

# Special topics in Logic and Security I

Master Year II, Sem. I, 2025-2026

Ioana Leuştean  
FMI, UB

# Running example: the Denning-Sacco protocol

We consider the Denning-Sacco protocol:

$$(1) \quad A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$$

$$(2) \quad B \longrightarrow A : \{ secret \}_k$$

The protocol is vulnerable to the following attack

$$(att1) \quad A \longrightarrow E : \{ \{ k \}_{sk(A)} \}_{pk(E)}$$

$$(att2) \quad E \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$$

$$(att3) \quad B \longrightarrow E : \{ secret \}_k$$

Since  $E$  is a corrupted agent, by (att1) the attacker knows the key  $k$  so, by (att3), he knows the *secret* (which was meant to be a secret between  $B$  and  $A$ ).

Solution:

$$(1) \quad A \longrightarrow B : \{ \{ A, B, k \}_{sk(A)} \}_{pk(B)}$$

# The Denning-Sacco protocol

We consider the Denning-Sacco protocol (without timestamps):

$$(1) \quad A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$$

$$(2) \quad B \longrightarrow A : \{ secret \}_k$$

The communication between  $A$  and  $B$  can be represented (from the attacker's point of view) by:

$$\text{att}(\text{senc}(\text{secret}, K)) \text{ :- att}(\text{aenc}(\text{sign}(K, skA), pk(skB))).$$

If  $B$  receives what he expects, then he behaves following the protocols and the messages are "added" to the adversary knowledge.

Note that the agents are identified with their secret keys.

# The Denning-Sacco protocol

(1)  $A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$

(2)  $B \longrightarrow A : \{ secret \}_k$

We've defined the fact that  $B$  acts when he receives a message, but we still need to define that fact that  $A$  starts the protocol. To do this we can think that " $A$  wants to speak to any principal (who's public key is known)":

$$\text{att}(\text{aenc}(\text{sign}(k, skA), pk(X))) :- \text{att}(pk(X)).$$

Moreover, we can express the fact that  $k$  is a key that depends on the communication partner:

$$\text{att}(\text{aenc}(\text{sign}(k(pk(X)), skA), pk(X))) :- \text{att}(pk(X)).$$

## Prolog: the Denning-Sacco protocol - naive approach

In order to analyze the protocol we need to represent the attacker inference system, as well as the adversary initial knowledge:

```
% initial knowledge
```

```
att(skI).
```

```
att(pk(skA)).
```

```
att(pk(skB)).
```

```
% the attacker inference system
```

```
att(pk(X)) :- att(X).
```

```
att(X) :- att(aenc(X, pk(Y))), att(Y).
```

```
att(aenc(X, Y)) :- att(X), att(Y).
```

```
att(X) :- att(sign(X, Y)).
```

```
att(sign(X,Y)) :- att(X), att(Y).
```

```
att(X) :- att(senc(X, Y)), att(Y).
```

```
att(senc(X, Y)) :- att(X), att(Y).
```

```
%the protocol
```

```
att(aenc(sign(K, skA), pk(X))) :- att(pk(X)).
```

```
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).
```

## Prolog: the Denning-Sacco protocol - naive approach

```
att(skI).
att(pk(skA)).
att(pk(skB)).
att(aenc(sign(K, skA), pk(X))) :- att(pk(X)).
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).
att(pk(X)) :- att(X).
att(X) :- att(senc(X, Y)), att(Y).
att(aenc(X, Y)) :- att(X), att(Y).
att(sign(X,Y)) :- att(X), att(Y).
att(X) :- att(aenc(X, pk(Y))), att(Y).
att(X) :- att(sign(X, Y)).
att(senc(X, Y)) :- att(X), att(Y).
```

```
?- att(secret).
true
```

# Prolog: the Denning-Sacco protocol - naive approach

Why "naive"?

- The clause ordering affects the results or it can provide false attacks.
- The clause ordering affects termination.
- The rules with the conclusion  $\text{att}(X)$  can always be applied.

However, the idea of using Horn clauses for modeling and analyzing protocols is valuable, and few tools and theories are known.

