

# CryptoVerif: Mechanising Game-Based Proofs

## Part II

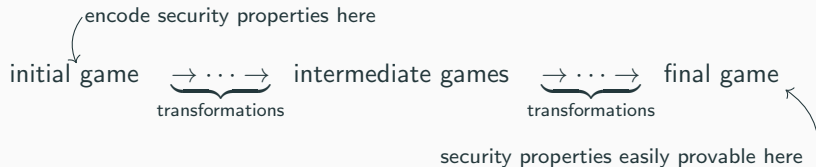
---

Benjamin Lipp

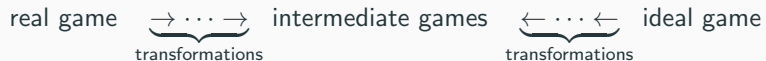
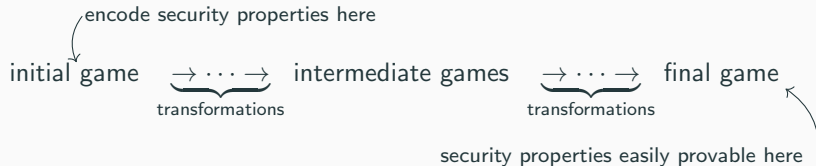
June 6, 2023

Max Planck Institute for Security and Privacy

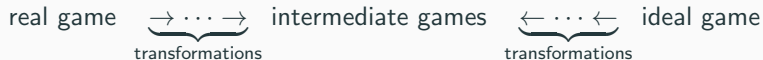
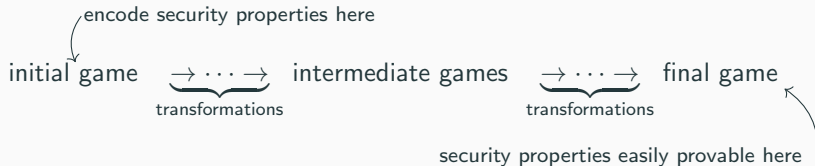
## Recap: CryptoVerif Formalizes Game-Based Proofs



## Recap: CryptoVerif Formalizes Game-Based Proofs

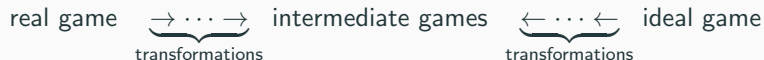
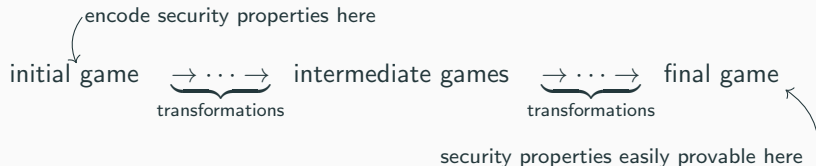


## Recap: CryptoVerif Formalizes Game-Based Proofs



- CryptoVerif *constructs* a sequence of computationally indistinguishable games
- built-in proof strategy, and detailed guidance by user

## Recap: CryptoVerif Formalizes Game-Based Proofs



- CryptoVerif *constructs* a sequence of computationally indistinguishable games
- built-in proof strategy, and detailed guidance by user
- supports indistinguishability, secrecy, authentication properties
- computes exact security probability bound

## What to Expect from Part II

A more complex example, a protocol with multiple messages:

Signed Diffie-Hellman, a 2-party Authenticated Key Exchange protocol

What's new?

- model a hash function as a random oracle
- use a Computational Diffie-Hellman (CDH) assumption
- prove key secrecy in a protocol
- prove authentication properties using correspondences between events
- model a Public-Key Infrastructure using a list (`table` in `CryptoVerif`)

# Cryptographic Building Blocks

---

# Cryptographic Building Block: Hash Function

## Hash Function

$\text{hash} : \{0, 1\}^* \rightarrow \{0, 1\}^{\text{hashlen}}.$

Example:

$k \leftarrow \text{hash}(m)$

Intuition: for different inputs, outputs are uniformly random and independent of each other.



