

# CryptoVerif: Mechanising Game-Based Proofs

## Part II

---

Benjamin Lipp

December 10, 2020

Inria Paris

## What to Expect from Part II

A more complex example, a protocol with multiple messages:  
Signed Diffie-Hellman Authenticated Key Exchange

What's new?

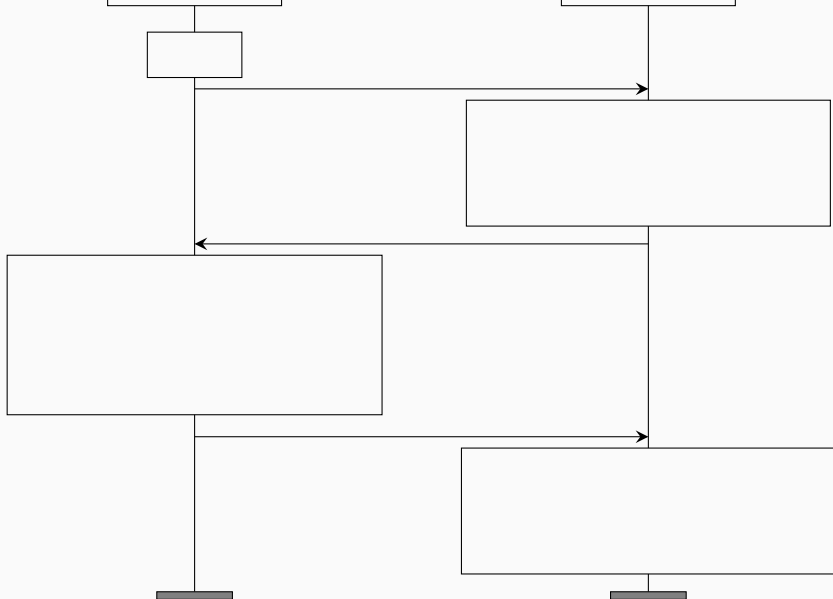
- model a random oracle
- use a Computational Diffie-Hellman (CDH) assumption
- prove key secrecy using **query secret**
- prove authentication properties using  
correspondence between events
- model a Public-Key Infrastructure using a list (**table** in  
CryptoVerif)

