Special topics in Logic and Security I

Master Year II, Sem. I, 2025-2026

Ioana Leuștean
FMI, UB

A π-calculus for protocols

Let $\mathcal{F}$ be a *signature*, $\mathcal{N}$ a set of *names* and $\mathcal{X}$ a set of variables such that $\mathcal{N}$ and $\mathcal{X}$ contain special subsets of *channel* names and *channel* variables, respectively.

- The *terms* are defined by:

u, v	$::=$	x, y, z	variables from $\mathcal{X}$
		a, b, c	names from $\mathcal{N}$
		$f(t_1, \ldots, t_n)$	function application for f from $\mathcal{F}$

- The *processes* are defined by:

P, Q	$::=$	0	the process that does nothing
		$in(u, x).P$	the input on channel u is bound to x
		$out(u, t).P$	the output on channel u is t
		$let\ x = t\ in\ P$	local definition
		$if\ t_1 = t_2\ then\ P\ else\ Q$	conditional
		$\nu\, n.P$	name restriction
		$P \mid Q$	parallel composition
		$!P$	replication, infinite parallel composition

where n is a name, x is a variable, t, t_1, t_2 are terms and u stands for a channel name or a channel variable.

Equational theory

- for asymmetric encryption the destructor is *adec* and the ecuation is $adec(aenc(x, pk(y)), y) = x$
- for the constructor *senc* the destructor is *sdec* and the equational theory for symmetric encryption is
  $sdec(senc(x, y), y) = x$
- for the constructors *sign*, the destructor can be *check* and the equational theory is
  $check(sign(x, y), pk(y)) = x$
- for the constructor *pair* the destructors are *fst* and *snd*, which extract the first and the second component.

Consequently, $\mathcal{F} = \{pk, aenc, senc, sign, pair\} \cup \{adec, sdec, check, fst, snd\}$. More operations (constructors, destructors) and equations can be considered, depending on the protocol we specify.

The Denning-Sacco protocol

$$A \longrightarrow B: \quad \{\!|\, \{\!|\, k \,|\!\}_{sk(A)} \,|\!\}_{pk(B)}$$
$$B \longrightarrow A: \quad \{\!|\, secret \,|\!\}_{k}$$

- a *process for each role*:

$$P_A(skA, pkB) = \quad \nu\, k.out(c, aenc(sign(k, skA), pkB)).\ in(c, x).\ let\ z = sdec(x, k)\ in\ 0$$

$$P_B(skB, pkA) = \quad in(c, y).\ let\ z = adec(y, skB)\ in\ if\ check(z, pkA) = xk\ then\ \nu\, s.out(c, senc(s, xk))$$

- a *process that brings them together*:

$$P_{DS} = \nu\, skA.\nu\, skB.(P_A(skA, pk(skB)) \mid P_B(skB, pk(skA)))$$

The Denning-Sacco protocol

- we allow multiple parallel executions for each role,
- the public keys available to a potential attacker,
- attacker interference: the initiator process receives the the public key of the communication partner through the channel c.

$$
\begin{aligned}
P_A(skA) = \quad & !\, in(c, x_{pk}).\\
& \nu\, k.out(c, aenc(sign(k, skA), x_{pk})).\\
& in(c, x).\\
& let\ z = sdec(x, k)\ in\ 0
\end{aligned}
$$

$$
\begin{aligned}
P_B(skB, pkA) = \quad & !\, in(c, y).\\
& let\ z = adec(y, skB)\ in\\
& \quad if\ check(z, pkA) = xk\ then\\
& \quad\quad \nu\, s.out(c, senc(s, xk))
\end{aligned}
$$