"The ProVerif tool takes protocols written in a variant of **the applied pi calculus** as input together with a security property to verify. The protocol is then **automatically translated into a set of first-order Horn clauses** and the properties are translated into derivability queries. The verification algorithm is based on a **dedicated Horn clause resolution** procedure."

[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>

<https://bblanche.gitlabpages.inria.fr/proverif/>

# References

- 0 <https://bblanche.gitlabpages.inria.fr/proverif/>
- 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. Using Horn Clauses for Analyzing Security Protocols. In Véronique Cortier and Steve Kremer, editors, Formal Models and Techniques for Analyzing Security Protocols, volume 5 of Cryptology and Information Security Series, pages 86-111. IOS Press, March 2011.
- 4 Abadi, M., B. Blanchet, and C. Fournet. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. Journal of the ACM Vol. 65Issue 1February 2018 Article No.: 1pp 1–41. Available at <http://arxiv.org/abs/1609.03003v1>.
- 5 B. Blanchet, V. Cheval and V. Cortier, "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More," 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2022, pp. 69-86, doi: 10.1109/SP46214.2022.9833653.

# ProVerif: protocols as Horn clauses

Let  $\mathcal{F}$  be a *signature* (a set of function symbols together with their arity),  $\mathcal{N}$  a set of *names* (representing nonces, keys, identities) and  $\mathcal{X}$  a set of variables.

- The *terms* are defined by:

$$\begin{array}{lll} \text{Term} & ::= & x, y, z \quad \text{variables from } \mathcal{X} \\ & & a[t_1, \dots, t_n] \quad \text{names from } \mathcal{N} \\ & & f(t_1, \dots, t_n) \quad \text{function application for } f \text{ from } \mathcal{F} \end{array}$$

- The *predicate* is  $\text{att}(t)$ , which is *true* if the attacker knows the message modelled by  $t$ .
- The *Horn clauses* are:
  - facts:  $\text{att}(t)$
  - rules:  $\text{att}(t_1) \wedge \dots \wedge \text{att}(t_n) \rightarrow \text{att}(t)$

# ProVerif: the attacker

$$\begin{aligned}att(x) \wedge att(y) &\rightarrow att(senc(x, y)) \\att(senc(x, y)) \wedge att(y) &\rightarrow att(x)\end{aligned}$$
$$\begin{aligned}att(x) &\rightarrow att(pk(x)) \\att(x) \wedge att(y) &\rightarrow att(aenc(x, y)) \\att(aenc(x, pk(y))) \wedge att(y) &\rightarrow att(x)\end{aligned}$$
$$\begin{aligned}att(x) \wedge att(y) &\rightarrow att(sign(x, y)) \\att(sign(x, y)) &\rightarrow att(x)\end{aligned}$$
$$\begin{aligned}att(x) \wedge att(y) &\rightarrow att((x, y)) \\att(x, y) &\rightarrow att(x) \\att(x, y) &\rightarrow att(y)\end{aligned}$$

$att(t)$  for any term  $t$  in the initial knowledge of the attacker  
 $att(a[])$  for at least one name  $a$

# ProVerif: Horn clauses for the Denning-Sacco protocol

proverif2.05

<http://proverif.inria.fr>

(\* Initialization \*)

c:c[];

c:pk(sA[]);

c:pk(sB[]);

(\* The attacker \*)

c:x & c:encrypt(m,pk(x)) -> c:m;

c:x & c:sencrypt(m,x) -> c:m;

c:x -> c:pk(x);

c:x & c:y -> c:encrypt(x,y);

c:x & c:y -> c:sencrypt(x,y);

c:sign(x,y) -> c:x;

c:x & c:y -> c:sign(x,y);

(\* The protocol \*)

(\* A \*)

c:pk(x) -> c:encrypt(sign(k[pk(x)], sA[]), pk(x));

(\* B \*)

c:encrypt(sign(k, sA[]), pk(sB[])) -> c:sencrypt(secret[], k).

