KEY EXCHANGE & MORE IN PROVERIF

CAPS 2025
Workshop on Computer-Aided Proofs of Security

Vincent Cheval
University of Oxford
vincent.cheval@cs.ox.ac.uk

Madrid

04/05/2025

Symbolic (Dolev-Yao) models

The attacker can…

- Read / Write
- Intercept

But they cannot…

- Break cryptography
- Use side channels

Created in the 80′ but we have come a long way!

Success stories (not exhaustif)

- TLS 1.3 with Encrypted Client Hello
- CHVote
- Swiss Post
- Wireguard
- 5G-AKA
- Signal
- ZCash
- Certificate Transparency
- Belenios
- Noise Framework
- EMV

Existing models

Computation model		Symbolic model
Real algorithms (or as close as it gets)	**Cryptographic primitives**	Function symbols (assumed “almost” perfect)
Bitstring	**Messages**	Terms
PPT	**Attacker**	Idealized
Difficult and by hand or with proof assistants	**Proofs**	« Easier » and mechanized
Strong	**Security guarantees**	Limited to the abstraction of the model

Symbolic terms

Nonces: $a, b, c, \ldots$

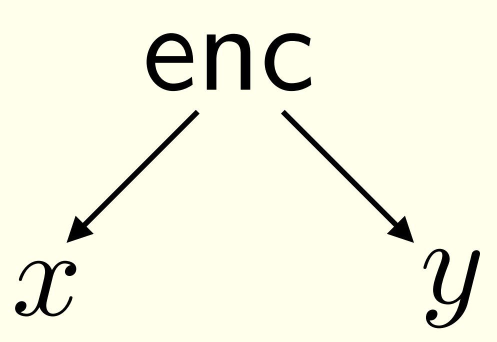
atomic elements (keys, random numbers, ...)

Variables: $x, y, z, \ldots$

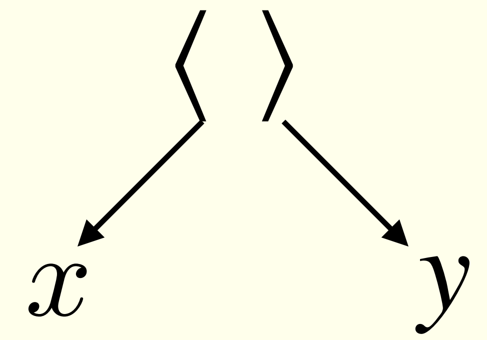
Functions symbols with their arity: enc/2, dec/2, $\oplus$/2, $\langle\ \rangle$/2, proj_1/1, proj_2/1,...

Abstract functions

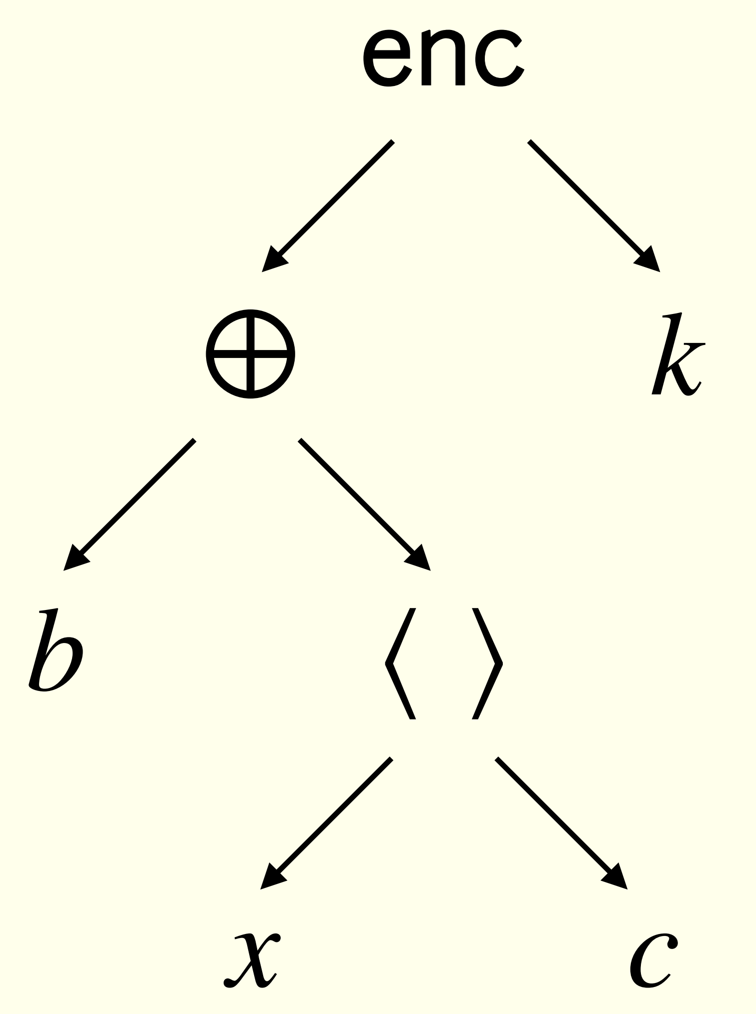
$\text{enc}(x, y)$



$\langle x, y \rangle$



$\text{enc}(b \oplus \langle x, c \rangle, k)$



MODELLING A PROTOCOL AND ITS SECURITY PROPERTIES IN PROVERIF

SignedDH

How do we translate an Alice-Bob description into something that we can analyse?

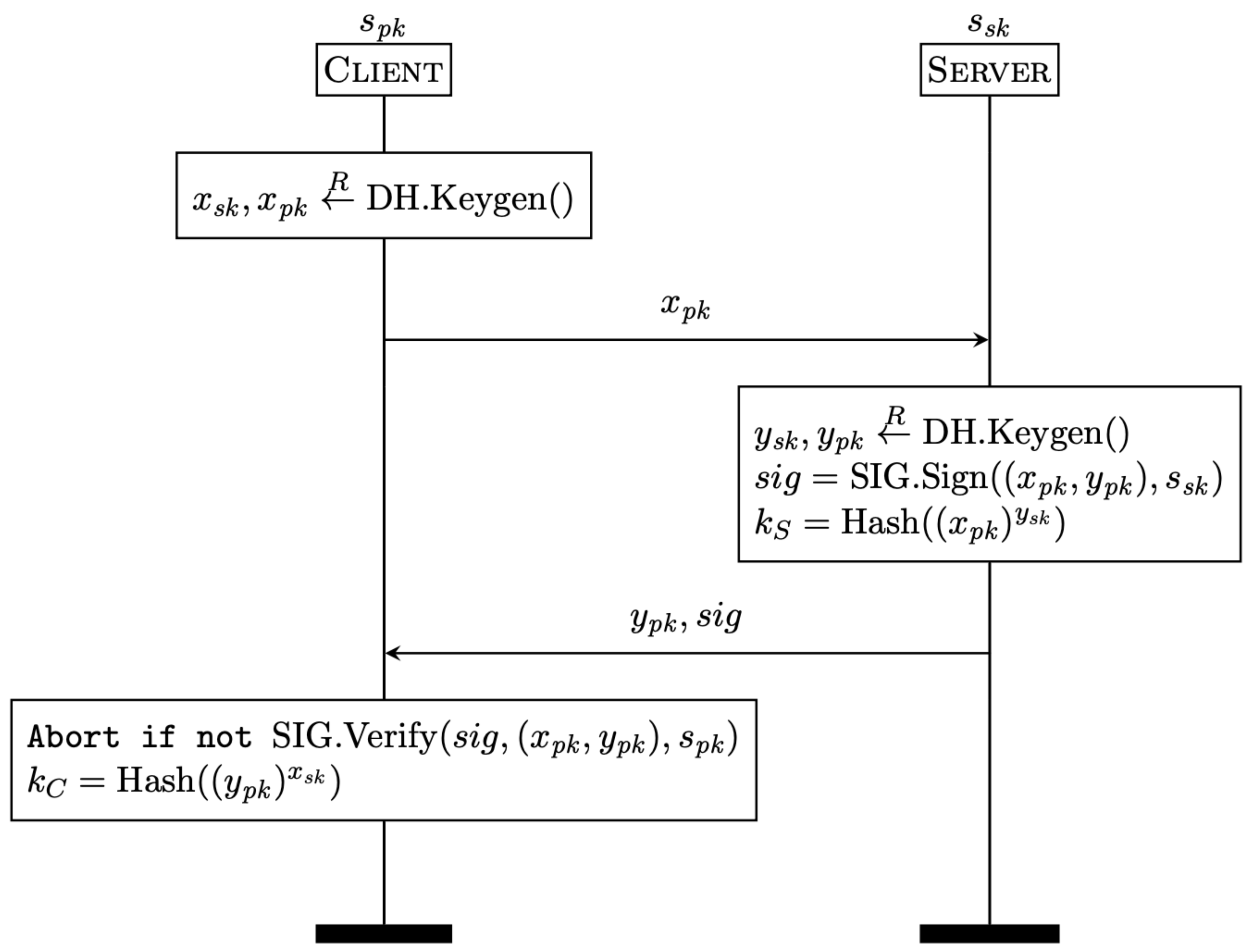


Figure 2: Signed Diffie-Hellman key exchange

ProVerif in a nutshell

ProVerif's input can be seen as a small typed programming language adapted to writing the programs executed by the different participants of the protocol.