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

The structure of Proverif

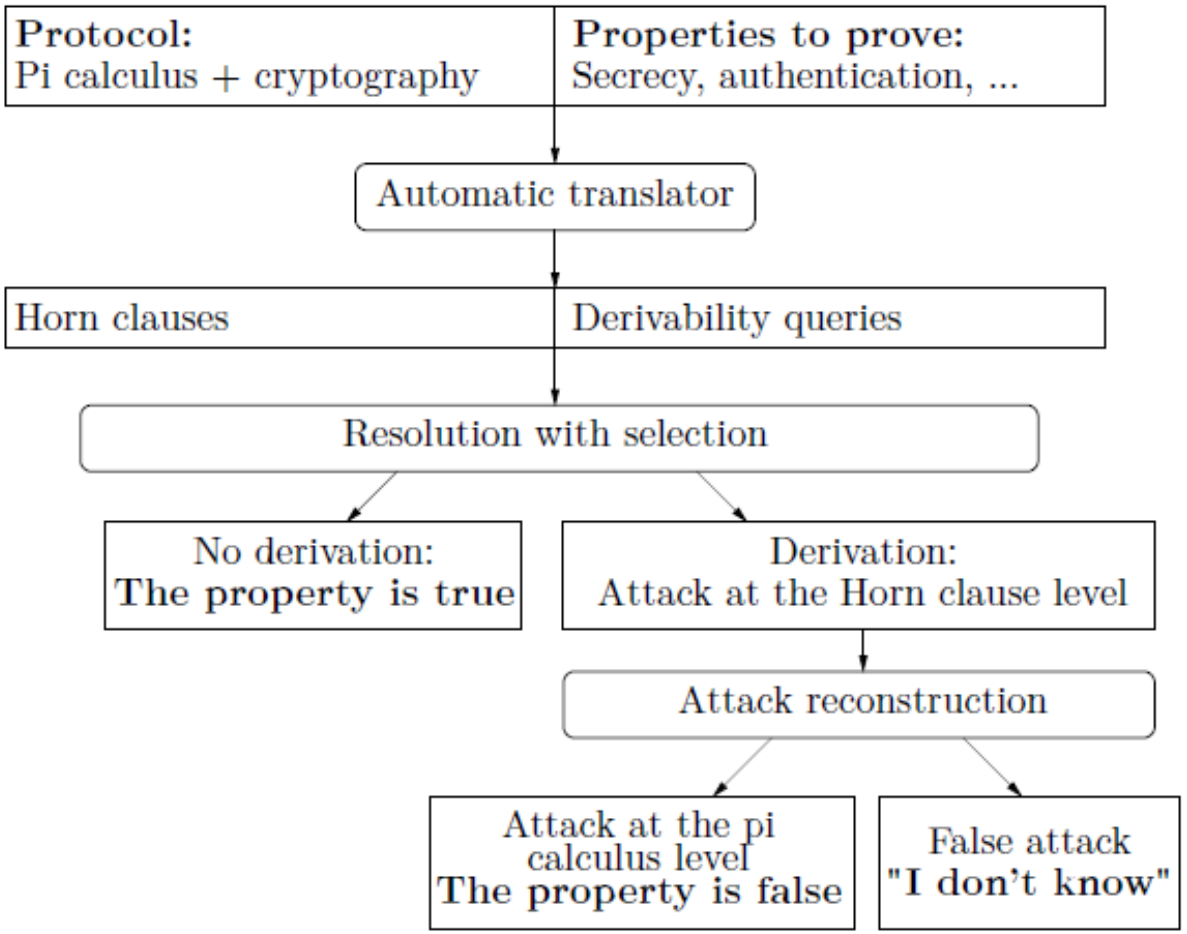


Figure 1.1: Structure of ProVerif

- Bruno Blanchet (2016), "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif", Foundations and Trends® in Privacy and Security: Vol. 1: No. 1-2, pp 1-135.

A false attack

```
fun sencrypt/2.
reduc sdecrypt(sencrypt(x,y),y) = x.

private free secret.
query attacker:secret.

process
        new k;
        out(c,sencrypt(sencrypt(secret,k),k));
        in(c,x);
        let s = sdecrypt(x,k) in
        out(c,s)
```

A false attack: output

```
Process 0 (that is, the initial process):
{1}new k;
{2}out(c, sencrypt(sencrypt(secret,k),k));
{3}in(c, x);
{4}let s = sdecrypt(x,k) in
{5}out(c, s)

-- Query not attacker:secret[] in process 0.
Translating the process into Horn clauses...
Completing...

Starting query not attacker:secret[]
goal reachable: attacker:secret[]
```

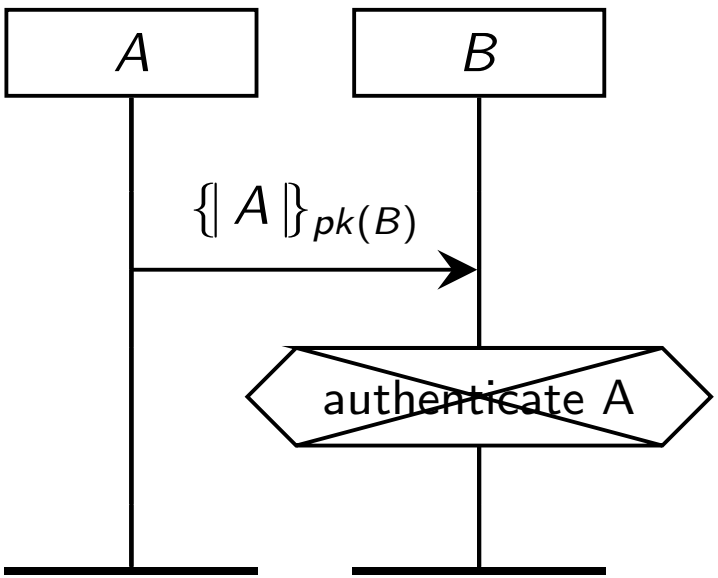
A false attack: output

```
Derivation:

1. The message sencrypt(sencrypt(secret[],k[]),k[]) may be sent to the attacker
at output {2}.
attacker:sencrypt(sencrypt(secret[],k[]),k[]).

2. The message sencrypt(sencrypt(secret[],k[]),k[]) that the attacker may have
by 1 may be received at input {3}.

So the message sencrypt(secret[],k[]) may be sent to the attacker at output {5}.
attacker:sencrypt(secret[],k[]).

3. The message sencrypt(secret[],k[]) that the attacker may have by 2 may be
received at input {3}.

So the message secret[] may be sent to the attacker at output {5}.
attacker:secret[].

4. By 3, attacker:secret[].
The goal is reached, represented in the following fact:
attacker:secret[].
```

A false attack: output

```
Unification of attacker:sencrypt(sencrypt(secret[],k[]),k[]) and
attacker:sencrypt(secret[],k[]) failed or made a constraint
 become false at the input at occurrence {3}.

Try adding a [precise] option on it.
Trying with the initial derivation tree instead.

Could not find a trace corresponding to this derivation.

RESULT not attacker:secret[] cannot be proved.
```

Simple Protocol: authentication



B cannot be sure that he speaks with A!