# ProVerif: Horn clauses for the Denning-Sacco protocol

proverif2.05

<http://proverif.inria.fr>

Clause 13:  $c:c[]$

Clause 12:  $c:pk(sA[])$

Clause 11:  $c:pk(sB[])$

Clause 10:  $c:x \ \& \ c:encrypt(m, pk(x)) \rightarrow c:m$

Clause 9:  $c:x \ \& \ c:sencrypt(m, x) \rightarrow c:m$

Clause 8:  $c:x \rightarrow c:pk(x)$

Clause 7:  $c:x \ \& \ c:y \rightarrow c:encrypt(x, y)$

Clause 6:  $c:x \ \& \ c:y \rightarrow c:sencrypt(x, y)$

Clause 5:  $c:sign(x, y) \rightarrow c:x$

Clause 4:  $c:x \ \& \ c:y \rightarrow c:sign(x, y)$

Clause 3:  $c:pk(x) \rightarrow c:encrypt(sign(k[pk(x)], sA[]), pk(x))$

Clause 2:  $c:encrypt(sign(k_1, sA[]), pk(sB[])) \rightarrow$   
 $c:sencrypt(secret[], k_1)$

Clause 1:  $c:new\text{-}name[!att = v]$

goal reachable:  $c:secret[]$

# ProVerif: Horn clauses for the Denning-Sacco protocol

proverif2.05

<http://proverif.inria.fr>

Derivation:

Abbreviations:

$k_1 = k[\text{pk}(x)]$

clause 9 c:secret[]

    clause 5 c:k<sub>1</sub>

        duplicate c:sign(k<sub>1</sub>,sA[])

    clause 2 c:sencrypt(secret[],k<sub>1</sub>)

        clause 7 c:encrypt(sign(k<sub>1</sub>,sA[]),pk(sB[]))

            clause 10 c:sign(k<sub>1</sub>,sA[])

                duplicate c:x

                    clause 3 c:encrypt(sign(k<sub>1</sub>,sA[]),pk(x))

                        clause 8 c:pk(x)

                            any c:x

        clause 11 c:pk(sB[])

**RESULT goal reachable: c:secret[]**

duplicate - demonstratia corespunzatoare se regaseste mai jos

# ProVerif: Horn clauses for the Denning-Sacco corrected

proverif2.05

<http://proverif.inria.fr>

(\* Initialization \*)

c:c[];

c:pk(sA[]);

c:pk(sB[]);

(\* The attacker \*)

.

.

.

(\* The protocol \*)

(\* A \*)

c:pk(x) -> c:encrypt(sign((pk(sA[]),pk(x),k[pk(x)]), sA[]), pk(x));

(\* B \*)

c:encrypt(sign((pk(sA[]),pk(sB[]),k), sA[]), pk(sB[])) ->  
c:sencrypt(secret[], k).

# ProVerif: Horn clauses for the Denning-Sacco protocol

proverif2.05

<http://proverif.inria.fr>

Clause 17:  $c:(v, v_1, v_2) \rightarrow c:v_2$

Clause 16:  $c:(v, v_1, v_2) \rightarrow c:v_1$

Clause 15:  $c:(v, v_1, v_2) \rightarrow c:v$

Clause 14:  $c:v \ \& \ c:v_1 \ \& \ c:v_2 \rightarrow c:(v, v_1, v_2)$

Clause 13:  $c:c[]$

Clause 12:  $c:pk(sA[])$

Clause 11:  $c:pk(sB[])$

Clause 10:  $c:x \ \& \ c:encrypt(m, pk(x)) \rightarrow c:m$

Clause 9:  $c:x \ \& \ c:sencrypt(m, x) \rightarrow c:m$

Clause 8:  $c:x \rightarrow c:pk(x)$

Clause 7:  $c:x \ \& \ c:y \rightarrow c:encrypt(x, y)$

Clause 6:  $c:x \ \& \ c:y \rightarrow c:sencrypt(x, y)$

Clause 5:  $c:sign(x, y) \rightarrow c:x$

Clause 4:  $c:x \ \& \ c:y \rightarrow c:sign(x, y)$

Clause 3:  $c:pk(x) \rightarrow c:encrypt(sign((pk(sA[]), pk(x), k[pk(x)]), sA[]), pk(x))$

Clause 2:  $c:encrypt(sign((pk(sA[]), pk(sB[]), k_1), sA[]), pk(sB[])) \rightarrow$   
 $c:sencrypt(secret[], k_1)$

Clause 1:  $c:new\text{-}name[!att = v]$

RESULT goal unreachable:  $c:secret[]$

- The protocols are specified in a variant of  $\pi$ - **calculus**.
- The description of the protocol is automatically translated in a set of **Horn clauses**.
- The security property is checked by performing a **particular version of resolution**, developed for this particular setting.