often called processes

One process for the client

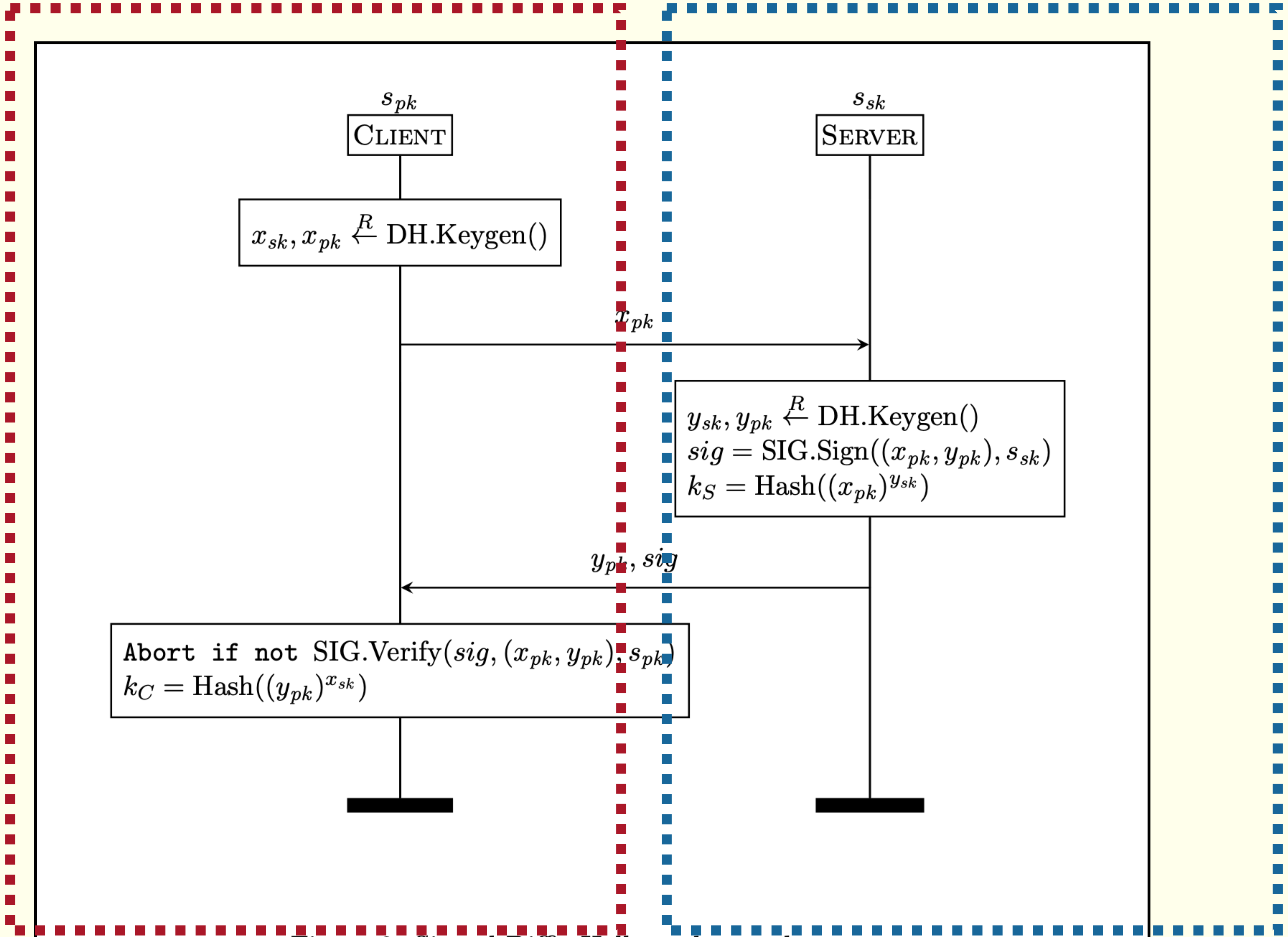


Figure 2: Signed Diffie-Hellman key exchange

One process for the server

ProVerif in a nutshell

What can we do in a process?

Generate random nonces

```
new k:bitstring;
...
```

Value assignment

```
let sig = SIG_sign((x_pk,y_pk),s_sk) in
...
```

Test on terms

```
if SIG_verify(sig,(x_pk,y_pk),s_pk) = true
then
  ...
else
  ...
```

ProVerif in a nutshell

What can we do in a process?

Sending over a channel

```
out(c, s_pk);
...
```

Receiving over a channel

```
in(c, (y_pk:G,sig:bitstring));
...
```

Raising events

used to described security properties

```
event ServerAccept(s_pk,x_pk,y_pk,k_S);
...
```

Declaring cryptographic primitives

Digital signature as described in the specification

- SIG.Gen : $\emptyset \rightarrow_\$ \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_\$ \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0,1\}$ Signature verification

$$\Pr[\text{SIG.Verify}(pk, m, \text{SIG.sign}(sk, m)) = 1] = 1$$

We need to link pk and sk

ProVerif has a simple type system

```
type SK. (* Types for secret signing keys *)
type PK. (* Types for public verification keys *)
type S.  (* Types for signature *)
```

Declaration of functions

```
fun SIG_pk(SK):PK.
fun SIG_sign(bitstring,SK):S.
```

Declaring cryptographic primitives

Digital signature as described in the specification

- SIG.Gen : $\emptyset \rightarrow_{\$} \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_{\$} \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0,1\}$ Signature verification

$$\Pr[\text{SIG.Verify}(pk, m, \text{SIG.sign}(sk, m)) = 1] = 1$$

Writing the algebraic property with reduction (rewrite) rules.

```
fun SIG_verify(S,bitstring,PK):bool
reduc
  forall m:bitstring, sk:SK; SIG_verify(SIG_sign(m,sk),m,SIG_pk(sk)) = true.
```

Declaring cryptographic primitives

Digital signature as described in the specification

- SIG.Gen : $\emptyset \to_{\$} \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \to_{\$} \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \to \{0,1\}$ Signature verification

$$\Pr[\text{SIG.Verify}(pk, m, \text{SIG.sign}(sk, m)) = 1] = 1$$

Writing the algebraic property with reduction (rewrite) rules.

```
fun SIG_verify(S,bitstring,PK):bool
reduc
  forall m:bitstring, sk:SK; SIG_verify(SIG_sign(m,sk),m,SIG_pk(sk)) = true.
```

A bit more precise…

```
fun SIG_verify(S,bitstring,PK):bool
reduc
  forall m:bitstring, sk:SK; SIG_verify(SIG_sign(sk,m),SIG_pk(sk)) = true
  otherwise forall pk:PK, m:bitstring, sig:S; SIG_verify(sig,pk,m) = false.
```

Declaring cryptographic primitives

Digital signature as described in the specification

- SIG.Gen : $\emptyset \rightarrow_{\$} \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_{\$} \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0,1\}$ Signature verification

What about SIG.Gen?

In the process

```
new sk:SK;
let pk = SIG_pk(sk) in
out(c,pk);
...
```

can become cumbersome and less *close* to the specification

Instead declare a macro function

```
letfun SIG_gen() = new sk:SK; (sk,SIG_pk(sk)).
```

Declaring cryptographic primitives

Digital signature as described in the specification

- SIG.Gen : $\emptyset \rightarrow_{\$} \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_{\$} \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0,1\}$ Signature verification

What about SIG.Gen?

In the process

```
let (sk:SK,pk:PK) = SIG_gen() in
out(c,pk);
...
```

As in the specification

Instead declare a macro function

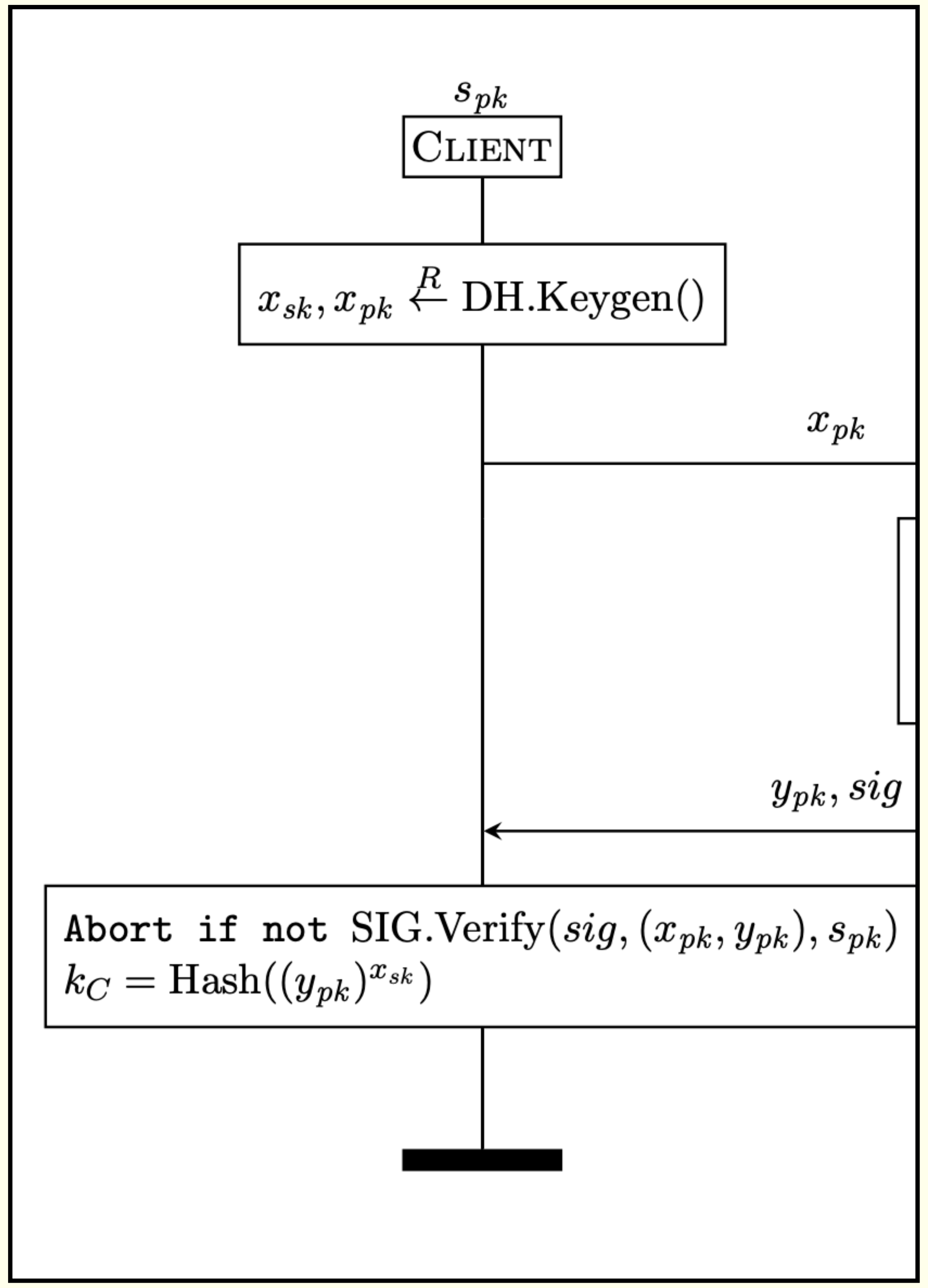
```
letfun SIG_gen() = new sk:SK; (sk,SIG_pk(sk)).
```