# Cryptographic Building Block: Signature

## Cryptographic Signature

$sk, pk \xleftarrow{\$} \text{keygenSig}()$

$\sigma \xleftarrow{\$} \text{sign}(m, sk)$

$b \leftarrow \text{verify}(m, pk, \sigma)$  returns 1 iff  $\sigma$  is a correct signature

Intuition: it is hard to forge a signature

# Cryptographic Building Block: Diffie-Hellman

## Diffie-Hellman Non-Interactive Key Exchange

For simplicity, in a prime-order cyclic group  $G = (\mathbb{Z}/p\mathbb{Z})^*$  of order  $p$  with generator  $g$ .

private keys:  $a, b \xleftarrow{\$} Z = \{1, \dots, p-1\}$

public keys:  $g^a \bmod p, g^b \bmod p \in G$ . ( $g^a, g^b$  in short)

DH shared secret:  $(g^a)^b \bmod p = (g^b)^a \bmod p = g^{ab} \bmod p$

Intuition: Knowing only the public keys, it is hard to recognize or compute the DH shared secret

**Our Case Study:**  
**The Signed Diffie-Hellman**  
**Protocol**

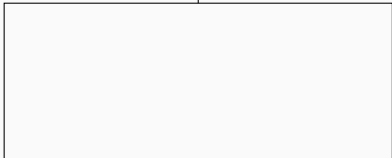
---

knows  $sk_A, pk_B$

$A$

knows  $sk_B, pk_A$

$B$



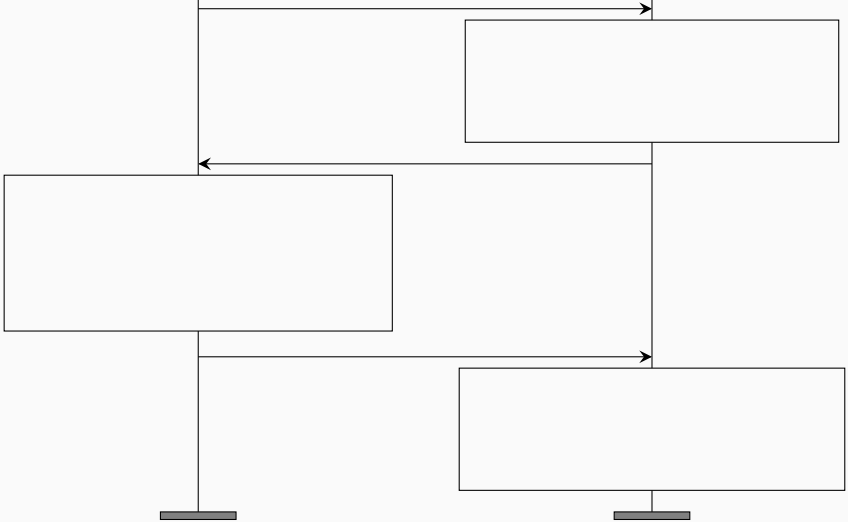
knows  $sk_A, pk_B$

$A$

$a \leftarrow \$ Z$

knows  $sk_B, pk_A$

$B$



knows  $sk_A, pk_B$

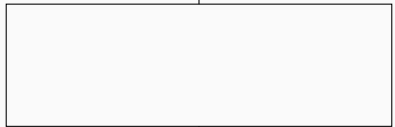
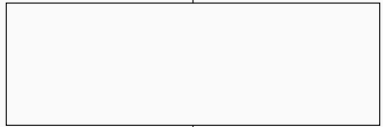
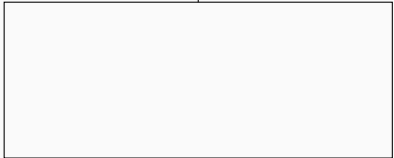
$A$