# A $\pi$ -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 $\mathcal{N}$
		$f(t_1, \dots, t_n)$	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
		$P Q$	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.

## A $\pi$ -calculus for protocols: example

For the Horn clause representation we've used the signature  $\mathcal{F} = \{pk, aenc, senc, sign, pair\}$ , which is a signature of constructors.

We can express the first exchange in NSPK

$A \longrightarrow B : \{\{ A, na \}\}_{pk(B)}$  by

$\nu n.out(c, aenc(pair(skA, n), pk(skB)))$

where  $n, skA$  and  $skB, c \in \mathcal{N}$  and  $c$  is a name for channels.

The second exchange in NSPK is:

$B \longrightarrow A : \{\{ na, nb \}\}_{pk(A)}$

and we note that  $B$  has to obtain  $na$  from the initial message, which means to perform decryption and to extract the second component of the decrypted message.

In order to define  $B$ 's actions, we add *destructors*, as well as equations for constructors and destructors.

# Equational theory

- for asymmetric encryption the destructor is *adec* and the equation 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.

# A $\pi$ -calculus for protocols: remarks

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

- Names and variables have scopes, defined by restrictions, inputs and local definitions and one can define the free (bound) names and variables.
- The expressions defining processes are equal modulo renaming of bound names and variables.
- The formal semantics is defined by *reduction*:

$$\begin{aligned} out(u, t).P|in(u, x).Q &\rightarrow P|Q\{t/x\} \\ if\ t = t\ then\ P\ else\ Q &\rightarrow P \\ if\ t_1 = t_2\ then\ P\ else\ Q &\rightarrow Q \end{aligned}$$

where  $\{t/x\}$  is the substitution that replaces the free occurrences of  $x$  with  $t$  and  $t_1 = t_2$  is not provable in the corresponding equational theory.

We refer to [4] for details.

# A $\pi$ -calculus for protocols: NSPK

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

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

the process corresponding to the initiator can be defined by:

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

# The Denning-Sacco protocol revisited

$$A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$$
$$B \longrightarrow A : \{ secret \}_k$$

In order to model our protocol we define

- a *process for each role* and
- a *process that brings them together*.

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

# The Denning-Sacco protocol

$A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$

$B \longrightarrow A : \{ secret \}_k$

- a process for each role:

$$\begin{aligned} P_A(skA, pkB) = & \nu k.out(c, aenc(sign(k, skA), pkB). \\ & in(c, x). \\ & let\ z = sdec(x, k)\ in\ 0 \end{aligned}$$
$$\begin{aligned} P_B(skB, pkA) = & in(c, y). \\ & let\ z = adec(y, skB)\ in \\ & if\ check(z, pkA) = xk\ then \\ & \nu s.out(c, senc(s, xk)) \end{aligned}$$

- destructors for encryption:

- for asymmetric encryption the destructor is *adec* and the equation 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  $check(sign(x, y), pk(y)) = x$

# The Denning-Sacco protocol

$$A \longrightarrow B : \{ \{ k \}_{sk(A)} \}_{pk(B)}$$
$$B \longrightarrow A : \{ secret \}_k$$

- a process for each role:

$$\begin{aligned} P_A(skA, pkB) = & \nu k. out(c, aenc(sign(k, skA), pkB)). \\ & in(c, x). \\ & let\ z = sdec(x, k)\ in\ 0 \end{aligned}$$
$$\begin{aligned} P_B(skB, pkA) = & in(c, y). \\ & let\ z = adec(y, skB)\ in \\ & if\ check(z, pkA) = xk\ then \\ & \nu s. out(c, senc(s, xk)) \end{aligned}$$

- 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 continued

We can improve the representation of the protocol by:

- allowing multiple parallel executions for each role:

$$P_A(skA, pkB) = \textcolor{blue}{!} \nu k. \text{out}(c, \text{aenc}(\text{sign}(k, skA), pkB)).$$
$$\text{in}(c, x).$$
$$\text{let } z = \text{sdec}(x, k) \text{ in } 0$$
$$P_B(skB, pkA) = \textcolor{blue}{!} \text{in}(c, y).$$
$$\text{let } z = \text{adec}(y, skB) \text{ in}$$
$$\text{if } \text{check}(z, pkA) = xk \text{ then}$$
$$\nu s. \text{out}(c, \text{senc}(s, xk))$$