ProVerif: π-calculus for the Simple Protocol

```
free c.

(* Public key cryptography *)
fun pk/1.
fun encrypt/2.
reduc decrypt(encrypt(x,pk(y)),y) = x.

(* Signatures *)
fun sign/2.
reduc getmess(sign(m,k)) = m.
reduc checksign(sign(m,k), pk(k)) = m.

fun host/1.
reduc getkey(host(x)) = x.

(* Secrecy assumptions *)
not attacker:skA.
not attacker:skB.
```

```
let processA =
   in(c, pk2);
   event beginBparam(pk2);
   out(c, encrypt(host(pk2), pk2)).

let processB =
    in(c, km);
    let hostA = decrypt(km,skB) in
                event endBparam(pk(skB)).
process
    new skA;
    let pkA = pk(skA) in
     let hostA = host(pkA) in out(c, pkA);
    new skB;
    let pkB = pk(skB) in out(c, pkB);
    ((!processA) | (!processB))
```

ProVerif - security properties

- Secrecy
  - We investigate if a term is available to an attacker.
  - To test secrecy of the term M in the model, the following query is included in the input file before the main process:
    `query attacker (M).`

- Authentication as correspondence
  - "Correspondence assertions [WL93] are used to capture relationships between events which can be expressed in the form *if an event e has been executed, then event e' has been previously executed.* Moreover, these events may contain arguments, which allow relationships between the arguments of events to be studied."
  - "Informally, correspondence means that the execution of the different principals in an authentication protocol proceeds in an locked-strep fashion. In particular, **when an authenticating principal finishes its part of the protocol, the authenticated principal must have been present and and participated in its part of the protocol**." [WL93]
  - The protocol is annotated with *events* and authentication is defined using correspondence assertions.

Proverif - security properties: authentication

- The process calculus with events of the form:
  `event: e(t1,..., tn).P`
  With respect to a protocol, an event marks the part of the protocol that has been reached, **but does not affect its behavior**.
- 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*.
- Authentication can be defined using correspondence assertions: protocol is intended to ensure that, if A thinks he executes the protocol with B, then he really does so and vice-versa.

ProVerif: π-calculus for the Simple Protocol

```
query ev:endBparam(x) ==> ev:beginBparam(x).

let processA =
    in(c, pk2);
    event beginBparam(pk2);
    out(c, encrypt(host(pk2), pk2)).

let processB =
     in(c, km);
     let hostA = decrypt(km,skB) in
              event endBparam(pk(skB)).
```

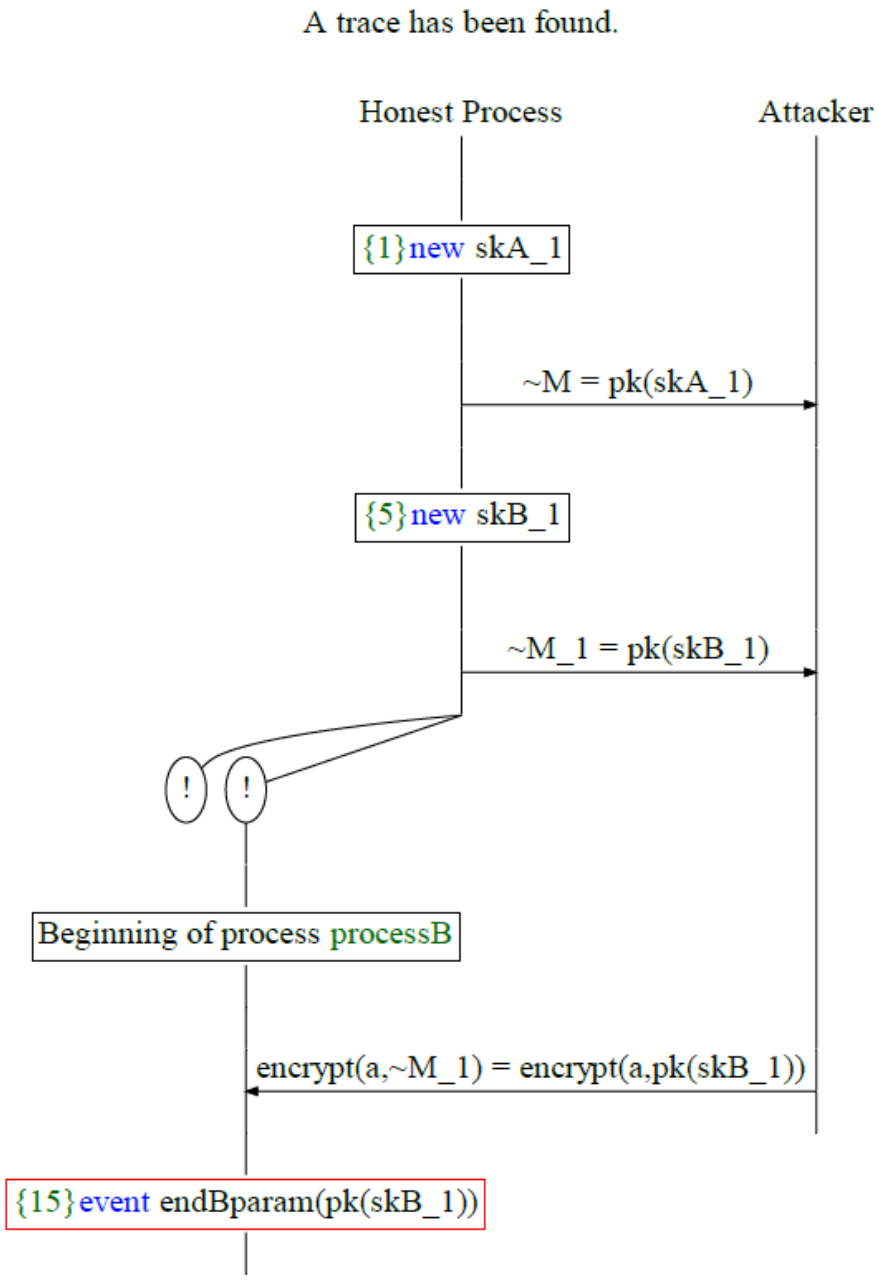
ProVerif: π-calculus for the Simple Protocol

```
query ev:endBparam(x) ==> ev:beginBparam(x).

let processA =

let processB =

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

⋮
A trace has been found.
RESULT: ev:endBparam(x) ==> ev:beginBparam(x) is false.
```