$a \leftarrow \$ Z$

knows  $sk_B, pk_A$

$B$

$A, B, g^a$



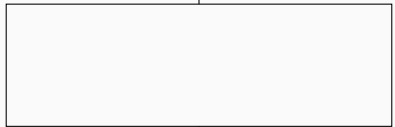
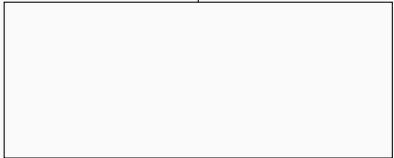
knows  $sk_A, pk_B$   
 $A$

$a \leftarrow \$Z$

$A, B, g^a$

knows  $sk_B, pk_A$   
 $B$

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



knows  $sk_A, pk_B$   
**A**

knows  $sk_B, pk_A$   
**B**

$a \leftarrow \$Z$

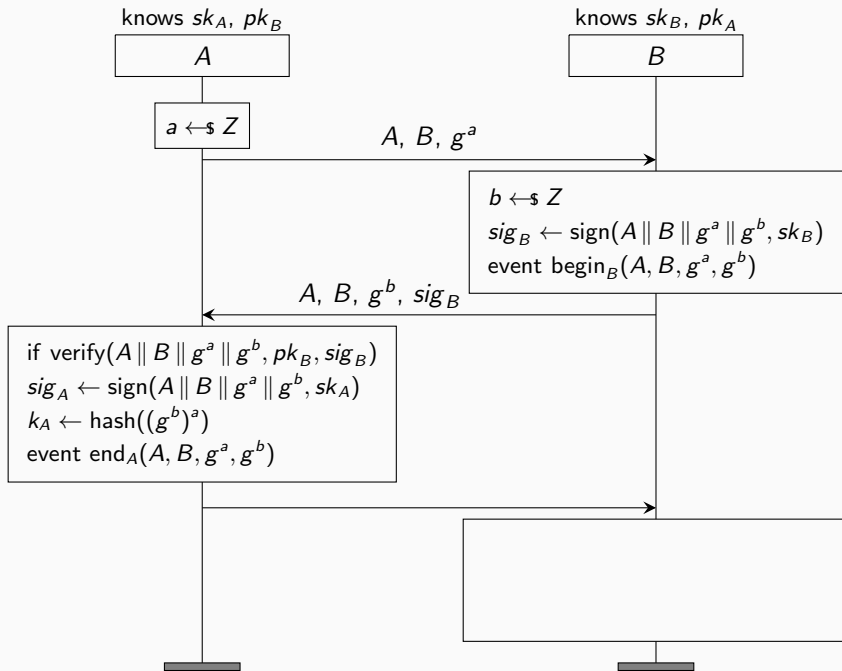
$A, B, g^a$

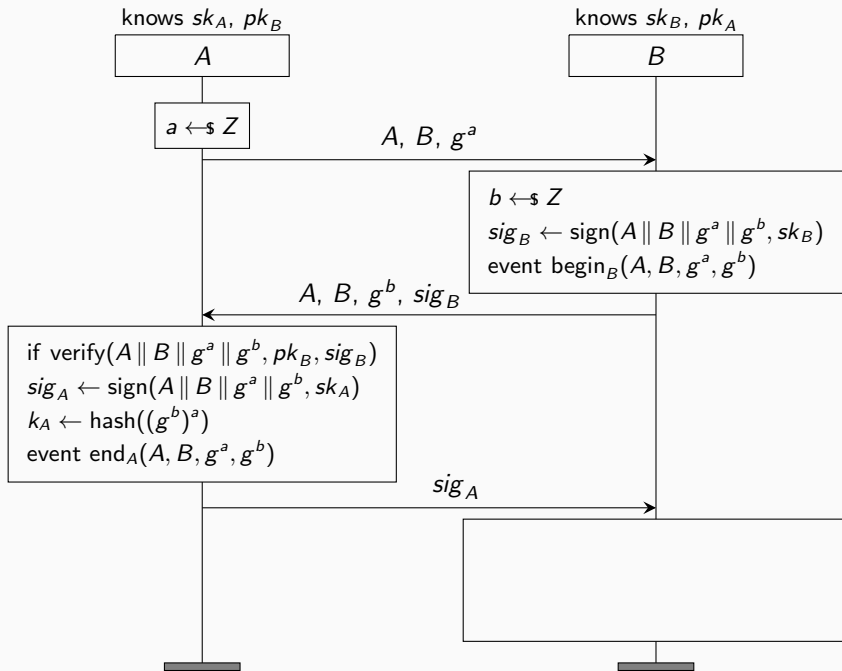
$b \leftarrow \$Z$   
 $sig_B \leftarrow \text{sign}(A \parallel B \parallel g^a \parallel g^b, sk_B)$   
event  $\text{begin}_B(A, B, g^a, g^b)$