Declaring cryptographic primitives

```
(* Digital signature *)

type SK. (* Types for secret signing keys *)
type PK. (* Types for public verification keys *)
type S.  (* Types for signature *)

fun SIG_pk(SK):PK.
fun SIG_sig(bitstring,SK):S.

fun SIG_verify(S,bitstring,PK):bool
reduc
  forall m:bitstring, sk:SK; SIG_verify(SIG_sig(m,sk),m,SIG_pk(sk)) = true
  otherwise forall pk:PK, m:bitstring, sig:S; SIG_verify(sig,m,pk) = false.

letfun SIG_gen() = new sk:SK; (sk,SIG_pk(sk)).

(* Hash function *)

fun Hash(bitstring):bitstring.

(* Diffie-Hellman *)

type G. (* Type for the group *)
type Z. (* Type for exponent *)

const g: G.
fun exp(G, Z): G.
equation forall x: Z, y: Z; exp(exp(g, x), y) = exp(exp(g, y), x).

letfun DH_keygen() = new a:Z; (a, exp(g,a)).
```

SignedDH

The client

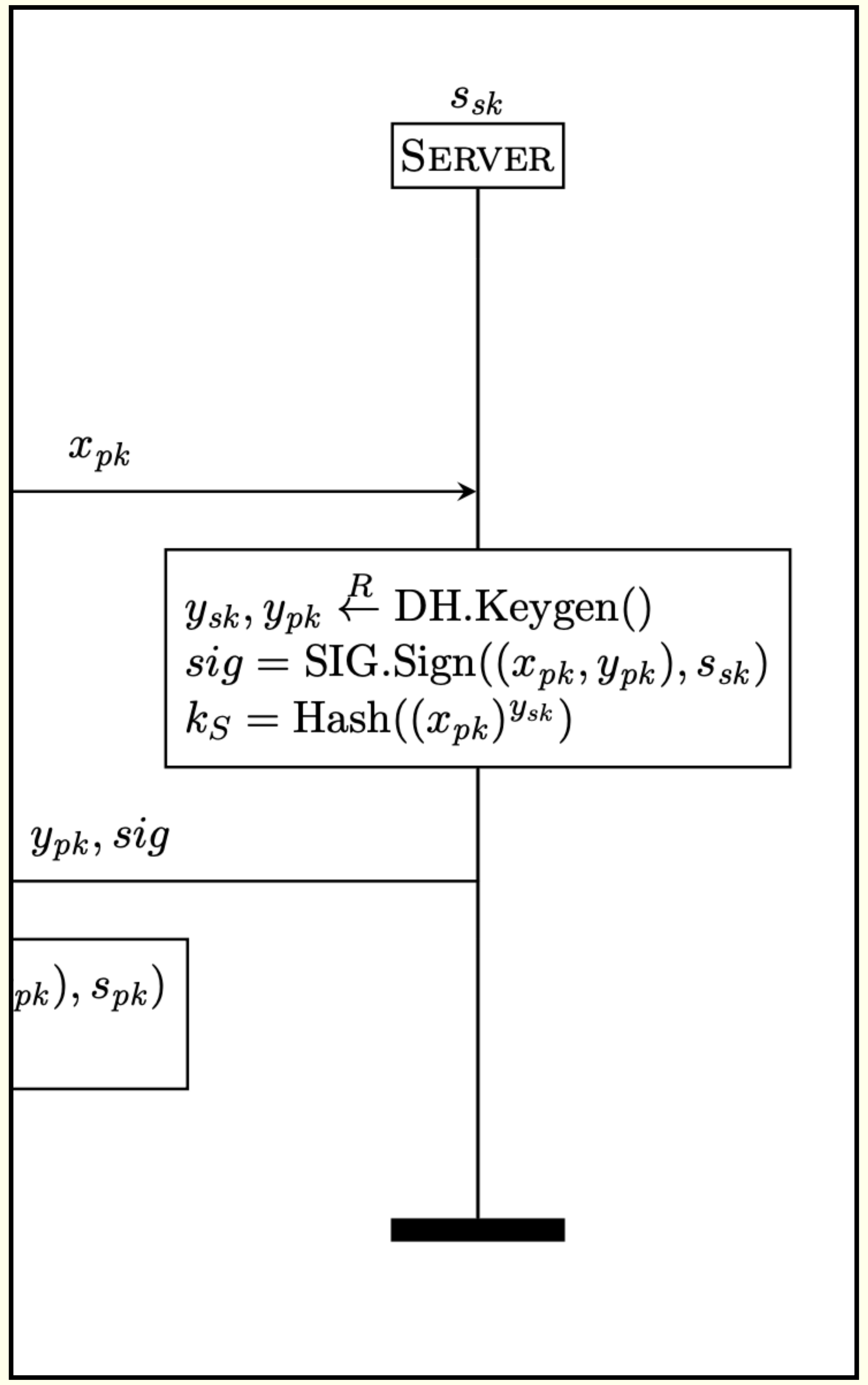



```
let Client(s_pk:PK) =

  (* Send first message *)
  let (x_sk:Z,x_pk:G) = DH_keygen() in
  out(c, x_pk);

  (* Receiving second message *)
  in(c,(y_pk:G,sig:S));
  if SIG_verify(sig, (x_pk,y_pk), s_pk) then
  let k_C = Hash(exp(y_pk,x_sk)) in
  0
.
```

SignedDH

The server

```
let Server(s_sk:SK) =

  (* Receiving first message *)
  in(c, x_pk:G);

  let (y_sk:Z,y_pk:G) = DH_keygen() in
  let sig = SIG_sign((x_pk,y_pk),s_sk) in
  let k_S = Hash(exp(x_pk,y_sk)) in

  (* Sending second message *)
  out(c,(y_pk,sig));
  0
.
```



The scenario / the system under study.

Concurrent processes

```
P | Q
```

Unbounded copies

```
! P
```

```
process
  (
    !
    (* Initialize a new server  *)
    let (s_sk:SK,s_pk:PK) = SIG_gen() in
    (* Make public the public key *)
    out(c,s_pk);

    !
    (* Run multiple session of the server *)
    Server(s_sk,s_pk)
  ) | (
    !
    (* Run multiple session of the client *)
    in(c,s_pk:PK); (* We let the attacker choose the public key *)
    Client(s_pk)
  )
```

Security properties

Sanity checks

Checking that the model is executable

Injective Client-side authentication

Every time an honest client completes a session thinking that it shares a key k with an honest server with public s_{pk}, then that server must have completed a distinct session of the protocol with that client and also obtained the shared key k

Forward secrecy

The attacker cannot learn the session key, even if they compromise the server in the future.

How to express security properties

Raising events

```
event HonestClientShare(G).
event HonestSever(PK).

event ServerAccept(PK,G,G,bitstring).
event ClientAccept(PK,G,G,bitstring).
```

```
let Client(s_pk:PK) =

  (* Send first message *)
  let (x_sk:Z,x_pk:G) = DH_keygen() in
  out(c, x_pk);

  (* Receiving second message *)
  in(c,(y_pk:G,sig:S));
  if SIG_verify(sig, (x_pk,y_pk), s_pk) then
  let k_C = Hash(exp(y_pk,x_sk)) in
  0
.
```

→

```
let Client(s_pk:PK) =

  (* Send first message *)
  let (x_sk:Z,x_pk:G) = DH_keygen() in
  event HonestClientShare(x_pk);
  out(c, x_pk);

  (* Receiving second message *)
  in(c,(y_pk:G,sig:S));
  if SIG_verify(sig, (x_pk,y_pk), s_pk) then
  let k_C = Hash(exp(y_pk,x_sk)) in
  event ClientAccept(s_pk,x_pk,y_pk,k_C);
  0
.
```

How to express security properties

Raising events

```
event HonestClientShare(G).
event HonestSever(PK).

event ServerAccept(PK,G,G,bitstring).
event ClientAccept(PK,G,G,bitstring).
```

```
let Server(s_sk:SK) =

  (* Receiving first message *)
  in(c, x_pk:G);

  let (y_sk:Z,y_pk:G) = DH_keygen() in
  let sig = SIG_sign((x_pk,y_pk),s_sk) in
  let k_S = Hash(exp(x_pk,y_sk)) in

  (* Sending second message *)
  out(c,(y_pk,sig));
  0
.
```

→

```
let Server(s_sk:SK) =

  (* Receiving first message *)
  in(c, x_pk:G);

  let (y_sk:Z,y_pk:G) = DH_keygen() in
  let sig = SIG_sign((x_pk,y_pk),s_sk) in
  let k_S = Hash(exp(x_pk,y_sk)) in
  event ServerAccept(s_pk,x_pk,y_pk,k_S);

  (* Sending second message *)
  out(c,(y_pk,sig));
  0
.
```

How to express security properties

Raising events

```
event HonestClientShare(G).
event HonestSever(PK).

event ServerAccept(PK,G,G,bitstring).
event ClientAccept(PK,G,G,bitstring).
```

```
process
  (
    !
    (* Initialize a new server  *)
    let (s_sk:SK,s_pk:PK) = SIG_gen() in
    (* Give make public the public key *)
    out(c,s_pk);

    !
    (* Run multiple session of the server *)
    Server(s_sk,s_pk)
  ) | (
    !
    (* Run multiple session of the client *)
    in(c,s_pk:PK);
    Client(s_pk)
  )
```

→

```
process
  (
    !
    (* Initialize a new server  *)
    let (s_sk:SK,s_pk:PK) = SIG_gen() in
    event HonestServer(s_pk);
    (* Give make public the public key *)
    out(c,s_pk);

    !
    (* Run multiple session of the server *)
    Server(s_sk,s_pk)
  ) | (
    !
    (* Run multiple session of the client *)
    in(c,s_pk:PK);
    Client(s_pk)
  )
```

How to express security properties

Key compromision

```
event CompromiseServer(PK).
event CompromiseClientShare(G).
event CompromiseServerShare(G).
```

```
...
(event CompromiseClientShare(x_pk); out(c,x_sk)) | P
```

```
let Client(s_pk:PK) =

  (* Send first message *)
  let (x_sk:Z,x_pk:G) = DH_keygen() in
  event HonestClientShare(x_pk);
  out(c, x_pk);

  (* Receiving second message *)
  in(c,(y_pk:G,sig:S));
  if SIG_verify(sig, (x_pk,y_pk), s_pk) then
  let k_C = Hash(exp(y_pk,x_sk)) in
  event ClientAccept(s_pk,x_pk,y_pk,k_C);
  0
.
```