- making the public keys available to a potential attacker:

$$P_{DS} = \nu skA. \nu skB.$$
$$\text{let } pkA = pk(skA) \text{ in}$$
$$\text{let } pkB = pk(skB) \text{ in}$$
$$\textcolor{blue}{\text{out}(c, pkA). \text{out}(c, pkB).}$$
$$P_A(skA, pkB) \mid P_B(skB, pkA)$$

# The Denning-Sacco protocol continued

In the above representation, the (honest) participants talk only to each other. In order to allow the attacker to interfere we can modify the initiator process such that the public key of the communication partner is received through the channel  $c$ :

$$\begin{aligned} P_A(skA) = & \text{! } in(c, x_{pk}). \\ & \nu k. out(c, aenc(sign(k, skA), x_{pk}). \\ & in(c, x). \\ & \text{let } z = sdec(x, k) \text{ in } 0 \end{aligned}$$
$$\begin{aligned} P_B(skB, pkA) = & \text{! } in(c, y). \\ & \text{let } z = adec(y, skB) \text{ in} \\ & \text{if } check(z, pkA) = xk \text{ then} \\ & \nu s. out(c, senc(s, xk)) \end{aligned}$$
$$\begin{aligned} P_{DS} = & \nu skA. \nu skB. \\ & \text{let } pkA = pk(skA) \text{ in} \\ & \text{let } pkB = pk(skB) \text{ in} \\ & out(c, pkA). out(c, pkB). \\ & P_A(skA, pkB) \mid P_B(skB, pkA) \end{aligned}$$

# ProVerif: $\pi$ -calculus for the Denning-Sacco protocol

proverif2.05/examples/pi

```
free c.
```

```
fun pk/1.
```

```
fun encrypt/2.
```

```
reduc decrypt(encrypt(x,pk(y)),y) = x.
```

```
fun sign/2.
```

```
reduc getmess(sign(m,k)) = m.
```

```
reduc checksign(sign(m,k), pk(k)) = m.
```

```
fun sencrypt/2.
```

```
reduc sdecrypt(sencrypt(x,y),y) = x.
```

```
(* secrecy assumptions *)
```

```
not attacker:skA.
```

```
not attacker:skB.
```

```
private free secretB.
```

```
query attacker:secretB.
```

- free names are known to the attacker;
- free names that are not known to the attacker must be declared private;
- the constructors are defined with fun;
- the destructors are defined with reduction rules reduc; they are known to the attacker, unless they are defined private;
- attacker:k means that k is known to the attacker; the secrecy assumption means that one cannot prove that skA and skB are known to the attacker; the secrecy assumptions are checked by ProVerif
- query attacker:s means that ProVerif checks if a state in which s is known to the attacker is reachable, i.e. it checks if not attacker:s is false.

# ProVerif: $\pi$ -calculus for the Denning-Sacco protocol

proverif2.05/examples/pi

```
let processA =  
  in(c, pk2);  
  new k; new r;  
  out(c, (encrypt(sign(k, skA), pk2)));  
  in(c, m);  
  let s = sdecrypt(m,k) in 0.  
  
let processB =  
  in(c, km);  
  let ks = decrypt(km,skB) in  
  let k = checksign(ks, pkA) in  
  out(c, sencrypt(secretB, k)).
```

# ProVerif: $\pi$ -calculus for the Denning-Sacco protocol

```
proverif2.05/examples/pi

let processA =

let processB =

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

>.\proverif direx\dser.pi
:
A trace has been found.
RESULT not attacker:secretB[] is false.
```

