M, B)|}sA) |
split (M, A) is (m, a); if a = A then new n; out net n; inp net y;
decrypt y is {|z|}pA; if z = #(m, B) then stop) |Sys) | Int

(2) -> (inp net x; out net {|#(M, B)|}sA | out net n; inp net y;
decrypt y is {|z|}pA; if z = #(M, B) then stop | Sys) | Int

(3) -> (out net {|#(M, B)|}sA | inp net y; decrypt y is {|z|}pA;
if z = #(M, B); | Sys) | inp net signed; (out net signed |
out net (M, A) | inp net r; out net signed;) | Int

(4) -> inp net y; decrypt y is {|z|}pA; if z = #(M, B); | Sys |
(out net {|#(M, B)|}sA | out net (M, A) |
inp net r; out net {|#(M, B)|}sA;) | Int

Execution trace with Intruder

-> decrypt {|#(M, B)|}sA is {|z|}pA; if z = #(M, B); | Sys |
(out net (M, A) | inp net r; out net {|#(M, B)|}sA;) | Int

(5) -> (Resp) inp net x; split x is (m, a); if a = A then new n;
out net n; inp net y; decrypt y is {|z|}pA; if z = #(m, B); | Sys |
(out net (M, A) | inp net r; out net {|#(M, B)|}sA;) | Int

-> split (M,A) is (m, a); if a = A then new n;
out net n; inp net y; decrypt y is {|z|}pA; if z = #(m, B); | Sys |
inp net r; out net {|#(M, B)|}sA; | Int

-> new n; out net n; inp net y; decrypt y is {|z|}pA; if z = #(M, B);
| Sys | inp net r; out net {|#(M, B)|}sA; | Int

Execution trace with Intruder

-> out net n; inp net y; decrypt y is {|z|}pA; if z = #(M, B); | Sys | inp net r; out net {|#(M, B)|}sA; | Int

(6) -> out net n; inp net y; decrypt y is {|z|}pA; if z = #(M, B); | Sys | inp net r; out net {|#(M, B)|}sA; | Int

-> inp net y; decrypt y is {|z|}pA; if z = #(M, B); | Sys | out net {|#(M, B)|}sA; | Int

(7) -> inp net y; decrypt y is {|z|}pA; if z = #(M, B); | Sys | out net {|#(M, B)|}sA; | Int

-> decrypt {|#(M, B)|}sA is {|z|}pA; if z = #(M, B);| Sys | Int

-> if #(M, B) = #(M, B);| Sys | Int

-> Sys | Int