Modelling and Analysing of Security Protocol: Lecture 8

Automatically Checking Protocols II

Tom Chothia
CWI

This Lecture

• Quick introduction to Prolog
• A protocol as Prolog rules
• From Prolog to ProVerif
• Checking secrecy
BREAKEK
• Writing protocols in the pi-calculus
• From secrecy to authenticity
• Examples: Diffie-Hellmen, STS & SKEME

The Pi-calculus

A mini language for writing protocols (and any other concurrent processes)

process P ::= in (channel, message); P
| out (channel, message); P
| let a = T in P
| new n; P
| P | Q
| ! P
| 0

The Pi-calculus

Firewall process

= ! ( in (168.42.12.5 , packet );
    let (port_no,payload) = packet in
    if port_no = 80 then
        out (to_server,payload)
    )

Pi-calculus Semantics

The Key Rules:

in (channel, var); P | out (channel, value) → P [ value/var ]

! P = P | ! P

P | new a; Q = new a; ( P | Q ) iff a not in P

The last rule means that:

out (a,b) | new a; in (a,x) ⇔

The Pi-calculus reduction

! Firewall_process
| out( 168.42.12.5 , ( 80, get, index.html ) )
| out( 168.42.12.5 , ( 22, login_ssh )
| out( 192.64.12.5 , ( 80, get, index.html ) )
Apply the !P ≡ P | !P rule

! Firewall process
| ( in (168.42.12.5 , packet );
  let (port_no,payload) = packet in
  if port_no = 80 then
    out (to_server,payload)
  )
| out( 168.42.12.5 , ( 80, (get, index.html) )
| out( 168.42.12.5 , ( 22, login_ssh)
| out( 192.64.12.5 , ( 80, (get, index.html) )

Apply the COMM rule

! Firewall process
| let (port_no,payload) = (80,(get, index.html) in
  if port_no = 80 then
    out (to_server,payload)
  )
| out( 168.42.12.5 , ( 22, login_ssh)
| out( 192.64.12.5 , ( 80, (get, index.html) )

Apply the LET rule

! Firewall process
| if 80 = 80 then
  out (to_server, (get, index.html))
| out( 168.42.12.5 , ( 22, login_ssh)
| out( 192.64.12.5 , ( 80, (get, index.html) )

Apply the IF rule

! Firewall process
| if 22 = 80 then
  out (to_server, login_ssh)
| out( 168.42.12.5 , ( 22, login_ssh)
| out( 192.64.12.5 , ( 80, (get, index.html) )

Apply the !P ≡ P | !P rule

! Firewall process
| ( in (168.42.12.5 , packet );
  let (port_no,payload) = packet in
  if port_no = 80 then
    out (to_server,payload)
  )
| out (to_server,(get, index.html))
| out( 168.42.12.5 , ( 22, login_ssh)
| out( 192.64.12.5 , ( 80, (get, index.html) )

Apply the COMM & LET rules

! Firewall process
| if 22 = 80 then
  out (to_server, login_ssh)
| out (to_server,(get, index.html))
| out( 192.64.12.5 , ( 80, (get, index.html) )
Apply the IF rules

<table>
<thead>
<tr>
<th>Firewall process</th>
</tr>
</thead>
<tbody>
<tr>
<td>out( (to_server,(get, index.html)) )</td>
</tr>
<tr>
<td>out(192.64.12.5, (80,(get, index.html)) )</td>
</tr>
</tbody>
</table>

The Applied Pi-calculus

- The Applied Pi-calculus extends the pi-calculus with equations...
  ... and we saw what you can do with equations in Lecture 2.
- It also adds frames '{ M/x }' to get track of what the environment knows.
- See the paper "Mobile Values, New Names and Secure Communication" for more info.

Example: Diffie-Hellman

- The Diffie-Hellman is a widely used key agreement protocol.
- It relies on some number theory:
  - a mod b = n where 3m s.t. a = m.b + n
- The protocol uses two public parameters
  - generator "g" (often 160 bits long)
  - prime "p" (often 1024 bits long)

Diffie-Hellman

- A and B pick random numbers \( r_A \) and \( r_B \) and calculate \( t_A = g^{r_A} \mod p \) and \( t_B = g^{r_B} \mod p \)
- The protocol just exchanges these numbers:
  1. \( A \rightarrow B : t_A \)
  2. \( B \rightarrow A : t_B \)
- A calculates \( t_B^{r_A} \mod p \) and B \( t_A^{r_B} \mod p \)
  - this is the key:
  - \( K = g^{r_A r_B} \mod p \)