# ProVerif: the output

Process 0 (that is, the initial process):

```
{1}new skA;
{2}let pkA = pk(skA) in
{3}out(c, pkA);
{4}new skB;
{5}let pkB = pk(skB) in
{6}out(c, pkB);
(
  {7}!
  {8}in(c, pk2);
  {9}new k;
  {10}new r;
  {11}out(c, (encrypt(sign(k,skA),pk2)));
  {12}in(c, m);
  {13}let s = sdecrypt(m,k) in
  0
) | (
  {14}!
  {15}in(c, km);
  {16}let ks = decrypt(km,skB) in
  {17}let k_1 = checksign(ks,pkA) in
  {18}out(c, sencrypt(secretB,k_1))
)
```

```
let processA =
let processB =

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

>.\proverif direx\dser.pi
:
:
A trace has been found.
RESULT not attacker:secretB[] is
false.
```

# ProVerif: the output

Process 0 (that is, the initial process):

```
{1}new skA;  
{2}let pkA = pk(skA) in  
{3}out(c, pkA);  
{4}new skB;  
{5}let pkB = pk(skB) in  
{6}out(c, pkB);  
(  
  {7}!  
  {8}in(c, pk2);  
  {9}new k;  
  {10}new r;  
  {11}out(c, (encrypt(sign(k,skA),pk2)));  
  {12}in(c, m);  
  {13}let s = sdecrypt(m,k) in  
    0  
) | (  
  {14}!  
  {15}in(c, km);  
  {16}let ks = decrypt(km,skB) in  
  {17}let k_1 = checksign(ks,pkA) in  
  {18}out(c, sencrypt(secretB,k_1))  
)
```

-- Query not attacker:secretB[] in process 0.

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 not attacker:secretB[]

goal reachable: attacker:secretB[]

Derivation:

Abbreviations:

k\_2 = k[pk2 = pk(y), !1 = @sid]

1. The attacker has some term y.  
attacker:y.

2. By 1, the attacker may know y.

Using the function pk the attacker may obtain  
attacker:pk(y).

# ProVerif: the output

Derivation:

Abbreviations:

`k_2 = k[pk2 = pk(y), !1 = sid]`

1. The attacker has some term `y`.

`attacker:y`.

2. By 1, the attacker may know `y`.

Using the function `pk` the attacker may obtain `pk(y)`.

`attacker:pk(y)`.

3. The message `pk(y)` that the attacker may have by 2 may be received at input 8.

So the message `(encrypt(sign(k_2, skA[]), pk(y)))` may be sent to the attacker at output 11.

`attacker:(encrypt(sign(k_2, skA[]), pk(y)))`.

4. By 3, the attacker may know `(encrypt(sign(k_2, skA[]), pk(y)))`.

Using the function `1-proj-1-tuple` the attacker may obtain `encrypt(sign(k_2, skA[]), pk(y))`.

`attacker:encrypt(sign(k_2, skA[]), pk(y))`.

5. By 4, the attacker may know `encrypt(sign(k_2, skA[]), pk(y))`.

By 1, the attacker may know `y`.

Using the function `decrypt` the attacker may obtain `sign(k_2, skA[])`.

`attacker:sign(k_2, skA[])`.

6. By 5, the attacker may know `sign(k_2, skA[])`.

Using the function `getmess` the attacker may obtain `k_2`.

`attacker:k_2`.

# ProVerif: the output

7. The message `pk(skB[])` may be sent to the attacker at output 6.

`attacker:pk(skB[]).`

8. By 5, the attacker may know `sign(k_2,skA[])`.

By 7, the attacker may know `pk(skB[])`.

Using the function `encrypt` the attacker may obtain `encrypt(sign(k_2,skA[]),pk(skB[]))`.

`attacker:encrypt(sign(k_2,skA[]),pk(skB[])).`

9. The message `encrypt(sign(k_2,skA[]),pk(skB[]))` that the attacker may have by 8 may be received.

So the message `sencrypt(secretB[],k_2)` may be sent to the attacker at output 18.

`attacker:sencrypt(secretB[],k_2).`

10. By 9, the attacker may know `sencrypt(secretB[],k_2)`.

By 6, the attacker may know `k_2`.

Using the function `sdecrypt` the attacker may obtain `secretB[]`.