→

```
let Client(s_pk:PK) =

  (* Send first message *)
  let (x_sk:Z,x_pk:G) = DH_keygen() in
  event HonestClientShare(x_pk);
  out(c, x_pk);
    (* Key compromission *)
    (event CompromiseClientShare(x_pk); out(c,x_sk)) |

  (* Receiving second message *)
  in(c,(y_pk:G,sig:S));
  if SIG_verify(sig, (x_pk,y_pk), s_pk) then
  let k_C = Hash(exp(y_pk,x_sk)) in
  event ClientAccept(s_pk,x_pk,y_pk,k_C);
  0
.
```

How to express security properties

Key compromision

```
event CompromiseServer(PK).
event CompromiseClientShare(G).
event CompromiseServerShare(G).
```

```
...
(event CompromiseClientShare(x_pk); out(c,x_sk)) | P
```

```
let Server(s_sk:SK) =

  (* Receiving first message *)
  in(c, x_pk:G);

  let (y_sk:Z,y_pk:G) = DH_keygen() in
  let sig = SIG_sign((x_pk,y_pk),s_sk) in
  let k_S = Hash(exp(x_pk,y_sk)) in
  event ServerAccept(s_pk,x_pk,y_pk,k_S);

  (* Sending second message *)
  out(c,(y_pk,sig));
  0
.
```

→

```
let Server(s_sk:SK,s_pk:PK) =
    (* Key compromision *)
    (event CompromiseServer(s_pk); out(c,s_sk)) |

  (* Receiving first message *)
  in(c, x_pk:G);

  let (y_sk:Z,y_pk:G) = DH_keygen() in
    (* Key compromision *)
    (event CompromiseServerShare(y_pk); out(c,y_sk)) |

  let sig = SIG_sign((x_pk,y_pk),s_sk) in
  let k_S = Hash(exp(x_pk,y_sk)) in
  event ServerAccept(s_pk,x_pk,y_pk,k_S);

  (* Sending second message *)
  out(c,(y_pk,sig));
  0.
```

Writing the queries

Correspondence queries

```
query x1,x2,x3:bitstring; F_1(x_1) && ... && F_n(x_2,x_3) ==> ϕ.
```

For all traces of the protocol, for all bitstrings x1,x2,x3,
if $F_1(x_1) \wedge \ldots \wedge F_n(x_2, x_3)$ occurs in the trace then ϕ is true

Writing the queries

Injective Client-side authentication

Every time an honest client completes a session thinking that it shares a key k with an honest server with public s_{pk}, then that server must have completed a distinct session of the protocol with that client and also obtained the shared key k

```
(* Client-side authentication *)
query s_pk:PK, x_pk,y_pk:G, k : bitstring;
  (* if a client accept *)
  inj-event(ClientAccept(s_pk,x_pk,y_pk,k)) &&
  (* and the s_pk is from an honest server *)
  event(HonestServer(s_pk))

  ==>
  (* then the server must have completed a distinct corresponding session *)
  inj-event(ServerAccept(s_pk,x_pk,y_pk,k))
.
```

Writing the queries

Injective Client-side authentication (with compromising scenarios)

Every time an honest client completes a session thinking that it shares a key k with an honest server with public s_{pk}, then that server must have completed a distinct session of the protocol with that client and also obtained the shared key k

```
(* Client-side authentication *)
query s_pk:PK, x_pk,y_pk:G, k : bitstring;
  (* if a client accept *)
  inj-event(ClientAccept(s_pk,x_pk,y_pk,k)) &&
  (* and the s_pk is from an honest server *)
  event(HonestServer(s_pk))

  ==>
  (* then the server must have completed a distinct corresponding session *)
  inj-event(ServerAccept(s_pk,x_pk,y_pk,k)) ||
  (* or the server was compromised *)
  event(CompromiseServer(s_pk))
.
```

Writing the queries

Forward secrecy

The attacker cannot learn the session key, even if they compromise the server in the future.

```
(* Forward Secrecy, client side *)
query s_pk:PK, x_pk,y_pk:G, k:bitstring, i,j:time;
  (* if a client accept *)
  event(ClientAccept(s_pk,x_pk,y_pk,k))@i
  &&
  (* and the s_pk is from an honest server *)
  event(HonestServer(s_pk))
  &&
  (* and the attacker knows k *)
  attacker(k)

  ==>
  (* then either the public server was compromised before the client accepted *)
  event(CompromiseServer(s_pk))@j && j<i ||
  (* or the corresponding client share was compromised *)
  event(CompromiseClientShare(x_pk)) ||
  (* or the corresponding server share was compromised *)
  event(CompromiseServerShare(y_pk))
.
```

Writing the queries

Forward secrecy

The attacker cannot learn the session key, even if they compromise the server in the future.

```
(* Forward Secrecy, server side *)
query s_pk:PK, x_pk,y_pk:G, k : bitstring;
  (* if a server accept *)
  event(ServerAccept(s_pk,x_pk,y_pk,k))
  &&
  (* and the x_pk is from an honest client *)
  event(HonestClientShare(x_pk))
  &&
  attacker(k)
  ==>
  (* either the corresponding client share was compromised *)
  event(CompromiseClientShare(x_pk)) ||
  (* or the corresponding server share was compromised *)
  event(CompromiseServerShare(y_pk))
.
```

Writing the queries

Sanity checks

Checking that the model is executable

```
(* Sanity check, executability. Must be false. *)
query s_pk:pkey, x_pk,y_pk:G, k : bitstring;
  event(ServerAccept(s_pk,x_pk,y_pk,k)) &&
  event(ClientAccept(s_pk,x_pk,y_pk,k))
  ==>
  false.
```

or to be sure that it's an uncompromised session

```
(* Sanity check, executability. Must be false. We want a session
to terminate with an honest and uncompromised server. *)
query s_pk:pkey, x_pk,y_pk:G, k : bitstring;
  event(ServerAccept(s_pk,x_pk,y_pk,k)) &&
  event(ClientAccept(s_pk,x_pk,y_pk,k))
  ==>
  event(CompromiseServer(s_pk)).
```

DEMO

Demo (The process)

```
Process 0 (that is, the initial process):
(
    {1}!
    {2}new sk: SK;
    {3}let (s_sk: SK,s_pk: PK) = (sk,SIG_pk(sk)) in
    {4}event HonestServer(s_pk);
    {5}out(c, s_pk);
    {6}!
    (
        {7}event CompromiseServer(s_pk);
        {8}out(c, s_sk)
    ) | (
        {9}in(c, x_pk: G);
        {10}new a: Z;
        {11}let (y_sk: Z,y_pk: G) = (a,exp(g,a)) in
        (
            {12}event CompromiseServerShare(y_pk);
            {13}out(c, y_sk)
        ) | (
            {14}let sig: S = SIG_sign((x_pk,y_pk),s_sk) in
            {15}let k_S: bitstring = Hash(exp(x_pk,y_sk)) in
            {16}event ServerAccept(s_pk,x_pk,y_pk,k_S);
            {17}out(c, (y_pk,sig))
        )
    )
) | (
    {18}!
    {19}in(c, s_pk_1: PK);
    {20}new a_1: Z;
    {21}let (x_sk: Z,x_pk_1: G) = (a_1,exp(g,a_1)) in
    {22}event HonestClientShare(x_pk_1);
    {23}out(c, x_pk_1);
    (
        {24}event CompromiseClientShare(x_pk_1);
        {25}out(c, x_sk)
    ) | (
        {26}in(c, (y_pk_1: G,sig_1: S));
        {27}if SIG_verify(sig_1,(x_pk_1,y_pk_1),s_pk_1) then
        {28}let k_C: bitstring = Hash(exp(y_pk_1,x_sk)) in
        {29}event ClientAccept(s_pk_1,x_pk_1,y_pk_1,k_C)
    )
)
```

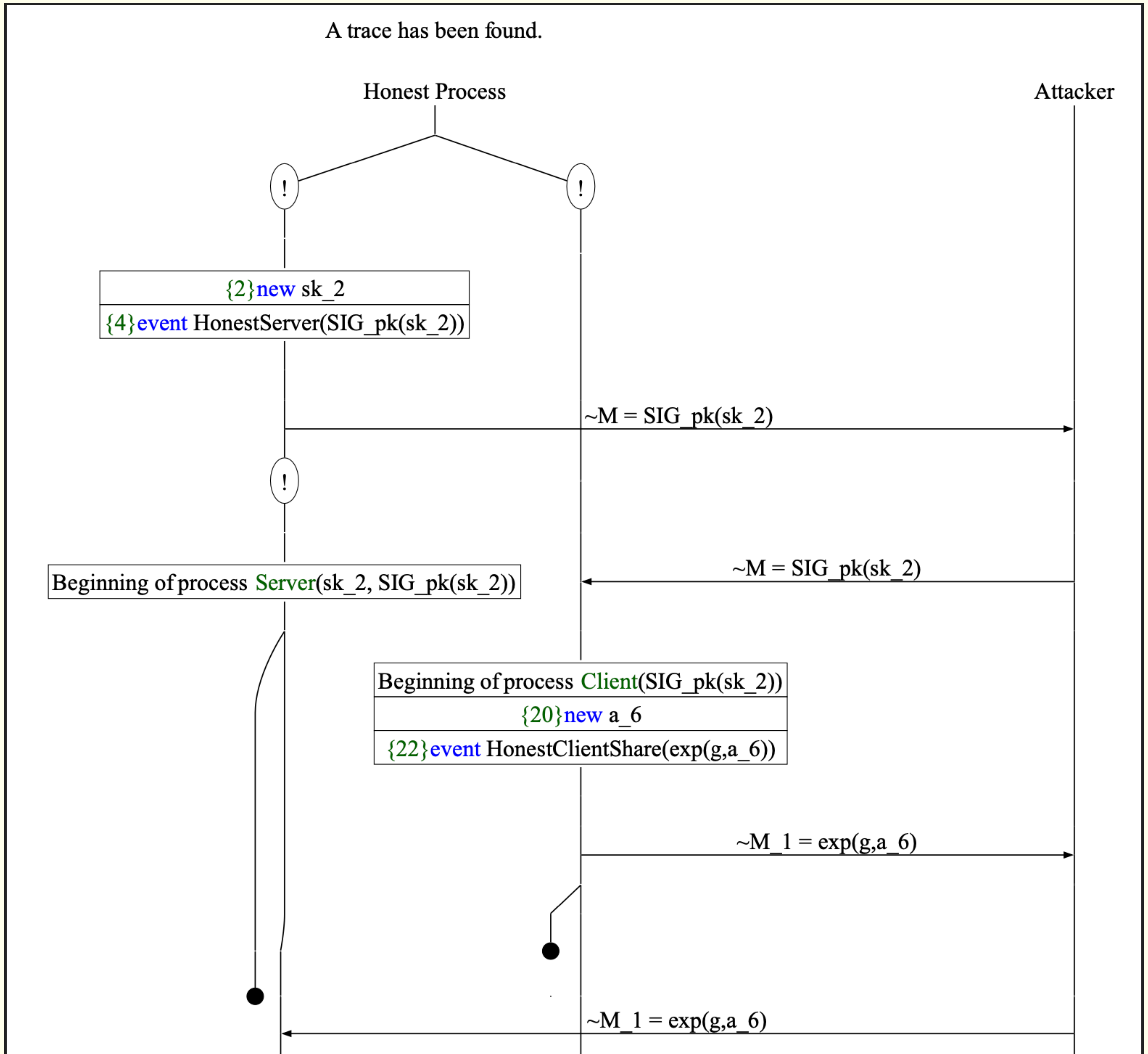
Demo (The summary)

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

Query inj-event(ClientAccept(s_pk_2,x_pk_2,y_pk_2,k)) && event(HonestServer(s_pk_2)) ==> inj-event(ServerAccept(s_pk_2,x_pk_2,y_pk_2,k)) is false.

