Special topics in Logic and Security I

Master Year II, Sem. I, 2025-2026

Ioana Leuştean
FMI, UB

References

1 Véronique Cortier, Steve Kremer. Formal Models and Techniques for Analyzing Security Protocols: A Tutorial. Foundations and Trends in Programming Languages, Now Publishers, 2014, 1 (3), pp.117. `https://hal.archives-ouvertes.fr/hal-01090874`

2 Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre, ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2023

A π-calculus for protocols: NSPK

For the Needham-Scroeder Public Key Protocol (NSPK):

$$
\begin{array}{ll}
A \longrightarrow B: & \{\!| A, na |\!\}_{pk(B)} \\
B \longrightarrow A: & \{\!| na, nb |\!\}_{pk(A)} \\
A \longrightarrow B: & \{\!| nb |\!\}_{pk(B)}
\end{array}
$$

the process corresponding to the initiator can be defined by:

$$
\begin{array}{ll}
P_A(skA, pkB) = & \nu\, n.out(c, aenc(pair(pk(skA), n), pkB)). \\
& in(c, x). \\
& \quad let\ z = adec(x, skA)\ in \\
& \quad\quad if\ \ fst(z) = n\ then\ out(c, aenc(snd(z), pkB)\ else\ 0.
\end{array}
$$

A π-calculus for protocols: NSPK [1]

For the Needham-Scroeder Public Key Protocol (NSPK):

$$
\begin{aligned}
A \longrightarrow B: &\quad \{\!|\, A, na \,|\!\}_{pk(B)} \\
B \longrightarrow A: &\quad \{\!|\, na, nb \,|\!\}_{pk(A)} \\
A \longrightarrow B: &\quad \{\!|\, nb \,|\!\}_{pk(B)}
\end{aligned}
$$

the process corresponding to the responder can be defined by:

$$
\begin{aligned}
P_B(skB) = \quad & in(c, x). \\
& \;let\ pkA = fst(adec(x, skB))\ in \\
& \;let\ y = snd(adec(x, skB))\ in \\
& \;\nu m.out(c, aenc(pair(y, m), pkA)). \\
& \;in(c, z). \\
& \;if\ adec(z, skB) = m\ then\ 0.
\end{aligned}
$$

A π-calculus for protocols: NSPK [1]

- a *process for each role*:

$$
\begin{aligned}
P_A(skA, pkB) = \quad & \nu\, n.out(c, aenc(pair(pk(skA), n), pkB). \\
& in(c, x). \\
& \ \ let\ z = adec(x, skA)\ in \\
& \ \ \ if\ \ fst(z) = n\ then\ out(c, aenc(snd(z), pkB)\ else\ 0.
\end{aligned}
$$

$$
\begin{aligned}
P_B(skB) = \quad & in(c, x). \\
& \ \ let\ pkA = fst(adec(x, skB))\ in \\
& \ \ let\ y = snd(adec(x, skB))\ in \\
& \ \ \nu m.out(c, aenc(pair(y, m), pkA)). \\
& \ \ in(c, z). \\
& \ \ if\ \ adec(z, skB) = m\ then\ 0.
\end{aligned}
$$

- a *process that brings them together*:

$$
\begin{aligned}
P_{NSPK} = \quad & \nu\, skA.\nu\, skB. \\
& \ \ let\ pkA = pk(skA)\ in \\
& \ \ \ let\ pkB = pk(skB)\ in \\
& \ \ \ \ \ out(c, pkA).out(c, pkB). \\
& \ \ \ \ \ \ (!P_A(skA, pkB)) \,|\, (!P_B(skB, pkA))
\end{aligned}
$$

ProVerif: typed π-calculus for NSPK

```
free c: channel.

(* Public key encryption *)
type pkey.
type skey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey;
                          adec(aenc(x, pk(y)),y) = x.

(* Signatures *)
type spkey.
type sskey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall x: bitstring, y: sskey; getmess(sign(x,y)) = x.
reduc forall x: bitstring, y: sskey; checksign(sign(x,y), spk(y)) =
```

ProVerif: typed π-calculus for NSPK

- *pi*-calculus

$$
\begin{aligned}
P_A(skA, pkB) = \quad & \nu\, n_a.out(c, aenc(pair(pk(skA), n_a), pkB)).\\
& in(c, x).\\
& \;\; let\ z = adec(x, skA)\ in\\
& \;\;\;\; if\ fst(z) = n_a\ then\ out(c, aenc(snd(z), pkB))\ else\ 0.
\end{aligned}
$$

- Proverif: typed π-calculus

```
let processA(pkB: pkey, skA: skey) =
  in(c, pkX: pkey);
  new Na: bitstring;
  out(c, aenc((Na, pk(skA)), pkX));
  in(c, m: bitstring);
  let (=Na, NX: bitstring) = adec(m, skA) in
  out(c, aenc(NX, pkX)).
```