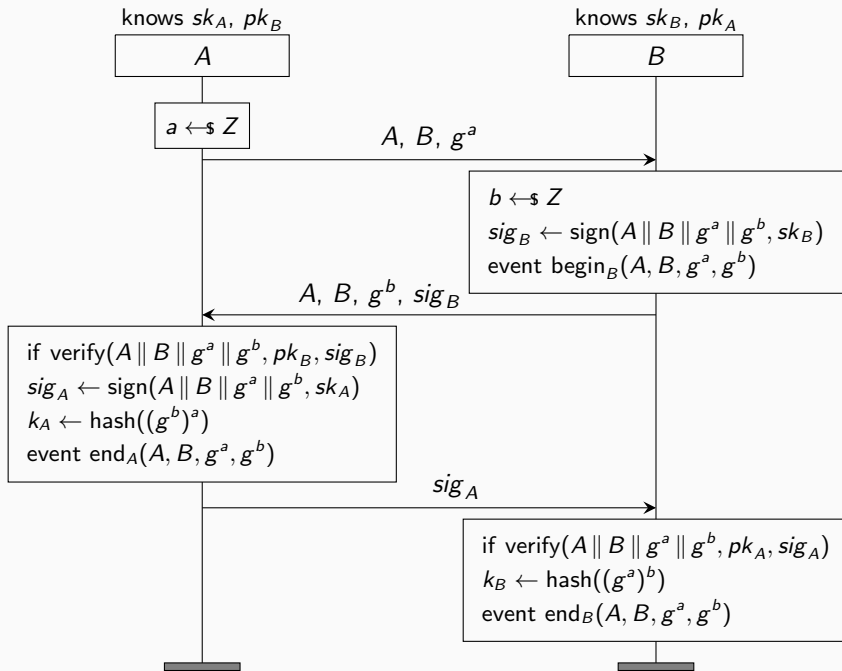
$A, B, g^b, sig_B$











## Signed Diffie-Hellman: Security Properties

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

## Signed Diffie-Hellman: Security Properties

- The shared secrets  $k_A$  and  $k_B$  are secret  
(indistinguishable from random bitstrings of equal length)
- If  $A$  is convinced to have concluded a session with  $B$  using ephemerals  $g^a, g^b$ , then  $B$  actually started such a session

## Signed Diffie-Hellman: Security Properties

- The shared secrets  $k_A$  and  $k_B$  are secret  
(indistinguishable from random bitstrings of equal length)
- If  $A$  is convinced to have concluded a session with  $B$  using ephemerals  $g^a, g^b$ , then  $B$  actually started such a session
- If  $B$  is convinced to have concluded a session with  $A$  using ephemerals  $g^a, g^b$ , then  $A$  is likewise convinced

# Cryptographic Assumptions

---

# 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  $G$



# Random Oracle as Ideal Model for Hash Functions

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, adversarial calls are observed by the security game through an oracle.