Query inj-event(ClientAccept(s_pk_2,x_pk_2,y_pk_2,k)) && event(HonestServer(s_pk_2)) ==> inj-event(ServerAccept(s_pk_2,x_pk_2,y_pk_2,k)) || event(CompromiseSer
ver(s_pk_2)) is true.

Query event(ClientAccept(s_pk_2,x_pk_2,y_pk_2,k))@i && event(HonestServer(s_pk_2)) && attacker(k) ==> (event(CompromiseServer(s_pk_2))@j && i > j) || event(Com
promiseClientShare(x_pk_2)) || event(CompromiseServerShare(y_pk_2)) is true.

Query event(ServerAccept(s_pk_2,x_pk_2,y_pk_2,k))@i && event(HonestClientShare(x_pk_2)) && attacker(k) ==> event(CompromiseClientShare(x_pk_2)) || event(Compro
miseServerShare(y_pk_2)) is true.

Query not (event(ServerAccept(s_pk_2,x_pk_2,y_pk_2,k)) && event(ClientAccept(s_pk_2,x_pk_2,y_pk_2,k))) is false.

Query event(ServerAccept(s_pk_2,x_pk_2,y_pk_2,k)) && event(ClientAccept(s_pk_2,x_pk_2,y_pk_2,k)) ==> event(CompromiseServer(s_pk_2)) is false.

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

Demo (The attack trace)

```
A more detailed output of the traces is available with
  set traceDisplay = long.

new sk: SK creating sk_2 at {2} in copy a_4

event HonestServer(SIG_pk(sk_2)) at {4} in copy a_4

out(c, ~M) with ~M = SIG_pk(sk_2) at {5} in copy a_4

in(c, ~M) with ~M = SIG_pk(sk_2) at {19} in copy a_5

new a_1: Z creating a_6 at {20} in copy a_5

event HonestClientShare(exp(g,a_6)) at {22} in copy a_5

out(c, ~M_1) with ~M_1 = exp(g,a_6) at {23} in copy a_5

event CompromiseClientShare(exp(g,a_6)) at {24} in copy a_5

out(c, ~M_2) with ~M_2 = a_6 at {25} in copy a_5

in(c, ~M_1) with ~M_1 = exp(g,a_6) at {9} in copy a_4, a_7

new a: Z creating a_8 at {10} in copy a_4, a_7

event ServerAccept(SIG_pk(sk_2),exp(g,a_6),exp(g,a_8),Hash(exp(exp(g,a_6),a_8))) at {16} in copy a_4, a_7 (goal)

out(c, (~M_3,~M_4)) with ~M_3 = exp(g,a_8), ~M_4 = SIG_sign((exp(g,a_6),exp(g,a_8)),sk_2) at {17} in copy a_4, a_7

event CompromiseServerShare(exp(g,a_8)) at {12} in copy a_4, a_7

out(c, ~M_5) with ~M_5 = a_8 at {13} in copy a_4, a_7

in(c, (~M_3,~M_4)) with ~M_3 = exp(g,a_8), ~M_4 = SIG_sign((exp(g,a_6),exp(g,a_8)),sk_2) at {26} in copy a_5

event ClientAccept(SIG_pk(sk_2),exp(g,a_6),exp(g,a_8),Hash(exp(exp(g,a_8),a_6))) at {29} in copy a_5 (goal)

The event ServerAccept(SIG_pk(sk_2),exp(g,a_6),exp(g,a_8),Hash(exp(exp(g,a_8),a_6))) is executed at {16} in copy a_4, a_7.
The event ClientAccept(SIG_pk(sk_2),exp(g,a_6),exp(g,a_8),Hash(exp(exp(g,a_8),a_6))) is executed at {29} in copy a_5.
A trace has been found.
```

Demo (The attack trace in PDF)



WHAT ELSE CAN WE DO?

Well adapted for large model

Signal

The Double Ratchet Algorithm

We apply the same principle as we seen on SignedDH

- ***GENERATE_DH()***: Returns a new Diffie-Hellman key pair.
- ***DH(dh_pair, dh_pub)***: Returns the output from the Diffie-Hellman calculation between the private key from the DH key pair *dh_pair* and the DH public key *dh_pub*. If the DH function rejects invalid public keys, then this function may raise an exception which terminates processing.
- ***KDF_RK(rk, dh_out)***: Returns a pair (32-byte root key, 32-byte chain key) as the output of applying a KDF keyed by a 32-byte root key *rk* to a Diffie-Hellman output *dh_out*.
- ***KDF_CK(ck)***: Returns a pair (32-byte chain key, 32-byte message key) as the output of applying a KDF keyed by a 32-byte chain key *ck* to some constant.
- ***ENCRYPT(mk, plaintext, associated_data)***: Returns an AEAD encryption of *plaintext* with message key *mk* [5]. The *associated_data* is authenticated but is not included in the ciphertext. Because each message key is only used once, the AEAD nonce may handled in several ways: fixed to a constant; derived from *mk* alongside an independent AEAD encryption key; derived as an additional output from *KDF_CK()*; or chosen randomly and transmitted.
- ***DECRYPT(mk, ciphertext, associated_data)***: Returns the AEAD decryption of *ciphertext* with message key *mk*. If authentication fails, an exception will be raised that terminates processing.
- ***HEADER(dh_pair, pn, n)***: Creates a new message header containing the DH ratchet public key from the key pair in *dh_pair*, the previous chain length *pn*, and the message number *n*. The

Well adapted for large model

The Double Ratchet Algorithm

We apply the same principle as we seen on SignedDH

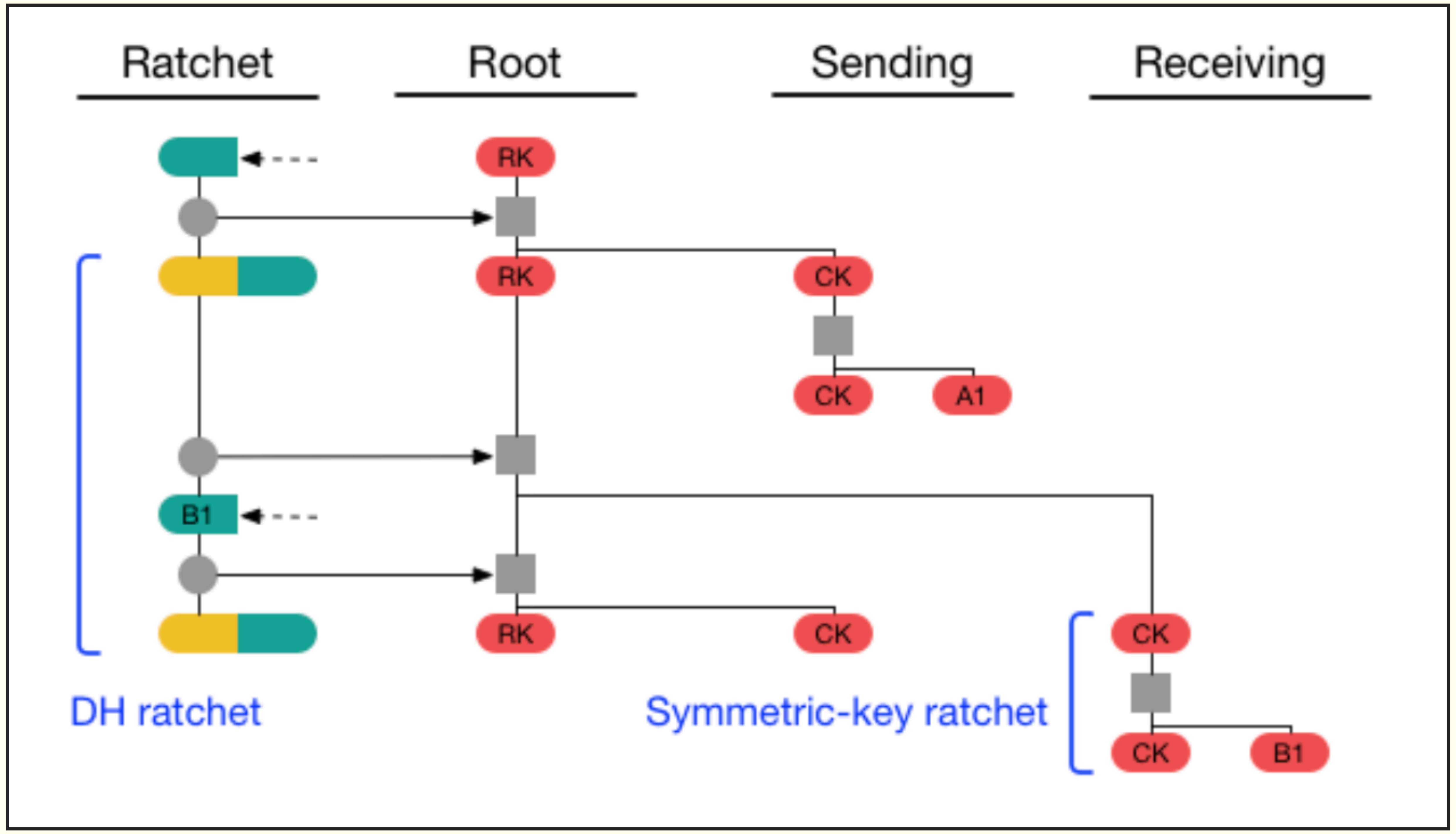
```
(** KDF_RK(rk, dh_out): Returns a pair (32-byte root key, 32-byte chain key) as the output of applying a KDF
keyed by a 32-byte root key rk to a Diffie-Hellman output dh_out.*)

fun kdf_rk_root(root_key, point): root_key.
fun kdf_rk_chain(root_key, point): chain_key.

letfun kdf_rk(rk:root_key, dh_out:point) =
  (kdf_rk_root(rk,dh_out), kdf_rk_chain(rk,dh_out)).

…

(** ENCRYPT(mk, plaintext, associated_data): Returns an AEAD encryption of plaintext with message key mk [5].*)
letfun encrypt(mk: message_key, plaintext:bitstring, ad: associated_data) =
  aead_enc(mk, encryption_nonce, plaintext, ad)
.

(** DECRYPT(mk, ciphertext, associated_data): Returns the AEAD decryption of ciphertext with message key mk. If
authentication fails, an exception will be raised that terminates processing. *)
letfun decrypt(mk: message_key, ciphertext:bitstring, ad: associated_data) =
  aead_dec(mk, encryption_nonce, ciphertext, ad)
.
```