ProVerif: Simple Protocol

```
Derivation:

1. The message pk(skB[]) may be sent to the attacker at output {7}.
attacker:pk(skB[]).

2. The attacker has some term hostA_2.
attacker:hostA_2.

3. By 2, the attacker may know hostA_2.
By 1, the attacker may know pk(skB[]).
Using the function encrypt the attacker may obtain encrypt(hostA_2,pk(skB[])).
attacker:encrypt(hostA_2,pk(skB[])).

4. The message encrypt(hostA_2,pk(skB[])) that the attacker may have by 3 may be received at input {13}.
So event endBparam(pk(skB[])) may be executed at {15}.
event:endBparam(pk(skB[])).

5. By 4, event:endBparam(pk(skB[])).
The goal is reached, represented in the following fact:
event:endBparam(pk(skB[])).
```

ProVerif: Simple Protocol



π-calculus for the Simple Protocol with signing

```
(* Signatures *)
fun sign/2.
reduc getmess(sign(m,k)) = m.
reduc checksign(sign(m,k), pk(k)) = m.

query ev:endBparam(x) ==> ev:beginBparam(x).

let processA = in(c, pk2);
               event beginBparam(pk2);
               out(c, sign(k, skA)).
let processB = in(c, km);
               let  k = checksign(ks, pkA) in
               event endBparam(pk(skB)).
process
    new skA;let pkA = pk(skA) inout(c, pkA);
    new skB; let pkB = pk(skB) in  out(c, pkB);
((!processA) | (!processB))
```

RESULT: ev:endBparam(x) ==> ev:beginBparam(x) is true.

Proverif: events

- The process calculus with events of the form:
- 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.".
[ProVerif manual]

Typed pi-calculus [ProVerif manual]

https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf

Figure 3.2 Term and process grammar

$M, N ::=$	terms
a, b, c, k, m, n, s	names
x, y, z	variables
$(M_1, \ldots, M_k)$	tuple
$h(M_1, \ldots, M_k)$	constructor/destructor application
$M = N$	term equality
$M <> N$	term disequality
M && M	conjunction
$M \mid\mid M$	disjunction
$\mathbf{not}(M)$	negation
$P, Q ::=$	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\mathbf{new}\ n : t;\ P$	name restriction
$\mathbf{in}(M, x : t);\ P$	message input
$\mathbf{out}(M, N);\ P$	message output
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$	term evaluation
$R(M_1, \ldots, M_k)$	macro usage

Typed applied pi-calculus [ProVerif manual]

- "The concept of bound and free names is similar to local and global scope in programming languages; that is, free names are globally known, whereas bound names are local to a process. By default, free names are known by the attacker. Free names that are not known by the attacker must be declared private with the addition of the keyword **[private]**."
- "ProVerif also has built-in types: bitstring , bool, nat, sid,..."
- Security properties:
  - **secrecy:** query attacker (M)
  - **correspondence:** event e(M1, . . . ,Mn) ; P

    query x1 : t1,..., xn : tn ; event(e(M1...,Mj))==> event(e'(N1,...,Nk))
    query x1 : t1,..., xn : tn ; inj-event(e(M1,...,Mj))==> inj-event(e'(N1,...,Nk))

    where M1,..., Mj, N1,..., Nk are terms with the variables x1,...,xn

    event(e) ==> event(e') is `true` if for any occurrence of e there is an earlier occurrence of e'
    inj-event(e) ==> inj-event(e') is `true` if for any occurrence of e there is a distinct earlier occurrence of e'