knows  $sk_A, pk_B$

A

knows  $sk_B, pk_A$

B



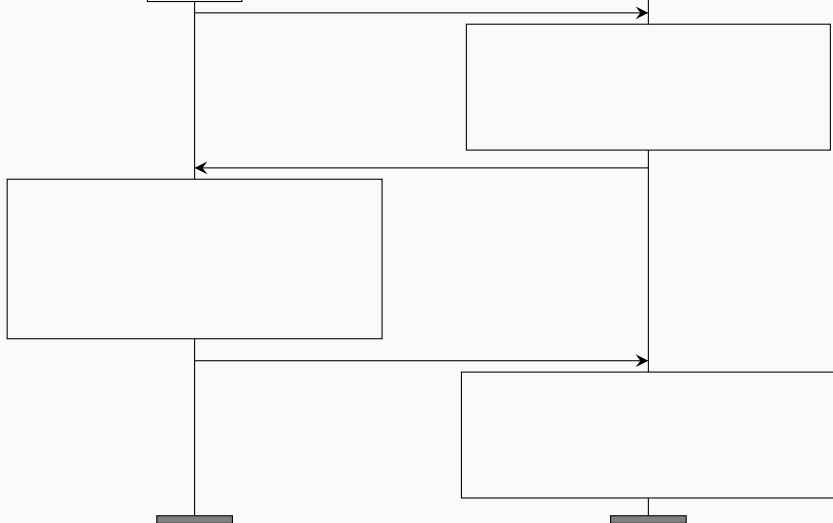
knows  $sk_A, pk_B$

A

$a \leftarrow \mathbb{Z}$

knows  $sk_B, pk_A$

B



knows  $sk_A, pk_B$

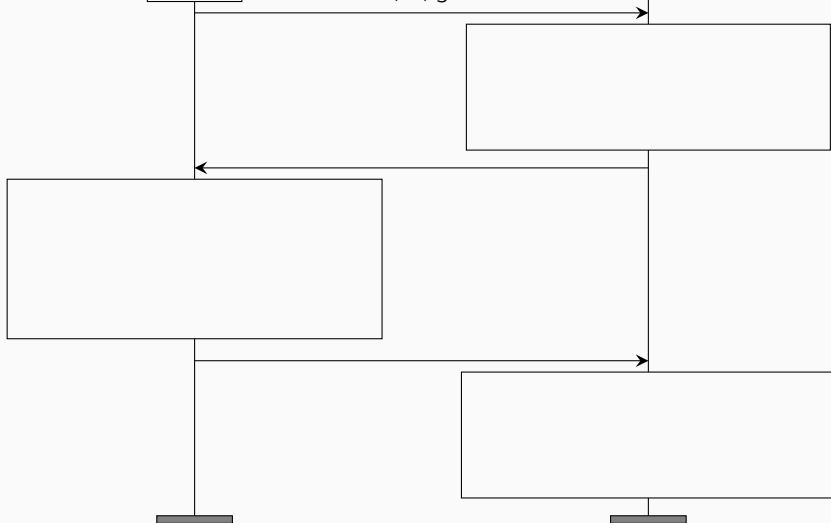
$A$

$a \leftarrow \mathbb{Z}$

$A, B, g^a$

knows  $sk_B, pk_A$

$B$



knows  $sk_A, pk_B$

A

$a \leftarrow \mathbb{Z}$

$A, B, g^a$

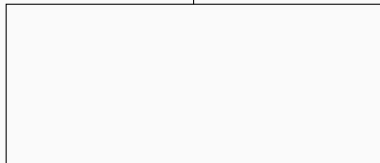
knows  $sk_B, pk_A$

B

$b \leftarrow \mathbb{Z}$

$sig_B \leftarrow \text{sign}(A \parallel B \parallel g^a \parallel g^b, sk_B)$

event  $\text{begin}_B(A, B, g^a, g^b)$



knows  $sk_A, pk_B$

A

$a \leftarrow \mathbb{Z}$

$A, B, g^a$

knows  $sk_B, pk_A$

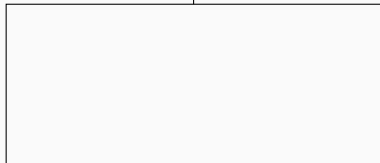
B

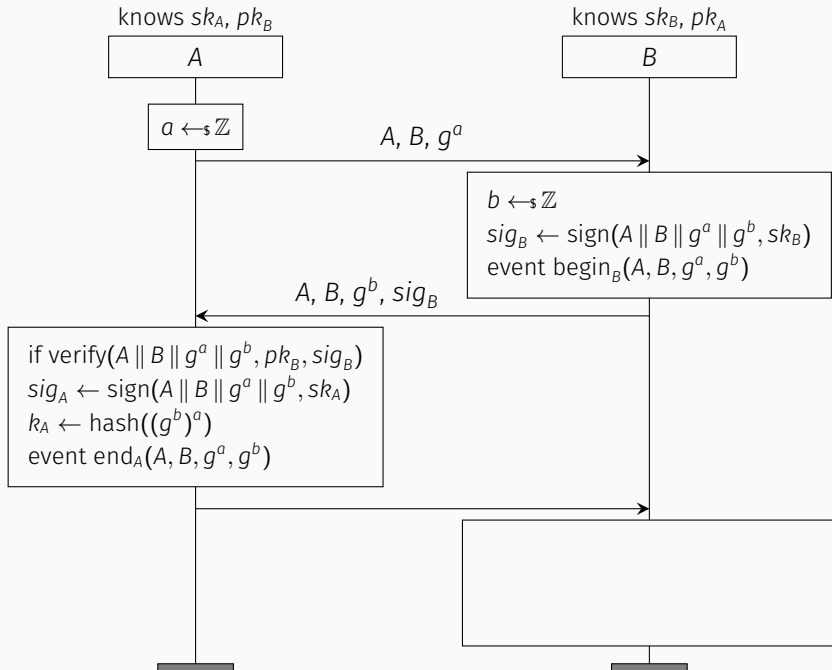
$b \leftarrow \mathbb{Z}$

$sig_B \leftarrow \text{sign}(A \parallel B \parallel g^a \parallel g^b, sk_B)$