Well adapted for large model

Signal

The Double Ratchet Algorithm

We apply the same principle as we seen on SignedDH



Well adapted for large model

Signal

The Double Ratchet Algorithm

We apply the same principle as we seen on SignedDH

```
def RatchetDecrypt(state, header, ciphertext, AD):
    plaintext = TrySkippedMessageKeys(state, header, ciphertext, AD)
    if plaintext != None:
        return plaintext
    if header.dh != state.DHr:
        SkipMessageKeys(state, header.pn)
        DHRatchet(state, header)
    SkipMessageKeys(state, header.n)
    state.CKr, mk = KDF_CK(state.CKr)
    state.Nr += 1
    return DECRYPT(mk, ciphertext, CONCAT(AD, header))

def TrySkippedMessageKeys(state, header, ciphertext, AD):
    if (header.dh, header.n) in state.MKSKIPPED:
        mk = state.MKSKIPPED[header.dh, header.n]
        del state.MKSKIPPED[header.dh, header.n]
        return DECRYPT(mk, ciphertext, CONCAT(AD, header))
    else:
        return None
```

Well adapted for large model

Signal

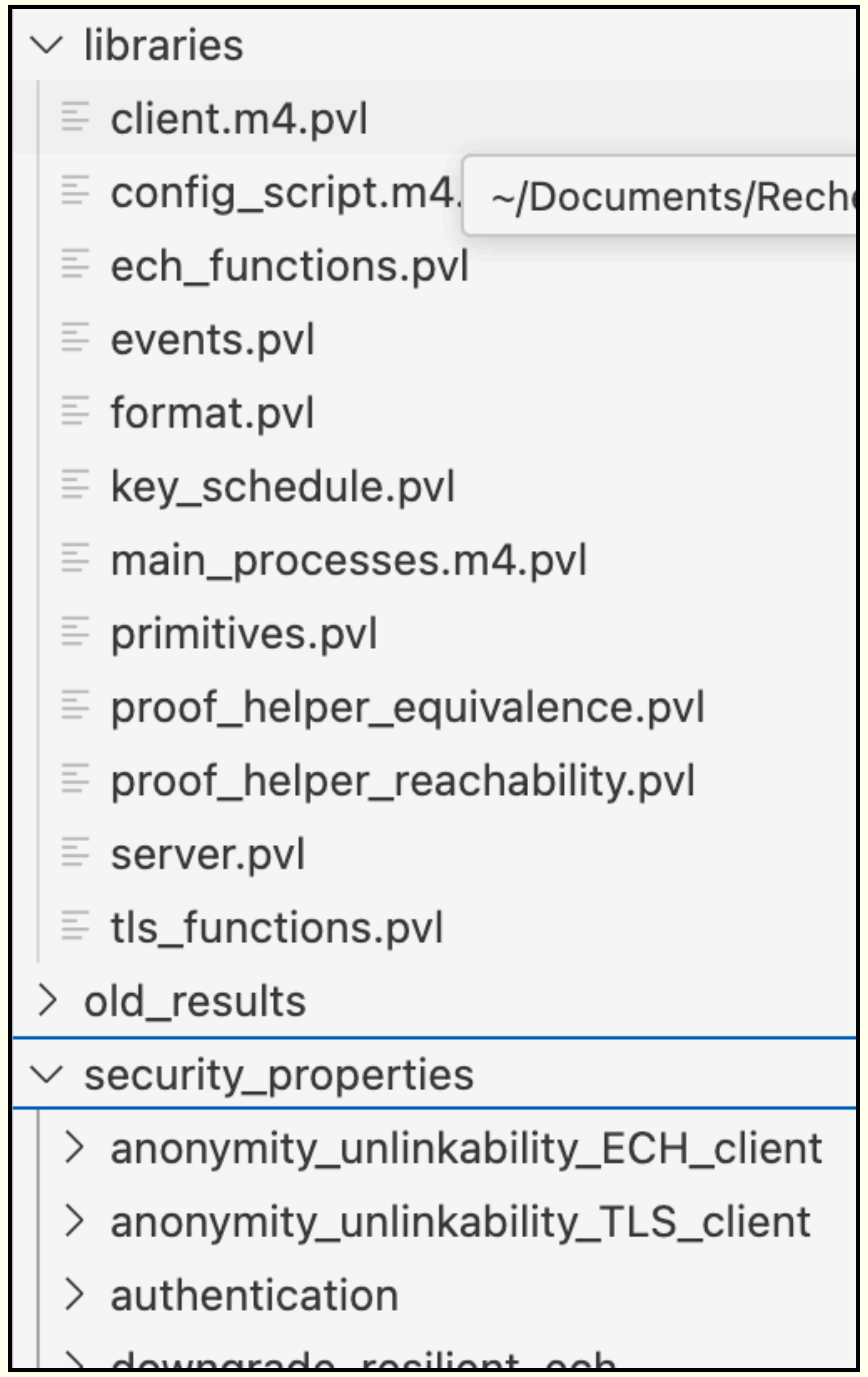
The Double Ratchet Algorithm

We apply the same principle as we seen on SignedDH

```
let BasicDecrypt(sinfo:session_info, hd:header, ciphertext:bitstring,
header_ad:associated_data,cell_value:cell_content) =
  let st_chan = state_chan(sinfo) in
  let Header(headerDH:point, headerPN:nat, headerN:nat) = hd in
  let cell(st,m_idx,r_idx,tr) = cell_value in
  let state(dhs, =headerDH, rk, cks, ckr, ns, =headerN, pn, hks, nhks, hkr, nhkr) = st in

  if ckr <> none_ck then

  let (newCKr:chain_key,mk:message_key) = kdf_ck(ckr) in
  event MessageKey(sinfo,Receiver,headerDH,mk,headerN,r_idx);
  let new_st = state(dhs, headerDH, rk, cks, newCKr, ns, headerN +1, pn, hks, nhks, hkr, nhkr) in

  let plaintext = decrypt(mk, ciphertext, header_ad) in
      event Receive(sinfo, headerDH,headerN,plaintext);
      out(st_chan,cell(new_st,m_idx+1,r_idx,tr+1))
    else
      event AuthFail(sinfo,headerDH,headerN);
      out(st_chan,cell(new_st,m_idx+1,r_idx,tr+1))
.
```

Well adapted for large model

- Uses libraries



Well adapted for large model

- Uses libraries

- Uses macro

```
(* Existential collisions *)
def NotCollisionResistantHash(t_input,t_output,h) {

  const coll_a:t_input.
  const coll_b:t_input.

  equation h(coll_a) = h(coll_b).
}

def NotMultipleCollisionResistantHash(t_input,t_output,h) {

  fun coll_a(bitstring):t_input.
  fun coll_b(bitstring):t_input.

  equation forall x:bitstring; h(coll_a(x)) = h(coll_b(x)).
}

(* Chosen-prefix collision attacks *)
(* Note: Chosen-prefix collision attack:  Given two different prefixes p1 and p2,
find two appendages m1 and m2 such that hash(p1 ‖ m1) = hash(p2 ‖ m2), where ‖
denotes the concatenation operation. *)
def ChosenPrefixCollisionAttacks(t_input,t_output,h) {
  fun to_append1(t_input, t_input):t_input.
  fun to_append2(t_input, t_input):t_input.

  fun t_input_OF_bitstring(bitstring): t_input [typeConverter]. (*need to convert
pairs (of type bitstring) into t_input *)


…
```

Well adapted for large model

- Uses libraries

- Uses macro

- SIG.Gen : $\emptyset \rightarrow_{\$} \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_{\$} \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0,1\}$ Signature verification

```
fun SIG_pk(SK):PK.
fun SIG_sign(bitstring,SK):S.
```

```
fun SIG_verify(S,bitstring,PK):bool
reduc
  forall m:bitstring, sk:SK; SIG_verify(SIG_sign(sk,m),SIG_pk(sk)) = true
  otherwise forall pk:PK, m:bitstring, sig:S; SIG_verify(sig,pk,m) = false.
```