ProVerif - typed pi-calculus: the Denning-Sacco protocol

Adapted from proverif2.05/examples/pitype/secr-auth/DenningSacco.pv

- The keys for encryption and the keys for signature are separated.
- The types used vor keys are user defined.
- For messages we usually use the built-in type `bitstring`.

```
free c: channel.

(* Shared key encryption *)
type key.

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

(* Signatures *)
type spkey.
type sskey.
```

ProVerif - typed pi-calculus: the Denning-Sacco protocol

(initial declarations continued)

```
(* Public key encryption *)
fun pk(skey): pkey.
fun encrypt(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey;
        decrypt(encrypt(x,pk(y)),y) = x.

(* Signatures *)
fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall m:bitstring, k:sskey; getmess(sign(m,k)) = m.
reduc forall m:bitstring, k:sskey; checksign(sign(m,k),spk(k)) = m.

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

ProVerif - typed pi-calculus: the Denning-Sacco Protocol

(initial declarations continued)

```
(* Authentication *)
event endBparam(pkey).
event beginBparam(pkey).

query x:pkey; event(endBparam(x)) ==> event(beginBparam(x)).
query x:pkey; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).


(* Secrecy assumptions *)
not attacker(new skA).
not attacker(new skB).

(* Queries *)
free  secretB: bitstring [private].
query attacker(secretB).
```

ProVerif: the Denning-Sacco Protocol

```
free c: channel.
free  secretB: bitstring [private].

let processInitiator(skA: sskey, pkA: spkey) = ...

let processResponder(skB: skey, pkB: pkey, pkA: spkey) = ...

(* Main process *)

process new skA: sskey;
let pkA = spk(skA) in
        out(c, pkA);
        new skB: skey;
let pkB = pk(skB) in
        out(c, pkB);
((!processInitiator(skA, pkA)) |
 (!processResponder(skB, pkB, pkA)))
```

ProVerif: the Denning-Sacco Protocol

The protocol is annotated with events:

- `event beginBparam(pkey)`, which denotes Initiator's intention to initiate the protocol with a partner whose public key is `pkey`,

```
event endBparam(pkey).
event beginBparam(pkey).
```

```
let processInitiator(skA: sskey, pkA: spkey) =
        in(c, pk2: pkey);
     event beginBparam(pk2);
        new k: key;
    out(c, encrypt(sign((pkA, pk2, k), skA), pk2));
    in(c, m: bitstring);
    let s = sdecrypt(m,k) in 0.
```

ProVerif: the Denning-Sacco Protocol

The protocol is annotated with events:

- `event endBparam(pkey)`, which records Responser's belief that he has completed the protocol with the Initiator; he supplies his public key pk(skB) as the parameter.

```
event endBparam(pkey).
event beginBparam(pkey).
```

```
let processResponder(skB: skey, pkB: pkey, pkA: spkey) =
    in(c, km: bitstring);
      let ks = decrypt(km,skB) in
    let (=pkA, =pkB, k:key) = checksign(ks, pkA) in
    out(c, sencrypt(secretB, k));
       event endBparam(pkB).
```

ProVerif: the Denning-Sacco Protocol

```
query x:pkey; event(endBparam(x)) ==> event(beginBparam(x)).
query x:pkey; inj-event(endBparam(x)) ==> inj-event(beginBparam(x)).
query attacker(secretB).

>.\proverif direx\myDenningSacco.pv

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

Query event(endBparam(x)) ==> event(beginBparam(x)) is true.

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

Query not attacker(secretB[]) is true.

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

ProVerif: the Denning-Sacco Protocol

```
-- Process 1-- Query
event(endBparam(x)) ==> event(beginBparam(x)) in process 1

Translating the process into Horn clauses...
Completing...

ok, secrecy assumption verified: fact unreachable attacker(skA[])
ok, secrecy assumption verified: fact unreachable attacker(skB[])

Starting query

event(endBparam(x)) ==> event(beginBparam(x))

goal reachable: b-event(beginBparam(pk(skB[]))) -> event(endBparam(p

RESULT event(endBparam(x)) ==> event(beginBparam(x)) is true.
```