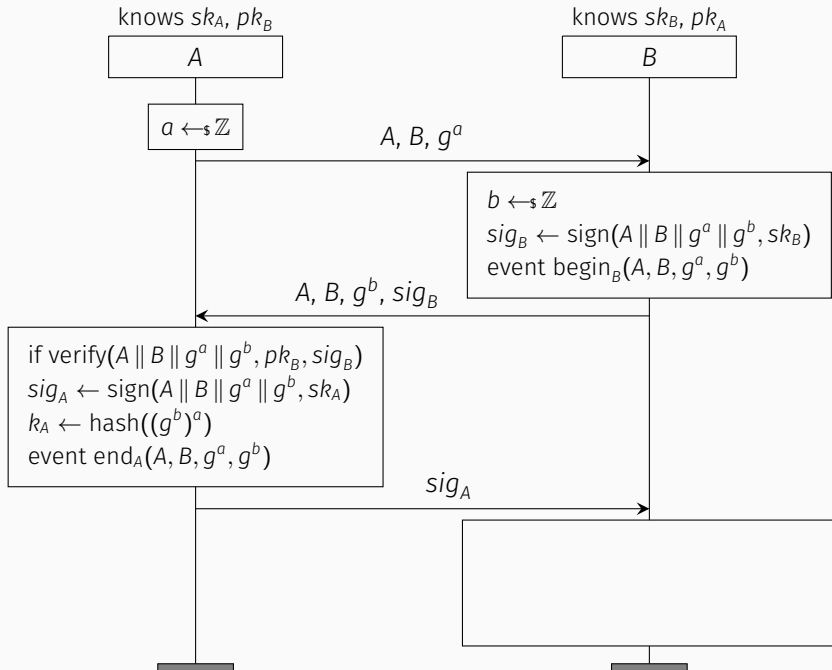
event  $\text{begin}_B(A, B, g^a, g^b)$

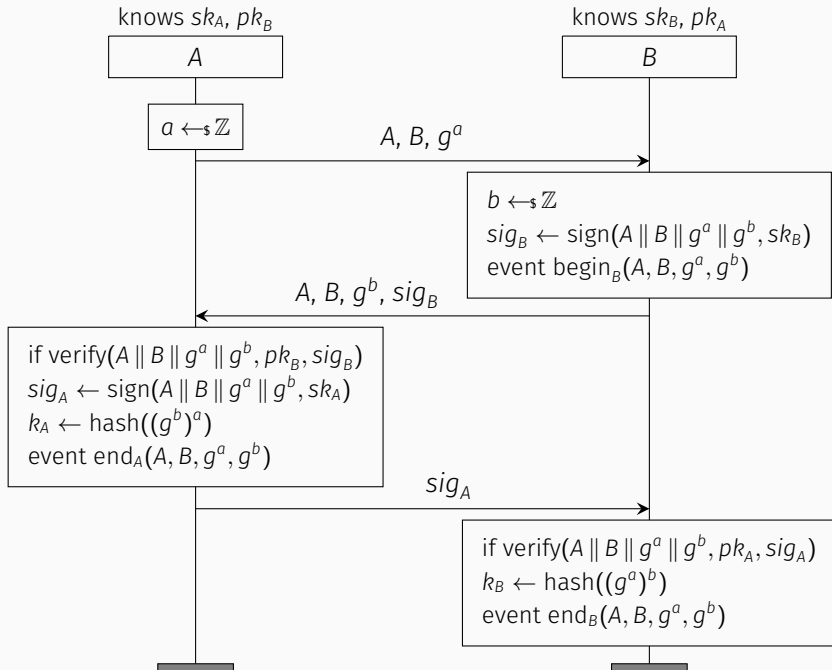
$A, B, g^b, sig_B$











## Signed Diffie-Hellman: Security Properties

- The shared secrets  $k_A$  and  $k_B$  are secret  
(indistinguishable from random bitstrings of equal length)  
query secret  $k_A$ . query secret  $k_B$ .

# Signed Diffie-Hellman: Security Properties

- The shared secrets  $k_A$  and  $k_B$  are secret  
(indistinguishable from random bitstrings of equal length)  
query secret  $k_A$ . query secret  $k_B$ .
- If  $A$  is convinced to have concluded a session with  $B$  using  
ephemerals  $g^a, g^b$ , then  $B$  actually started such a session  
query  $x:G, y:G$ ; inj-event(endA( $A, B, x, y$ )) ==>  
inj-event(beginB( $A, B, x, y$ )).