ProVerif: typed π-calculus for NSPK

- *pi*-calculus

$$
\begin{aligned}
P_B(skB) = \quad & in(c, x). \\
& let\ pkA = fst(adec(x, skB))\ in \\
& let\ y = snd(adec(x, skB))\ in \\
& \nu n_b.out(c, aenc(pair(y, n_b), pkA)). \\
& in(c, z). \\
& if\ adec(z, skB) = n_b\ then\ 0.
\end{aligned}
$$

- Proverif: typed π-calculus

```
let processB(pkA: pkey, skB: skey) =
  in(c, m: bitstring);
  let (NY: bitstring, pkY: pkey) = adec(m, skB) in
  new Nb: bitstring;
  out(c, aenc((NY, Nb), pkY));
  in(c, m3: bitstring);
      if Nb = adec(m3, skB) then 0.
```

ProVerif: typed π-calculus for NSPK

- *pi*-calculus

$$
\begin{aligned}
P_{NSPK} = \quad & \nu\, skA.\nu\, skB. \\
& \quad let\ pkA = pk(skA)\ in \\
& \quad\ let\ pkB = pk(skB)\ in \\
& \quad\quad out(c, pkA).out(c, pkB). \\
& \quad\quad\ (!P_A(skA, pkB))\,|\,(!P_B(skB, pkA))
\end{aligned}
$$

- Proverif: typed π-calculus

```
process
new skA: skey; let pkA = pk(skA) in out(c, pkA);
new skB: skey; let pkB = pk(skB) in out(c, pkB);
( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

ProVerif: typed π-calculus for NSPK

```
let processA(pkB: pkey, skA: skey) =
  in(c, pkX: pkey);
  new Na: bitstring; out(c, aenc((Na, pk(skA)), pkX));
  in(c, m: bitstring);
  let (=Na, NX: bitstring) = adec(m, skA) in
  out(c, aenc(NX, pkX)).

let processB(pkA: pkey, skB: skey) =
  in(c, m: bitstring);
  let (NY: bitstring, pkY: pkey) = adec(m, skB) in
  new Nb: bitstring; out(c, aenc((NY, Nb), pkY));
  in(c, m3: bitstring);
  if Nb = adec(m3, skB) then 0.

process
  new skA: skey; let pkA = pk(skA) in out(c, pkA);
  new skB: skey; let pkB = pk(skB) in out(c, pkB);
( (!processA(pkB, skA)) | (!processB(pkA, skB)) )
```

ProVerif - security properties

- **Secrecy.** For the NSPK protcol we want to investigate if the nonces generated by A and B are available to an attacker. Recall that "the standard secrecy queries of ProVerif deal with the secrecy of private free names." [2]

  `private free secret.`

  `query attacker:secret.`

  Since in this case, the nonces are represented as new names (`Na` and `Nb`) or as variables (`NX` and `NY`) we use "the following general technique: instead of directly testing the secrecy of the nonces, we use them as session keys in order to encrypt some free name and test the secrecy of that free name." [2]

```
(* Shared key encryption *)
fun senc(bitstring,bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; sdec(senc(x,y),y) = x.

(* Secrecy queries *)
free secretANa, secretANb, secretBNa, secretBNb: bitstring [private].

query attacker(secretANa);
      attacker(secretANb);
      attacker(secretBNa);
      attacker(secretBNb).
```

ProVerif: typed π-calculus for NSPK