ProVerif: the Denning-Sacco Protocol

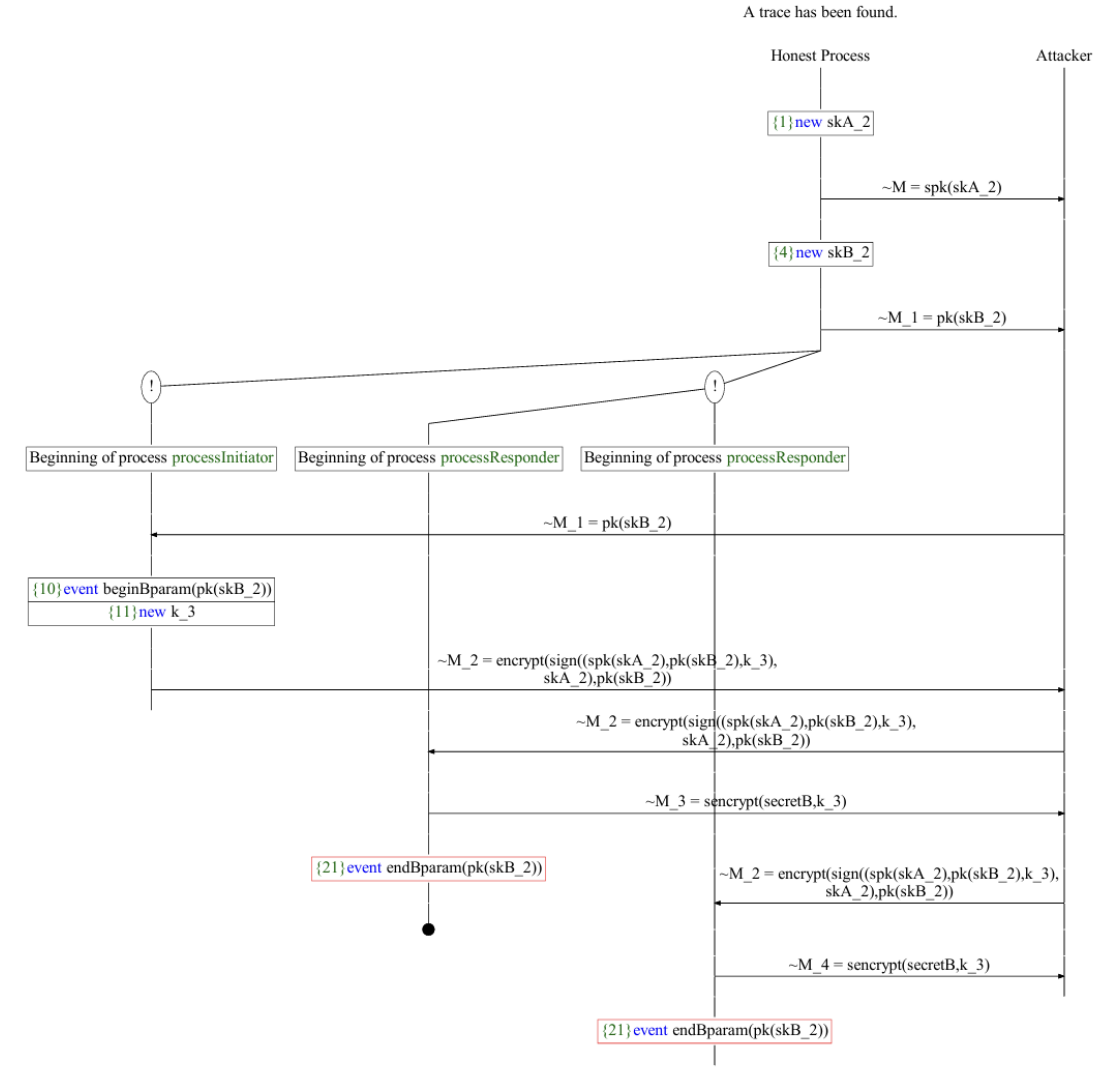
```
-- Query inj-event(endBparam(x)) ==> inj-event(beginBparam(x)) in process 1
Translating the process into Horn clauses...
Completing...
ok, secrecy assumption verified: fact unreachable attacker(skA[])
ok, secrecy assumption verified: fact unreachable attacker(skB[])
Starting query inj-event(endBparam(x)) ==> inj-event(beginBparam(x))
goal reachable: @sid =/= @sid_1 && b-inj-event(beginBparam(pk(skB[])),@occ10_1) -> i
```

In reprezentarea interna, evenimentele sunt parametrizate

```
Abbreviations:
k_2 = k[pk2 = pk(skB[]),!1 = @sid_2]
@occ21_1 = @occ21[km = encrypt(sign((spk(skA[]),pk(skB[]),k_2),skA[]),pk(skB[])),!1
@occ21_2 = @occ21[km = encrypt(sign((spk(skA[]),pk(skB[]),k_2),skA[]),pk(skB[])),!1
@occ10_1 = @occ10[pk2 = pk(skB[]),!1 = @sid_2]
RESULT inj-event(endBparam(x)) ==> inj-event(beginBparam(x)) is false.
```

Proverif - graphic representation

```
>.\proverif -graph dirimg direx\myDenningSacco.pv
```



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. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. Foundations and Trends® in Privacy and Security , Now publishers inc, 2016, 1 (1-2), pp.1 - 135.https://hal.inria.fr/hal-01423760/