Well adapted for large model

- Uses libraries
- Uses macro

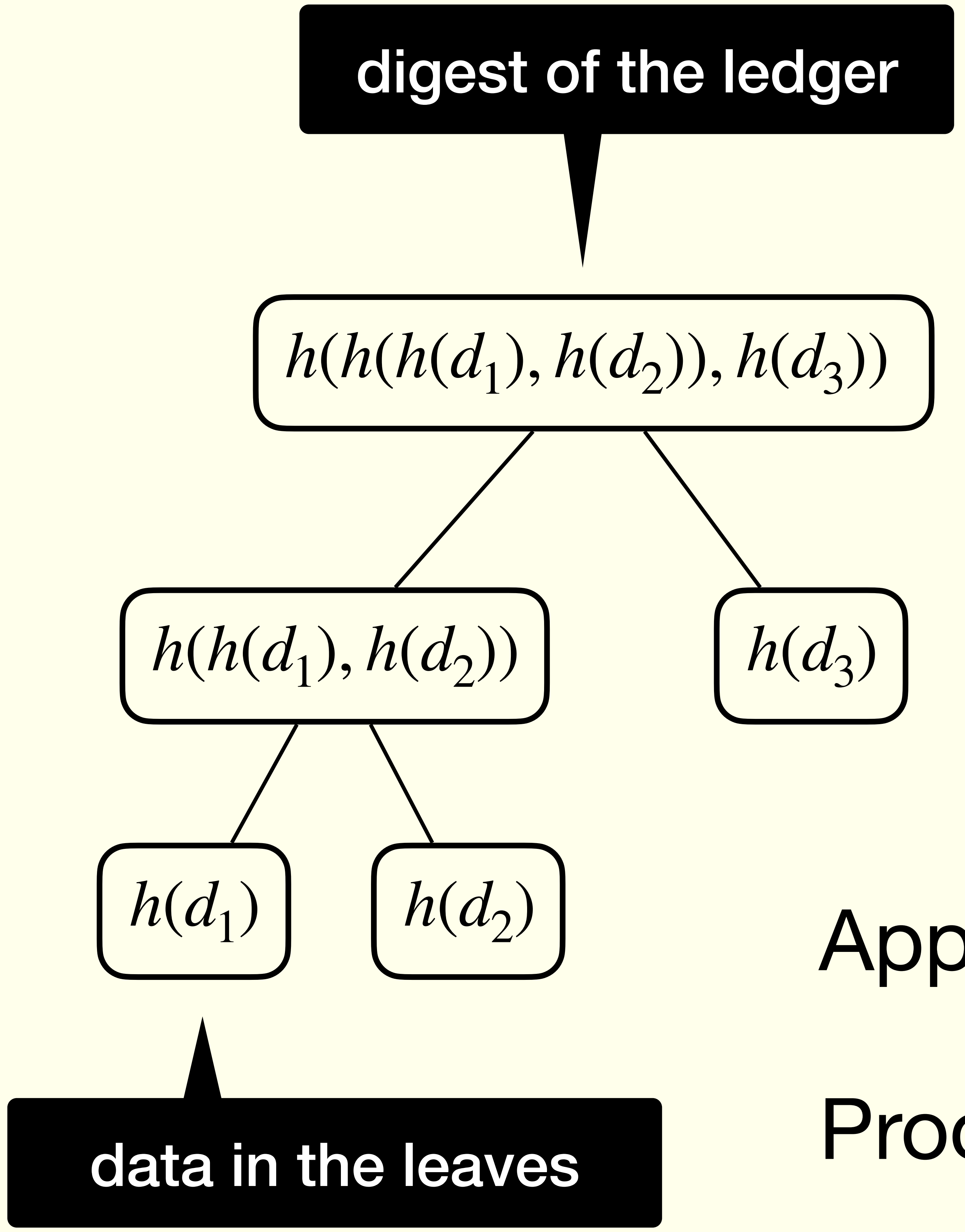
- SIG.Gen : $\emptyset \rightarrow_\$ \mathcal{SK} \times \mathcal{PK}$ Signing and verification key generation
- SIG.Sign : $\mathcal{M} \times \mathcal{SK} \rightarrow_\$ \mathcal{S}$ Signing procedure
- SIG.Verify : $\mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{0, 1\}$ Signature verification

```
def SignatureRandom(SIG_sign,SIG_verify) {
  type random.
  fun SIG_sign_aux(bitstring,random,SK):S.

  letfun SIG_sign(m:bitstring,sk:SK) =
    new r:random;
    SIG_sign_aux(m,r,sk)
  .

  fun SIG_verify(S,bitstring,PK):bool
  reduc
    forall m:bitstring, r:random, sk:SK; SIG_verify(SIG_sign_aux(m,r,sk),m,SIG_pk(sk)) = true
    otherwise forall pk:PK, m:bitstring, sig:S; SIG_verify(sig,m,pk) = false.
}

expand SignatureRandom(SIG_sign,SIG_verify).
expand SignatureRandom(SIG_sign2,SIG_verify2).
```

Well adapted for large model

- Uses libraries
- Uses macro
- Complex data structure

Merkel tree



Append only structure

Proof of presence in O(log(n))

Proof of extension in O(log(n))

Well adapted for large model

- Uses libraries
- Uses macro
- Complex data structure

```
(* Proof of presence *)
fun PP(list):proof_of_presence [data].

clauses
  forall x:bitstring;
    verify_pp(PP(nil),x,hash(leaf(x)));
  forall pl:list, x:bitstring, d_left,d_right:digest;
    verify_pp(PP(pl),x,d_left) ->
    verify_pp(PP(cons((left,d_right),pl)),x,hash(node(d_left,d_right)));
  forall pl:list, x:bitstring, d_left,d_right:digest;
    verify_pp(PP(pl),x,d_right) ->
    verify_pp(PP(cons((right,d_left),pl)),x,hash(node(d_left,d_right)))
.
```

Well adapted for large model

- Uses libraries
- Uses macro
- Complex data structure
- Locking memory cell
- Global Table

Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

Lock and read

Write and unlock

```
let Q =
  …
  in(cell,x:nat);
  event A;
  event C;
  out(cell,n);
  …
```

Communication are synchronous on private channels: If no output available, all processes trying to input are « blocked »

Well adapted for large model

- Uses libraries
- Uses macro
- Complex data structure
- Locking memory cell
- Global Table
- Lemmas, axioms, restrictions

Lemmas, axioms, restrictions

```
restriction phi_1.
…
restriction phi_n.

axiom aphi_1.
…
axiom aphi_m.

lemma lphi_1.

lemma lphi_k.

query attacker(s) ==>
false.
```

Restrictions « restrict » the traces considered in axioms, lemmas and queries.

`query attacker(s).` holds if no trace satisfying `phi_1, …, phi_n` reveals `s`

1. Proverif assumes that the axioms `aphi_1, …, aphi_n` hold.
2. Proverif tries to prove in order the lemmas `lphi_1, …, lphi_k` reusing all axioms and previously proved lemmas
3. Proverif tries to prove the query `query attacker(s).` reusing all axioms and all lemmas.

Well adapted for large model

- Uses libraries
- Uses macro
- Complex data structure
- Locking memory cell
- Global Table
- Lemmas, axioms, restrictions
- Proof by induction

```
query pe:proof_of_extension, pp1,pp2:proof_of_presence, d1,d2:digest,
x:bitstring;
  verify_pe(pe,d1,d2) && verify_pp(pp1,x,d1) ==> verify_pp(pp2,x,d2)
  [induction]
.
```

Well adapted for large model

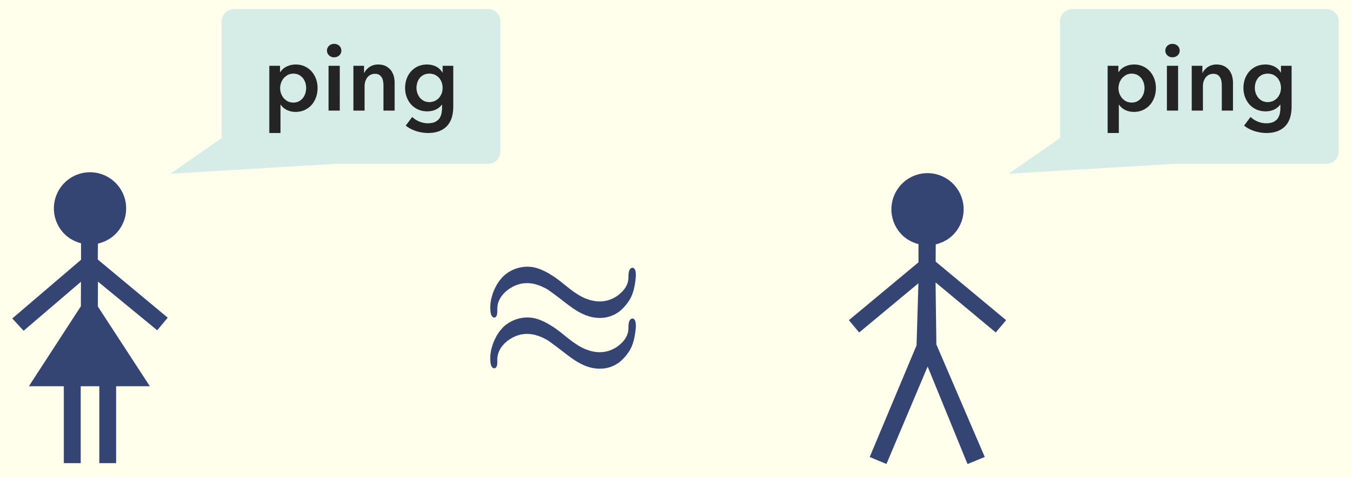
- Uses libraries
- Uses macro
- Complex data structure
- Locking memory cell
- Global Table
- Lemmas, axioms, restrictions
- Proof by induction
- Simple arithmetic on natural number
- …

```
let P =
  in(c,x:bitstring);
  in(cellP,i:nat);
  let j = sdec(x,k) in
  if j > i
  then
    event Accept(j);
    out(cellP,j)
  else
    out(cellP,i)
.
```