# Signed Diffie-Hellman: Security Properties

- The shared secrets  $k_A$  and  $k_B$  are secret  
(indistinguishable from random bitstrings of equal length)  
query secret  $k_A$ . query secret  $k_B$ .
- If  $A$  is convinced to have concluded a session with  $B$  using  
ephemerals  $g^a, g^b$ , then  $B$  actually started such a session  
query  $x:G, y:G$ ; inj-event(endA( $A, B, x, y$ )) ==>  
inj-event(beginB( $A, B, x, y$ )).
- If  $B$  is convinced to have concluded a session with  $A$  using  
ephemerals  $g^a, g^b$ , then  $A$  is likewise convinced  
query  $x:G, y:G$ ; inj-event(endB( $A, B, x, y$ )) ==>  
inj-event(endA( $A, B, x, y$ ))

# Cryptographic Assumptions

We use the following cryptographic assumptions to prove these security properties:

- hash is a random oracle
- (sign, verify) is a UF-CMA-secure probabilistic signature
- the CDH assumption holds in the group  $G$

# Cryptographic Assumptions

We use the following cryptographic assumptions to prove these security properties:

- hash is a random oracle
- (sign, verify) is a UF-CMA-secure probabilistic signature
- the CDH assumption holds in the group  $G$

Now: Step-by-step presentation of `signedDH.ocv`

## Types and Probabilities for the Signature

Types define names for subsets of the bitstrings. The annotations restrict them on a high level.

```
type keyseed [large,fixed].
type pkey [bounded].
type skey [bounded].
type message [bounded].
type signature [bounded].
```

We define names for probabilities. They will appear in the final probability bound.

```
proba Psign.      (* breaking the UF-CMA property *)
proba Psigncoll. (* probability of collision between
                  independently generated keys *)
```



## Using the Macro: UF-CMA-secure Signature

```
expand UF_CMA_proba_signature(  
  (* types, to be defined outside the macro *)  
  keyseed,  
  pkey,  
  skey,  
  message,  
  signature,  
  (* names for functions defined by the macro *)  
  skgen,  
  pkgen,  
  sign,  
  verify,  
  (* probabilities, to be defined outside the macro *)  
  Psign,  
  Psigncoll  
).
```

In this example, we use a *probabilistic* signature. The macro makes this transparent for us, by defining the seed type and a **sign** wrapper function.

```
fun skgen(keyseed):skey.  
fun pkgen(keyseed):pkey.  
  
fun verify(message, pkey, signature): bool.  
fun sign_r(message, skey, sign_seed): signature.  
  
letfun sign(m: message, sk: skey) =  
  r <-R sign_seed; sign_r(m, sk, r).
```

The macro in CryptoVerif's default library defines the equation for correctness (not shown here).

## Diffie-Hellman Part I

```
type Z [large,bounded].  
type G [large,bounded].
```

```
proba PCollKey1.  
proba PCollKey2.
```

```
expand DH_proba_collision(  
  G,      (* type of group elements *)  
  Z,      (* type of exponents *)  
  g,      (* group generator *)  
  exp,    (* exponentiation function *)  
  exp',   (* exp. func. after transformation *)  
  mult,   (* func. for exponent multiplication *)  
  PCollKey1, (*  $g^{(\text{fresh } x)}$  collides with indep.  $Y$  *)  
  PCollKey2 (*  $g^{(\text{fr. } x * \text{fr. } y)}$  coll. w/ indep.  $Y$  *)  
).
```

CryptoVerif's default library comes with several macros for groups. We'll use a basic group in which some collision probabilities are negligible.

The macro defines the exponentiation function, a group generator, and equations for exponent multiplication. An extract:

```
fun exp(G, Z): G.  
  const g: G.
```

```
  fun mult(Z, Z): Z.  
    equation builtin commut(mult).
```

```
  equation forall a:G, x:Z, y:Z;  
    exp(exp(a, x), y) = exp(a, mult(x, y)).
```

## Diffie-Hellman Part III

Assumptions like CDH, DDH, GDH, ... must be instantiated with a separate macro. We use CDH, indicating the previously defined group:

```
proba pCDH. (* probability of breaking CDH in G *)  
expand CDH(G, Z, g, exp, exp', mult, pCDH).
```

This macro implements a multi-key version of (simplified presentation):

$$\text{Succ}_G^{\text{CDH}}(t) = \max_{\mathcal{A}} \Pr_{x,y \leftarrow \mathbb{Z}} [g^{xy} \leftarrow \mathcal{A}(g^x, g^y)] \text{ is negligible.}$$

## Random Oracle Part I – Definition

A random oracle is an idealized random function that returns

- an independent uniformly random value on new input,
- the same value than before on previously seen input.