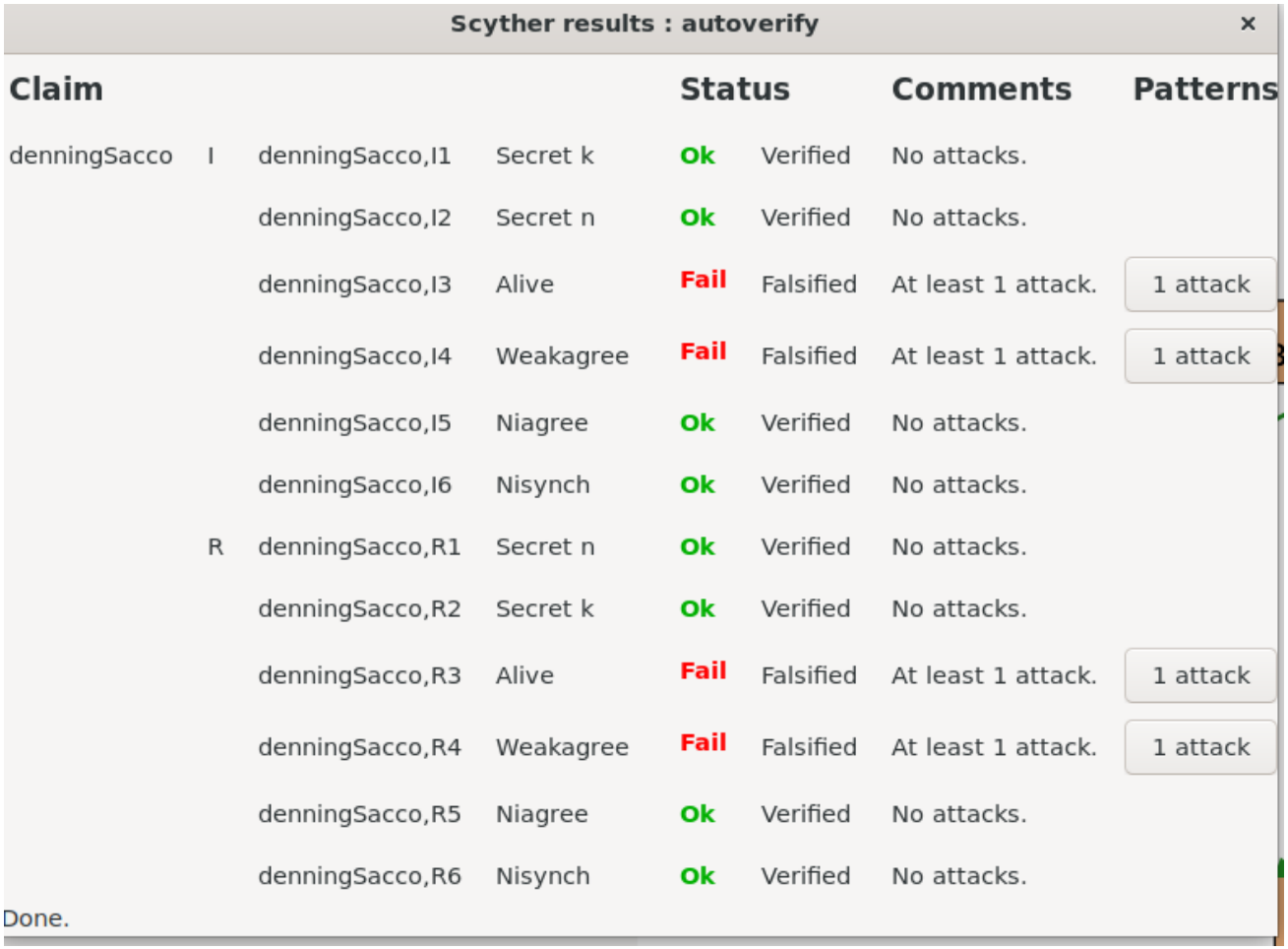
3 Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre, ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2023

4 Thomas Y. C. Woo, Simon S. Lam. A semantic model for authentication protocols. In Proceedings IEEE Symposium on Research in Security and Privacy, pages 178–194, Oakland, California, May 1993.

Denning-Sacco Scyther

```
usertype SessionKey;
protocol denningSacco(I,R)
{role I
    { var n: Nonce;
      fresh k: SessionKey;
      send_1(I,R,{{I,R, k}sk(I)}pk(R));
      recv_2(R,I, {n}k);}

    role R
    {   fresh n: Nonce;
        var k: SessionKey;
        recv_1(I,R, {{I,R, k}sk(I)}pk(R));
        send_2(R,I, {n}k);}}
```