`attacker:secretB[].`

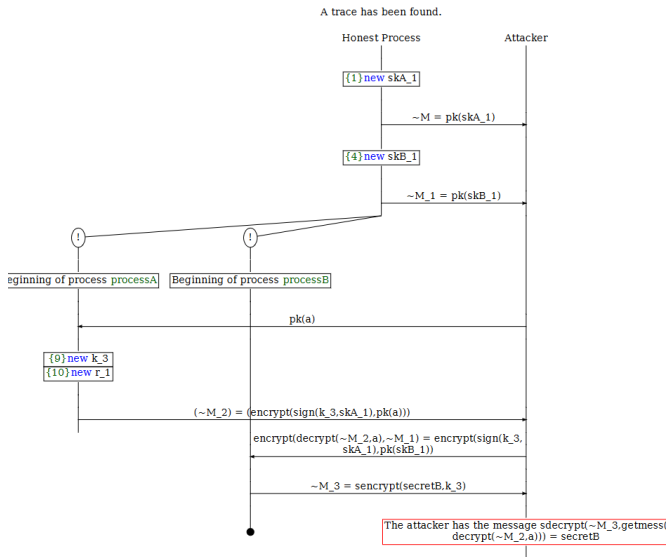
11. By 10, `attacker:secretB[]`.

The goal is reached, represented in the following fact:

`attacker:secretB[].`

# Proverif - graphic representation

```
>.\proverif -graph dirimg direx\dser.pi
```



# ProVerif: the Denning-Sacco protocol corrected

```
proverif2.05/examples/pi
```

```
let processA =  
  in(c, pk2);  
  new k; new r;  
  out(c, (encrypt(sign((pkA,pk2,k), skA), pk2)));  
  in(c, m);  
  let s = sdecrypt(m,k) in 0.
```

```
let processB =  
  in(c, km);  
  let ks = decrypt(km,skB) in  
  let (=pkA,=pkB, k) = checksign(ks, pkA) in  
  out(c, sencrypt(secretB, k)).
```

Note that  $(=pkA,=pkB, k) = \text{checksign}(ks, pkA)$  checks if  $ks$  is a signature under  $skA$  containing a triple, where the first two elements are public keys.

# ProVerif: the Denning-Sacco protocol corrected

```
proverif2.05/examples/pi
```

```
let processA =  
  in(c, pk2);  
  new k; new r;  
  out(c, (encrypt(sign((pkA, pk2, k), skA), pk2)));  
  in(c, m);  
  let s = sdecrypt(m, k) in 0.
```

```
let processB =  
  in(c, km);  
  let ks = decrypt(km, skB) in  
  let (=pkA, =pkB, k) = checksign(ks, pkA) in  
  out(c, sencrypt(secretB, k)).
```

```
>.\proverif direx\ds.pi
```

```
⋮
```

```
RESULT not attacker:secretB[] is true.
```

# The structure of ProVerif

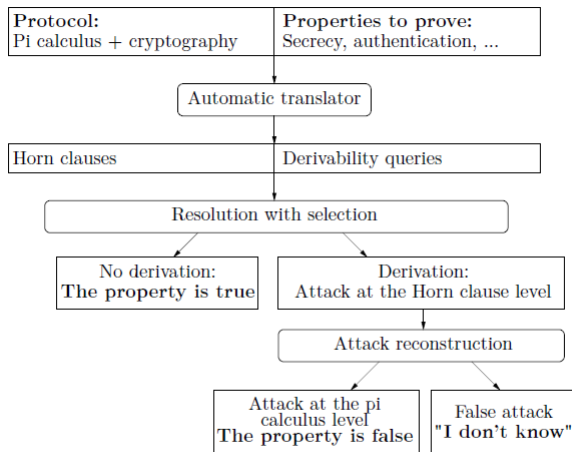


Figure 1.1: Structure of ProVerif

- 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.

# 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. Using Horn Clauses for Analyzing Security Protocols. In Véronique Cortier and Steve Kremer, editors, Formal Models and Techniques for Analyzing Security Protocols, volume 5 of Cryptology and Information Security Series, pages 86-111. IOS Press, March 2011.
- 4 Abadi, M., B. Blanchet, and C. Fournet. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. Journal of the ACM Vol. 65Issue 1February 2018 Article No.: 1pp 1–41. Available at <http://arxiv.org/abs/1609.03003v1>.
- 5 B. Blanchet, V. Cheval and V. Cortier, "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More," 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2022, pp. 69-86, doi: 10.1109/SP46214.2022.9833653.