Diffie-Hellman

- An observer cannot work out \( r_A \) and \( r_B \) from \( t_A \) and \( t_B \) therefore the attacker cannot calculate the key
- The values of "g" and "p" must be big enough to make it intractable to try all possible combinations.
- So we have a "Good Key" but know nothing about the participants.
- We did not need to share any keys at the start, therefore this is a very powerful protocol.
Station-to-Station Protocol

- The Station-to-Station (STS) protocol adds authentication:

1. $A \rightarrow B : t_A$
2. $B \rightarrow A : t_B, \{ \text{Sign}_B(t_A, t_B) \}_{k_{AB}}$
3. $A \rightarrow B : \{ \text{Sign}_A(t_A, t_B) \}_{k_{AB}}$

The Needham-Schroeder Public Key Protocol

A famous authentication protocol

1. $A \rightarrow B : E_B(N_a, A)$
2. $B \rightarrow A : E_A(N_a, N_b)$
3. $A \rightarrow B : E_B(N_b)$

$N_a$ and $N_b$ can then be used to generate a symmetric key

Needham-Schroeder in the Applied Pi-calculus

equations:  fun pk/1.  fun encrypt/2.
reduce decrypt(encrypt(x,pk(y)),y) = x.

A = new Na;
out (channel, encrypt( (Na, pk(skA)),pk(skB) ));
in (channel, message);
let (=Na,Nb) = decrypt( message, skA ) in
if Nx = Na then
out channel encrypt( Nb, pk(Bs) )

Needham-Schroeder in the Applied Pi-calculus

B = in (channel, message1);
let (Nx,pkA) = decrypt ( message, skB) in
new Nb;
out channel encryptl( (Nx,Nb), pkA );
in (channel, message2);
let Ny = decrypt (message2, skB) in
if (Ny = Nb) then ...
We must let the attacker pick who A talks to.

A = in(talk_to, pkB);
    new Na;
    out (channel, encrypt( (Na, pkA) pkB );
    in (channel, message);
    let (Nx,N) = decrypt( message, skA ) in
        if Nx = Na then
            out channel encrypt( N, pkB )

Correspondence Assertions

• We don’t really want to check secrecy.
• We want to check correspondence.
• We add events, and require implications between events.

Likewise for B

B = in(talk_to, pkA)
    in (channel, message1);
    let (Nx,=pkA) = decrypt ( message, skB) in
        new Nb;
        out channel encrypt( (Nx,Nb), pkA );
        in (channel, message2);
        let Ny = decrypt (message2, skB) in
            if (Ny = Nb) then ...

Adding A “begin” Assertions

A = in(talk_to, pkB);
    event begin(pk(skA),pkB);
    new Na;
    out (channel, encrypt( (Na,pkA) pkB );
    in (channel, message);
    let (Nx,N) = decrypt( message, skA ) in
        if Nx = Na then
            out channel encrypt( N, pkB )

Adding an “end” Assertions

B = in(talk_to, pkA)
    in (channel, message1);
    let (Nx,=pkA) = decrypt (message,skB) in
        new Nb;
        out channel encrypt( (Nx,Nb), pkA );
        in (channel, message2);
        let Ny = decrypt (message2, skB) in
            if (Ny = Nb) then
                event end(pkA,pk(skB))

Correctness

We now check than “end (A,B)” can happen if and only if “begin (A,B)” happened first.
Correctness

We now check that "end (A,B)" can happen if and only if "begin (A,B)" happened first.

This can be done by replacing everything after "begin(A,B)" with 0, then checking that "end(A,B)" stays secret.

The SKEME Protocol

- A better Diffie-Hellman from IBM.

- See handout.

This Lecture

- Quick introduction to Prolog
- A protocol as Prolog rules
- From Prolog to ProVerif
- Checking secrecy

BREAK
- Writing protocols in the pi-calculus
- From secrecy to authenticity
- Examples: Diffie-Hellman, STS & SKEME

Assessment

- You will get your homework back next week.

- And a new homework that involves BAN and ProVerif.

- TRY THEM OUT THIS WEEK.
Assessment

• From the 29th onwards you will be giving 20 mins presentations.

• You can formalise and verify a protocol (hard).

• or present a state-of-the-art research paper on security protocol (suggestions will be put on the website next week).

• Aim: show that you know what your talking about when it comes to security protocols.

• Try to find a paper you enjoy.