Denning-Sacco Scyther

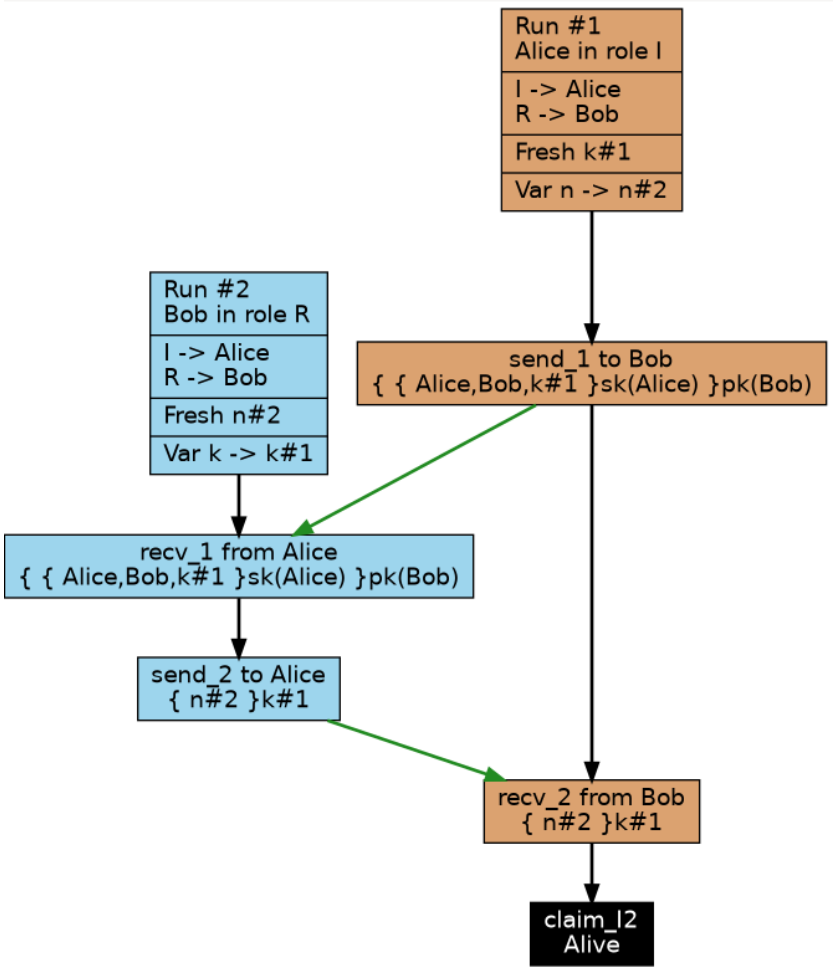


Scyther results : autoverify

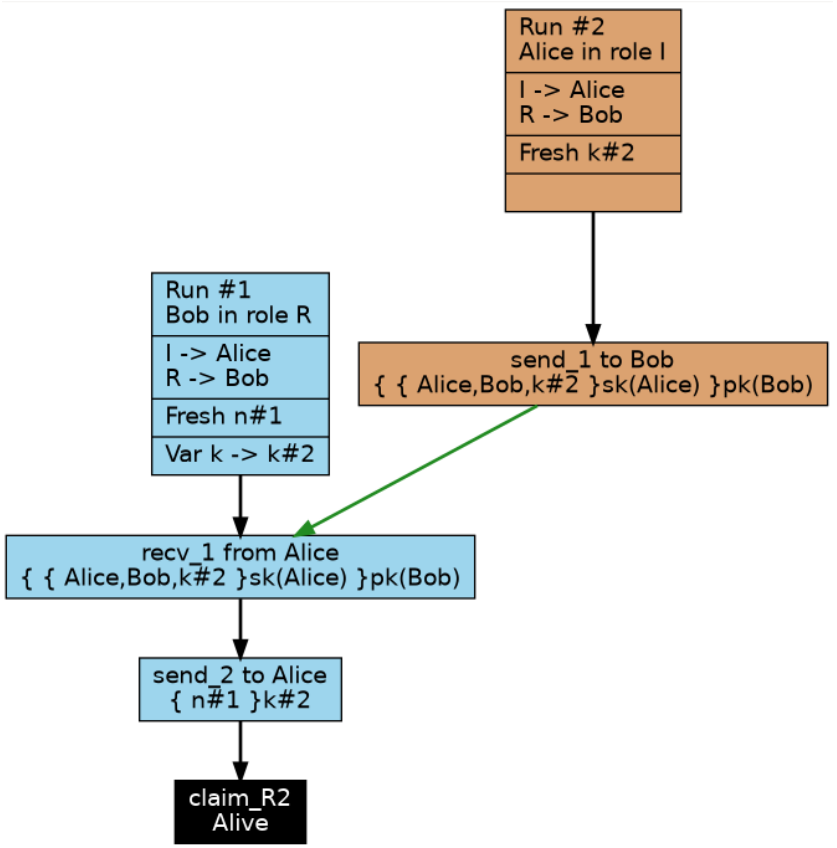
Claim				Status		Comments	Patterns
denningSacco	I	denningSacco,I1	Secret k	Ok	Verified	No attacks.	
		denningSacco,I2	Secret n	Ok	Verified	No attacks.	
		denningSacco,I3	Alive	Fail	Falsified	At least 1 attack.	1 attack
		denningSacco,I4	Weakagree	Fail	Falsified	At least 1 attack.	1 attack
		denningSacco,I5	Niagree	Ok	Verified	No attacks.	
		denningSacco,I6	Nisynch	Ok	Verified	No attacks.	
	R	denningSacco,R1	Secret n	Ok	Verified	No attacks.	
		denningSacco,R2	Secret k	Ok	Verified	No attacks.	
		denningSacco,R3	Alive	Fail	Falsified	At least 1 attack.	1 attack
		denningSacco,R4	Weakagree	Fail	Falsified	At least 1 attack.	1 attack
		denningSacco,R5	Niagree	Ok	Verified	No attacks.	
		denningSacco,R6	Nisynch	Ok	Verified	No attacks.	

Done.

Denning-Sacco Scyther - Alive



[Id 1] Protocol denningSacco, role I, claim type Alive



[Id 2] Protocol denningSacco, role R, claim type Alive