To model this, *all* calls, also adversarial ones, must be observed by the game.

type hashfunction [fixed].

```
expand ROM_hash(  
    hashfunction, (* type for hash function choice *)  
    G,            (* type of input *)  
    key,          (* type of output *)  
    h,            (* name of hash function *)  
    hashoracle,   (* process defining the hash oracle *)  
    qH            (* parameter: number of calls *)  
).
```

The macro defines the hash function. The first parameter models the choice of the specific hash function: The adversary could call **hash**, but does not know the value the protocol uses for the 1st parameter.

```
fun hash(hashfunction, G): key.
```

The macro defines the oracle we must expose such that the adversary can use the RO:

```
param qH.
```

```
let hashoracle(hf: hashfunction) :=  
  foreach ih <= qH do  
    Ohash(x: G) :=  
      return(hash(hf, x)).
```

It allows **qH** calls, a parameter that will appear in the final probability formula.

## Random Oracle Part III – Usage

In the initial game, we sample a random hash function

```
hf <-R hashfunction;
```

and use it in each call of hash:

```
kA <- hash(hf, gab);
```

We must include the process defined by the macro, such that the adversary can access the random oracle for its own calls:

```
run hashoracle(hf)
```



When applying the RO assumption, CryptoVerif replaces each call of the hash function

```
foreach i <= N do (* ... *) hash(hf, x) (* ... *)
```

by an array lookup, comparing with *all* other inputs:

```
find j <= N suchthat defined(x[j], k[j]) && x = x[j]  
then k[j]  
else k <-R key; k
```

There will be one **find** branch per hash call.

In particular, the **hash** call in the **hashoracle** process will be replaced by a table lookup, comparing with all hash inputs used in the entire game.

## Setting up the Game

In the game setup, we create signature keypairs for the two honest parties. We can define functions (`letfun`) that `CryptoVerif` will inline.

```
letfun keygen() =  
  rk <-R keyseed;  
  sk <- skgen(rk);  
  pk <- pkgen(rk);  
  (sk, pk).
```

The initial game starts after the `process` keyword.

```
process  
  Ostart() :=  
    hf <-R hashfunction;  
    let (skA: skey, pkA: pkey) = keygen() in  
    let (skB: skey, pkB: pkey) = keygen() in  
    return(pkA, pkB);
```

## The Complete Main Process

```
param NA, NB, NK. (* number of calls *)

process
  Ostart() :=
    hf <-R hashfunction;
    let (skA: skey, pkA: pkey) = keygen() in
    let (skB: skey, pkB: pkey) = keygen() in
    return(pkA, pkB);
    (
      (foreach iA <= NA do run processA(hf, skA))
      |
      (foreach iB <= NB do run processB(hf, skB))
      |
      (foreach iK <= NK do run pki(pkA, pkB))
      |
      run hashoracle(hf) (* # of calls def. inside *)
    )
  )
```

# Public Key Infrastructure

We define a type for hosts, a list for (host, public key) tuples, and two honest hosts.

```
type host [bounded].  
table keys(host, pkey).  
const A, B: host. (* The two honest peers *)
```

We allow the adversary to register additional entries:

```
let pki(pkA: pkey, pkB: pkey) =  
  
  Opki(hostZ: host, pkZ: pkey) :=  
    if      hostZ = B then insert keys(B, pkB)  
  else if  hostZ = A then insert keys(A, pkA)  
  else                                           insert keys(hostZ, pkZ).
```

We will use `get keys(=hostX, pkX)` to retrieve X's key.

## Sequential Oracles in Processes

We expose one oracle for each protocol message.