```
let processA(pkB: pkey, skA: skey) =
       in(c, pkX: pkey);
       new Na: bitstring; out(c, aenc((Na, pk(skA)), pkX));
     in(c, m: bitstring);
     let (=Na, NX: bitstring) = adec(m, skA) in
     out(c, aenc(NX, pkX));
     if pkX = pkB  then
     event endAparam(pk(skA));
     out(c, senc(secretANa, Na));
     out(c, senc(secretANb, NX)).
```

"in the process for Alice, we output senc(secretANa,Na) and we test the secrecy of secretANa: secretANa is secret if and only if the nonce Na that Alice has is secret. Similarly, we output senc(secretANb,NX) and we test the secrecy of secretANb: secretANb is secret if and only if NX (that is, the nonce Nb that Alice has) is secret." [2]

ProVerif: typed π-calculus for NSPK

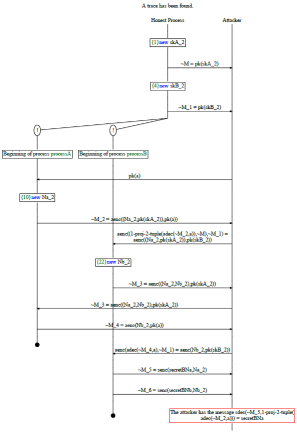
```
let processB(pkA: pkey, skB: skey) =
   in(c, m: bitstring);
   let (NY: bitstring, pkY: pkey) = adec(m, skB) in
   event beginAparam(pkY);
   new Nb: bitstring;
       out(c, aenc((NY, Nb), pkY));
   in(c, m3: bitstring);
   if Nb = adec(m3, skB) then
   if pkY = pkA then
   event endBparam(pk(skB));
   out(c, senc(secretBNa, NY));
   out(c, senc(secretBNb, Nb)).
```

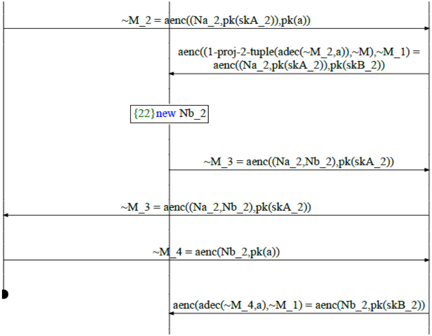
"Observe that the use of four names secretANa, secretANb, secretBNa, secretBNb for secrecy queries allows us to analyze the precise point of failure; that is, we can study secrecy for Alice and secrecy for Bob. Moreover, we can analyze both nonces Na and Nb independently for each of Alice and Bob."[2]

ProVerif: typed π-calculus for NSPK

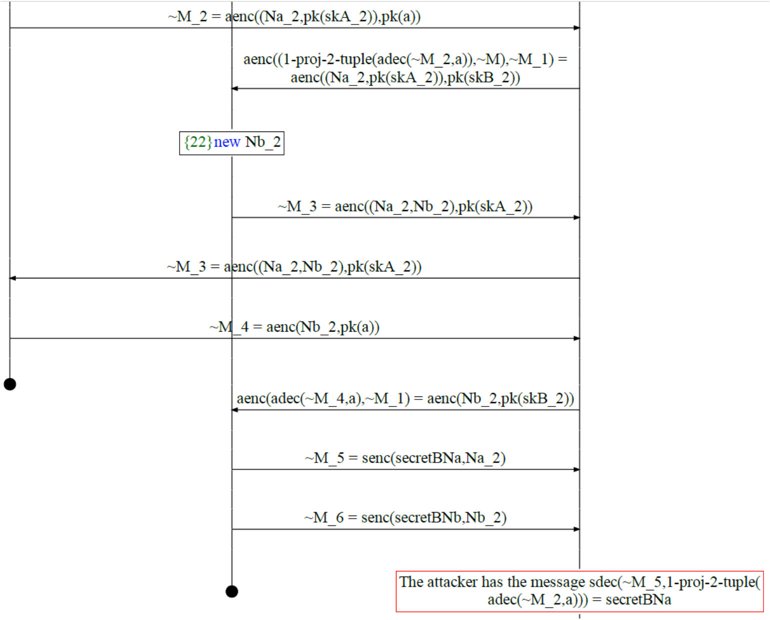
```
--------------------------------------------------------------
Verification summary:

Query not attacker(secretANa[]) is true.

Query not attacker(secretANb[]) is true.

Query not attacker(secretBNa[]) is false.

Query not attacker(secretBNb[]) is false.

--------------------------------------------------------------
```

Proverif - MiM attack





Proverif - MiM attack



ProVerif: security properties

- **Authentication as correspondence**. The protocol is annotated with *events* and authentication is defined using correspondence assertions:
  - `event beginAparam(pkey)` means that Bob believes that Alice initiated a run of the protocol with him,
  - `event endAparam(pkey)` means that Alice believes she has successfully completed the protocol with Bob,
  - `event beginBparam(pkey)` means that Alice initiate the protocol with Bob,
  - `event endBparam(pkey)` means that Bob believes that he has completed the protocol with Alice.

Proverif: events

- The relations between events are defined using correspondence assertions of the type:
  `query ev:e(t1,...,tn) ==> ev:e'(x1,...,xm).`
  which means that *if the event* `e` *has been executed then the event* `e'` *has been previously executed*.

"the event that occurs before the arrow ==> can be placed at the end of the protocol, but the event that occurs after the arrow ==> must be followed by at least one message output. Otherwise, the whole protocol can be executed without executing the latter event, so the correspondence certainly does not hold.

One can also note that moving an event that occurs before the arrow ==> towards the beginning of the protocol strengthens the correspondence property, and moving an event that occurs after the arrow ==> towards the end of the protocol also strengthens the correspondence property. Adding arguments to the events strengthens the correspondence property as well."[2]

ProVerif: security properties

```
(* Shared key encryption *)
fun senc(bitstring,bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; sdec(senc(x,y),y) = x.