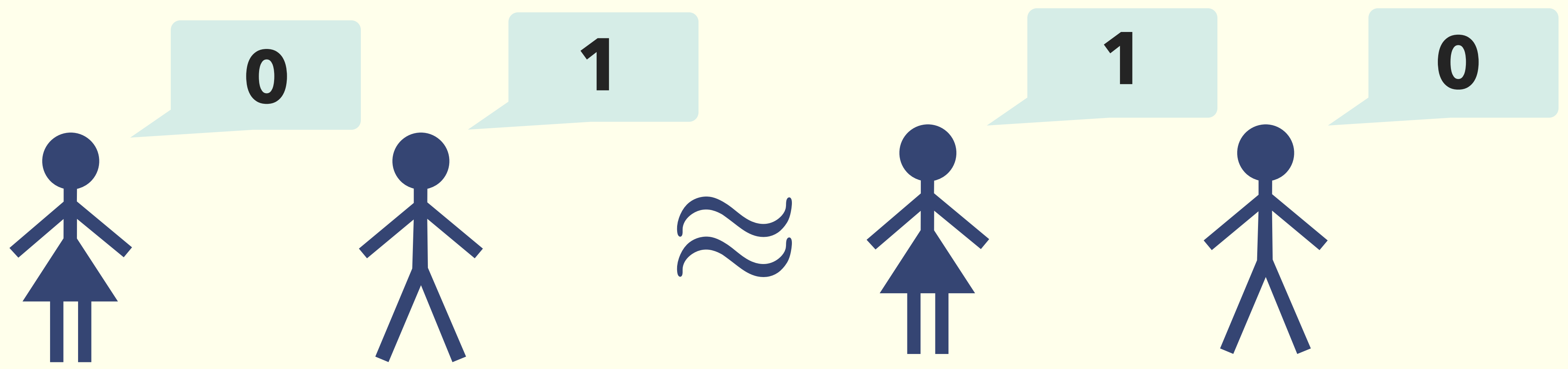
Equivalence properties

Indistinguishability

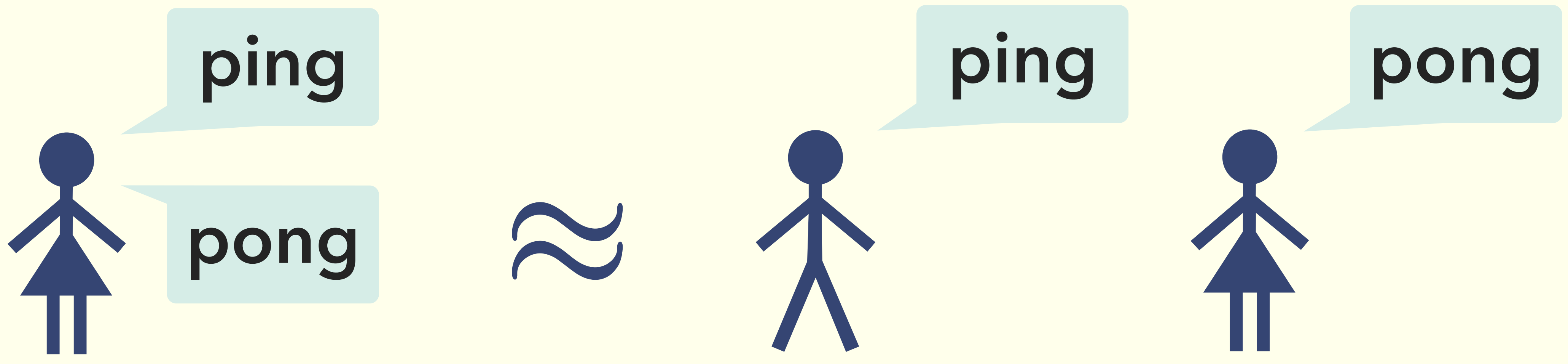
of two situations where the private attribute differs



Anonymity



Vote privacy



Unlinkability

```
let system1 = setup | voter(skA,v1) | voter(skB,v2).
let system2 = setup | voter(skA,v2) | voter(skB,v1).

equivalence system1 system2
```

ProVerif

Mature

24 years !

Large user base

350+ papers using it

Reachability properties

Equivalence properties but

Fails on many unlinkability case studies

Expressive but

May not terminate
May yield false attacks specially on protocols with mutable states

Efficient but

Can still take hours/days on large case studies
(e.g. Noise Protocol Framework, TLS)

WHAT´S NEXT?

Heavy memory consumption: TLS with ECH

Many features

- HelloRetryRequest
- Certificate-based Client Authentication
- Pre-Shared Keys and Tickets
- 0RTT
- Post Handshake Authentication
- Other TLS extensions (e.g. SNI)

Status	Reachability		Equivalence		Total	
Verified	358	60%	208	69%	566	63%
Stopped mostly due to OM (200-300GB)	230	39%	87	29%	317	36%
Total	**592**		**300**		**892**	

Many security properties

- Server Authentication
- Client Authentication
- Key and Transcript Agreement
- Data Stream Integrity
- Key Uniqueness
- Downgrade Resilience
- Key Secrecy
- Key Indistinguishability
- 1RTT Data Forward Secrecy
- 0RTT Data Secrecy
- Server Identity Privacy

We are limited by the memory capacity of our server

Heavy memory consumption: Ongoing prototype

Efficiency

Query	ProVerif 2.05	Prototype	Memory ratio
Key secrecy & Uniqueness	162 GB	6 GB	28.9
Authentication	141 GB	22 GB	6.4
Secrecy & Authenticity	162 GB	2 GB	67.5
Forward secrecy & Stream integrity	46 GB	11 GB	4.2
Post-handshake authentication	61 GB	39 GB	1.6
Key indistinguishability	34 GB	2 GB	18.9

Limited algebraic properties handled by equational theories

Scope

Currently handled equational theories

Equational theory with Finite Variant property

Linear equational theory

Encryption / Decryption

Digital signature

Limited Exponentiation

$$(g\hat{\ }x)\hat{\ }y = (g\hat{\ }y)\hat{\ }x$$

Some weak hash function

...

Not yet handled

Associative-Commutative

Homomorphism

Abelian groups

Natural number arithmetic

XOR

Homomorphic encryption

$$(g\hat{\ }x)\hat{\ }y = g\hat{\ }(y \times x)$$

$$(g\hat{\ }x) \times (g\hat{\ }y) = g\hat{\ }(x + y)$$

User Interface

Usability

Is the tool's output understandable by non-expert?

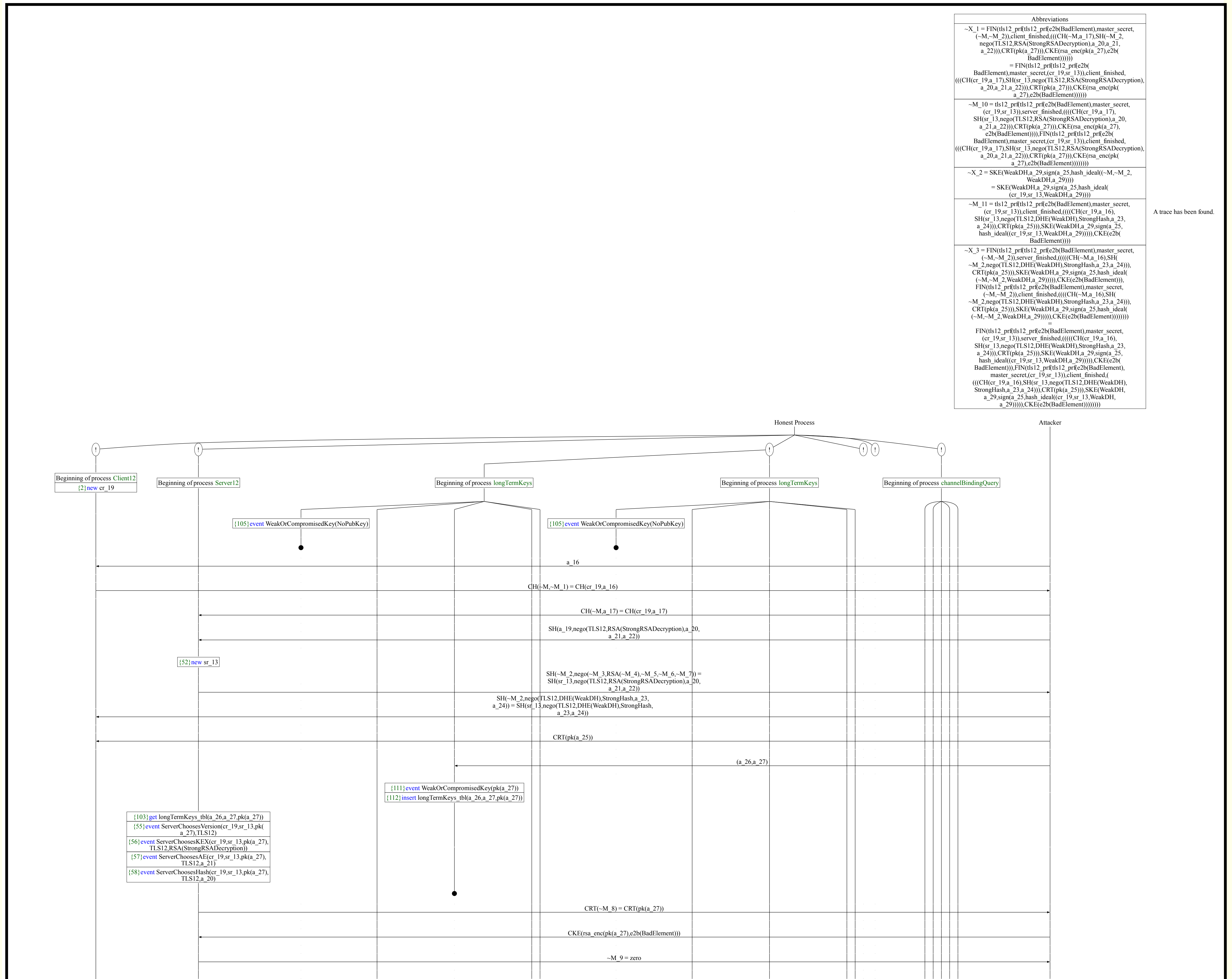
```
Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
mess(cellQ[],i_2) -> mess(cellQ[],i_2)
The hypothesis occurs before the conclusion.
1 rules inserted. Base: 1 rules (0 with conclusion selected). Queue: 3 rules.

Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
is_nat(i_2) && mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 1)
The hypothesis occurs strictly before the conclusion.
2 rules inserted. Base: 2 rules (0 with conclusion selected). Queue: 5 rules.

Rule with conclusion selected:
mess(cellQ[],0)
3 rules inserted. Base: 3 rules (1 with conclusion selected). Queue: 4 rules.

Rule with hypothesis fact 0 selected: attacker(cellQ[])
attacker(cellQ[]) && attacker(i_2) -> mess(cellQ[],i_2)
The 1st, 2nd hypotheses occur before the conclusion.
4 rules inserted. Base: 4 rules (1 with conclusion selected). Queue: 3 rules.

Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
is_nat(i_2) && mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 2)
The hypothesis occurs strictly before the conclusion.
5 rules inserted. Base: 5 rules (1 with conclusion selected). Queue: 5 rules.

Rule with conclusion selected:
mess(cellQ[],1)
6 rules inserted. Base: 6 rules (2 with conclusion selected). Queue: 4 rules.

Rule with hypothesis fact 0 selected: attacker(cellQ[])
is_nat(i_2) && attacker(cellQ[]) && attacker(i_2) -> mess(cellQ[],i_2 + 1)
The 1st, 2nd hypotheses occur strictly before the conclusion.
7 rules inserted. Base: 7 rules (2 with conclusion selected). Queue: 3 rules.
```

ProVerif' terminal output can be very "verbose" and hard to follow

User Interface

Usability

Is the tool’s output understandable by non-expert?



ProVerif can graphically display the attack on an Alice-Bob graph but it can become messy very quickly

How to make use of the many features in the tools?

Usability

ProVerif have many features that allow to go around non-termination issues even on large industrial case studies…

… but it usually requires to to understand the internal algorithm of the tools

Satisfactory for experts (to a certain degree)
but what about standard users?

- ???
  Need feedback from the community
- feature suggestion
- push for more automation

Conclusion

Try it !

http://proverif.inria.fr

Register to mailing list

Feel free to send your models if you reach a dead-end

Contact us to discuss and request new features

Thank you !