OA1, OA3, OAfin, and OB2, OBfin can only be called in this order. A “session” identifier is implicit (the replication index).

```
let processA(...) =  
  OA1(...) :=  
    ...  
  return(...);
```

```
OA3(...) :=  
  ...  
  return(...);
```

```
OAfin(...) :=  
  ...  
  return(...).
```

```
let processB(...) =  
  OB2(...) :=  
    ...  
  return(...);
```

```
OBfin(...) :=  
  ...  
  return(...)
```

## 1st and 2nd Message

Creating the 1st message. The adversary chooses A's peer.

```
let processA(hf:hashfunction, skA:skey) =  
  OA1(hostX: host) :=  
    a <-R Z;    ga <- exp(g,a);  
    return(A, hostX, ga);
```

Consuming the 1st and creating the 2nd message. B only continues if the message is for B: =B. Event **beginB** is recorded.

```
let processB(hf:hashfunction, skB:skey) =  
  OB2(hostY: host, =B, ga: G) :=  
    b <-R Z;    gb <- exp(g,b);  
    sig <- sign(msg2(hostY, B, ga, gb), skB);  
    event beginB(hostY, B, ga, gb);  
    return(hostY, B, gb, sig);
```

## 2nd and 3rd Message

```
let processB(hf:hashfunction, skB:skey) =  
  OB2(hostY:host, =B, ga:G) :=  
    (* ... *)  
  return(hostY, B, gb, sig);
```

If A can verify the signature, event endA is recorded.

```
let processA(hf:hashfunction, skA:skey) =  
  (* ... *)  
  
  OA3(=A, =hostX, gb: G, s: signature) :=  
    get keys(=hostX, pkX) in  
    if verify(msg2(A, hostX, ga, gb), pkX, s) then  
      gab <- exp(gb, a);    kA <- hash(hf, gab);  
      sig <- sign(msg3(A, hostX, ga, gb), skA);  
      event endA(A, hostX, ga, gb);  
      return(sig);
```

### 3rd Message and Finish

If B can verify the signature, event **endB** is recorded.

```
OBfin(s:signature) :=  
  get keys(=hostY, pkY) in  
  if verify(msg3(hostY, B, ga, gb), pkY, s) then  
    gab <- exp(ga, b);  
    kB <- hash(hf, gab);  
    event endB(hostY, B, ga, gb);
```

We want to prove secrecy only in case the two honest peers interacted. Only in this case we assign the shared secret to another variable.

```
if hostY = A then (  
  keyB:key <- kB  
) else  
  return(kB).
```



## Finish on A's Side

We could have merged that into OA3, but it is clearer this way.

```
OAfin() :=  
  if hostX = B then (keyA:key <- kA)  
  else return(kA).
```

Now we have variables **keyA** and **keyB** that are only defined for honest sessions, for which we want to prove key secrecy. Thus, we can ask CryptoVerif to prove:

```
query secret keyA.  
query secret keyB.
```

Note that this way, *all* honest sessions are “test” sessions.

## Definition: Key Secrecy for $k_A$ (and similar $k_B$ ) ...

[1]

... if an adversary has a negligible probability of distinguishing keys  $k_A$  from uniformly random bitstrings of same length:

## Definition: Key Secrecy for $k_A$ (and similar $k_B$ ) ...

[1]

... if an adversary has a negligible probability of distinguishing keys  $k_A$  from uniformly random bitstrings of same length:

$$\text{Succ}_{\text{SDH}}^{\text{key-secrecy}, k_A}(t, n_A, n_B, n_K, q_H) = \max_{\mathcal{A}} | \Pr[\mathcal{G}_{\text{real}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}_{\text{random}}(\mathcal{A}) \Rightarrow 1] |$$

- where  $\mathcal{G}_{\text{real}}$  is the original game, and
- in  $\mathcal{G}_{\text{random}}$ , the keys  $k_A$  are replaced by independent uniformly random bitstrings of the same length

## Definition: Key Secrecy for $k_A$ (and similar $k_B$ ) ...

[1]

... if an adversary has a negligible probability of distinguishing keys  $k_A$  from uniformly random bitstrings of same length:

$$\text{Succ}_{\text{SDH}}^{\text{key-secrecy}, k_A}(t, n_A, n_B, n_K, q_H) = \max_{\mathcal{A}} | \Pr[\mathcal{G}_{\text{real}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}_{\text{random}}(\mathcal{A}) \Rightarrow 1] |$$

- where  $\mathcal{G}_{\text{real}}$  is the original game, and
- in  $\mathcal{G}_{\text{random}}$ , the keys  $k_A$  are replaced by independent uniformly random bitstrings of the same length

and where  $\mathcal{A}$

- runs in time at most  $t$
- starts at most  $n_A$  sessions for  $A$ , and at most  $n_B$  for  $B$
- registers at most  $n_K$  public keys (incl.  $A$  and  $B$ )
- calls the hash oracle at most  $q_H$  times.