(* Authentication queries *)
event beginBparam(pkey).
event endBparam(pkey).
event beginAparam(pkey).
event endAparam(pkey).

query x: pkey; event(endBparam(x)) ==> event(beginBparam(x)).
query x: pkey; event(endAparam(x)) ==> event(beginAparam(x)).

(* Secrecy queries *)
free secretANa, secretANb, secretBNa, secretBNb: bitstring [private]
query attacker(secretANa);
      attacker(secretANb);
      attacker(secretBNa);
      attacker(secretBNb).
```

ProVerif: NSPK security properties

```
let processA(pkB: pkey, skA: skey) =
    in(c, pkX: pkey);
    event beginBparam(pkX);
    new Na: bitstring;
    out(c, aenc((Na, pk(skA)), pkX));
    in(c, m: bitstring);
    let (=Na, NX: bitstring) = adec(m, skA) in
    out(c, aenc(NX, pkX));
    if pkX = pkB  then
    event endAparam(pk(skA));
    out(c, senc(secretANa, Na));
    out(c, senc(secretANb, NX)).
```

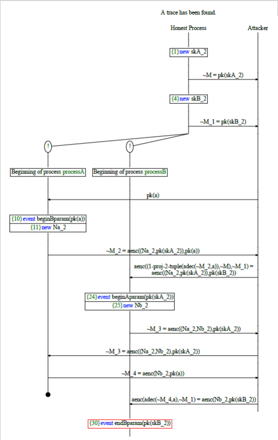
ProVerif: NSPK security properties

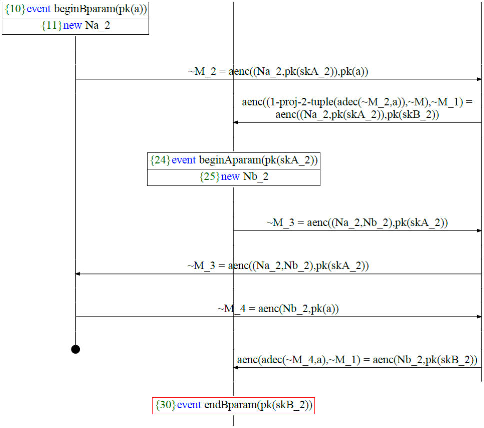
```
let processB(pkA: pkey, skB: skey) =
    in(c, m: bitstring);
    let (NY: bitstring, pkY: pkey) = adec(m, skB) in
    event beginAparam(pkY);
    new Nb: bitstring;
    out(c, aenc((NY, Nb), pkY));
    in(c, m3: bitstring);
    if Nb = adec(m3, skB) then
    if pkY = pkA then
    event endBparam(pk(skB));
    out(c, senc(secretBNa, NY));
    out(c, senc(secretBNb, Nb)).
```

ProVerif: typed π-calculus for NSPK

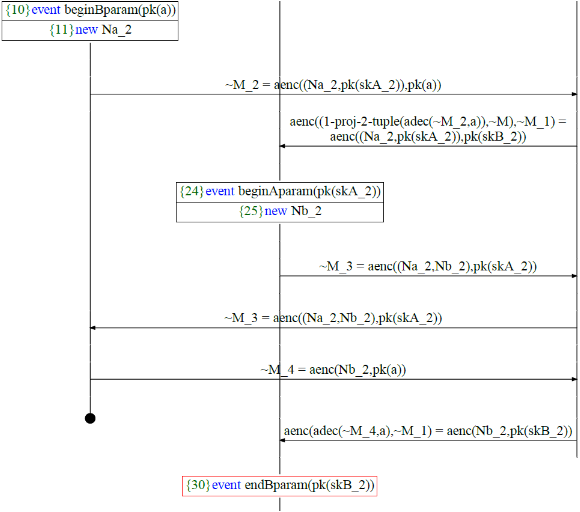
```
--------------------------------------------------------------
Verification summary:

Query inj-event(endBparam(x) ==>inj-event(beginBparam(x)) is false.

Query inj-event(endAparam(x))==>inj-event(beginAparam(x)) is true.

Query not attacker(secretANa[]) is true.

Query not attacker(secretANb[]) is true.

Query not attacker(secretBNa[]) is false.

Query not attacker(secretBNb[]) is false.

--------------------------------------------------------------
```

Proverif - MiM attack





Proverif - MiM attack



{10} `event beginBparam(pk(a))` {30} `event endBparam(pk(skB_2))`

References

1 Véronique Cortier, Steve Kremer. Formal Models and Techniques for Analyzing Security Protocols: A Tutorial. Foundations and Trends in Programming Languages, Now Publishers, 2014, 1 (3), pp.117. https://hal.archives-ouvertes.fr/hal-01090874

2 Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre, ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2023