## Correspondence Queries

Events need to be declared:

```
event endA(host, host, G, G).  
event beginB(host, host, G, G).  
event endB(host, host, G, G).
```

*A* can authenticate *B*, even if any shared secret leaks:

```
query y: G, x: G;  
  inj-event(endA(A, B, x, y))  
  ==> inj-event(beginB(A, B, x, y))  
  public_vars keyA, keyB.
```

*B* can authenticate *A*, even if any shared secret leaks:

```
query y: G, x: G;  
  inj-event(endB(A, B, x, y))  
  ==> inj-event(endA(A, B, x, y))  
  public_vars keyA, keyB.
```

## Definition: Authentication of $A$ (and similar for $B$ ) ...

[2]

... if an adversary has a negligible probability of producing a sequence of events that violates the correspondence property:

## Definition: Authentication of $A$ (and similar for $B$ ) ...

[2]

... if an adversary has a negligible probability of producing a sequence of events that violates the correspondence property:

$$\text{Succ}_{\text{SDH}}^{\text{auth}, A}(t, n_A, n_B, n_K, q_H) = \max_{\mathcal{A}} \Pr \left[ \begin{array}{l} \mathcal{A}^{O_{\text{start}}, OA, OB, O_{\text{pk}}, OH} : \mathcal{A} \text{ produces a sequence of events} \\ \text{such that not every } \text{end}_B(A, B, g^a, g^b) \text{ is preceded} \\ \text{by a distinct } \text{end}_A(A, B, g^a, g^b) \end{array} \right]$$

## Definition: Authentication of $A$ (and similar for $B$ ) ...

[2]

... if an adversary has a negligible probability of producing a sequence of events that violates the correspondence property:

$$\text{Succ}_{\text{SDH}}^{\text{auth}, A}(t, n_A, n_B, n_K, q_H) = \max_{\mathcal{A}} \Pr \left[ \begin{array}{l} \mathcal{A}^{O_{\text{start}}, OA, OB, O_{\text{pk}}, OH} : \mathcal{A} \text{ produces a sequence of events} \\ \text{such that not every } \text{end}_B(A, B, g^a, g^b) \text{ is preceded} \\ \text{by a distinct } \text{end}_A(A, B, g^a, g^b) \end{array} \right]$$

where  $\mathcal{A}$

- runs in time at most  $t$
- starts at most  $n_A$  sessions for  $A$ , and at most  $n_B$  for  $B$
- registers at most  $n_K$  public keys (incl.  $A$  and  $B$ )
- calls the hash oracle at most  $q_H$  times.



## Proof and Result

```
(* demo *)
```

## Interactive Mode

Include `interactive` in the proof environment to start the interactive mode:

```
proof {  
  interactive  
}
```

- `out_game "filename"` outputs the current game. Use a `.ocv` extension such that your editor highlights the syntax.
- `crypto assumption(function)` applies the assumption to the function. Example: `crypto rom(hash)`
- `success` tries to prove the queries
- `simplify` tries to simplify the current game
- `quit` leaves interactive mode and continues non-interactively.
- Ctrl+D ends the programme

# What We Covered Today

- Introduction to the syntax and semantics of games
- Model simple primitives and protocols
- Use macros from the default library: symmetric encryption, MAC, signature, random oracle, basic Diffie-Hellman
- Basic interactive interaction with CryptoVerif
- Prove secrecy and correspondence properties
- Read the final result

## Next Steps with CryptoVerif

- Try the exercises and reach us on VeriCrypt's Zulip during the next days
  - syntax highlighting is available for Vim and Emacs
- The reference manual is in `docs/manual.pdf`
- More examples are in the directory `examples`
  - beware, spoilers for the exercises
  - look for `.ocv` files, they use the oracle syntax presented in this tutorial. (`.pcv` and `.cv` use the *channel* frontend)
- Subscribe to the mailinglist (low activity)  
<https://sympa.inria.fr/sympa/subscribe/cryptoverif>

# References

- References to the case studies are in the slides of Part I
- References for how CryptoVerif proves (titles are clickable links)
  - Secrecy:

[1] Bruno Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. IEEE Transactions on Dependable and Secure Computing, 5(4):193-207, October-December 2008. Special issue IEEE Symposium on Security and Privacy 2006.
  - Correspondence:

[2] Bruno Blanchet. Computationally Sound Mechanized Proofs of Correspondence Assertions. Cryptology ePrint Archive, Report 2007/128.