Definitional rewriting step done by CryptoVerif:

$$\frac{\text{ROM}_b}{\mathcal{L} \leftarrow \emptyset}$$
$$\text{return } \mathcal{A}^{\text{hash}_b}()$$
$$\frac{\text{hash}_0(m)}{\text{return hash}(m)}$$
$$\frac{\text{hash}_1(m)}{\text{if } \exists k : (m, k) \in \mathcal{L}}$$
$$\quad \text{return } k$$
$$\text{else}$$
$$\quad k \xleftarrow{\$} \{0, 1\}^{\text{hashlen}}$$
$$\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{(m, k)\}$$
$$\quad \text{return } k$$

## Random Oracle – Preamble in CryptoVerif

Using a random oracle in CryptoVerif:

```
type hashfunction [fixed].
```

```
expand ROM_hash(  
  hashfunction, (* type for hash function choice *)  
  G,            (* type of input *)  
  key,          (* type of output *)  
  hash,         (* 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 – Usage

In the setup of 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)
```

The hash function might be called within a replicated oracle:

```
foreach i <= N do (* ... *) kA <- hash(hf, gab) (* ... *)
```

Variables inside a replication are implicitly defined as arrays. Values are accessible via the replication index: `gab[i]`, `kA[i]`

The hash function might be called within a replicated oracle:

```
foreach i <= N do (* ... *) kA <- hash(hf, gab) (* ... *)
```

Variables inside a replication are implicitly defined as arrays. Values are accessible via the replication index: `gab[i]`, `kA[i]`

An array lookup using `find` can access specific values. Here is how to locally model the call by a random oracle (assuming that there is only this one call to `hash`):

```
foreach i <= N do (* ... *)  
  
  (find j <= N suchthat defined(gab[j], kA[j]) && gab = gab[j]  
   then kA[j]  
   else kA <-R key; kA)  
  
(* ... *)
```

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

When applying the RO assumption, CryptoVerif replaces each call of the hash function by an array lookup, comparing with *all* other inputs:

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

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

```
foreach i <= N do
  (* ... *)
  kA <- hash(hf, gab)
  (* ... *)

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



```
foreach i <= N do
  (* ... *)
  kA <- hash(hf, gab) (* before rewriting *)
  (* ... *)

let hashoracle(hf: hashfunction) :=
  foreach ih <= qH do
    Ohash(x: G) :=
      find j <= qH suchthat defined(x[j], k[j]) && x = x[j] then
        return(k[j])
      else find i <= N suchthat
        defined(gab[i], kA[i]) && x = gab[i] then
          return(kA[i])
      else
        k <-R key;
        return(k).
```

# UF-CMA-Secure Probabilistic Signature

- Unforgeability under Chosen Message Attack (UF-CMA)
- Security notion implemented by the appropriate CryptoVerif macro (simplified), where the adversary advantage

$$\text{Adv}_{\text{sign}}^{\text{UF-CMA}}(\mathcal{A}) = |\Pr[\text{UF-CMA}_0(\mathcal{A}) \Rightarrow 1] - \Pr[\text{UF-CMA}_1(\mathcal{A}) \Rightarrow 1]| \text{ is negligible.}$$

Oracle Sign( $m$ )

$\mathcal{L} \leftarrow \mathcal{L} \cup \{m\}$

$\sigma \xleftarrow{\$} \text{sign}(m, \text{sk}(r))$

**return**  $\sigma$

Oracle Verify<sub>0</sub>( $m, \sigma$ )

**return**  $\text{verify}(m, \text{pk}(r), \sigma)$

Oracle Verify<sub>1</sub>( $m, \sigma$ )

**return**  $m \in \mathcal{L} \wedge \text{verify}(m, \text{pk}(r), \sigma)$

UF-CMA<sub>b</sub>

$r \xleftarrow{\$} \mathcal{K}$

$\mathcal{L} \leftarrow \emptyset$

**return**  $\mathcal{A}^{\text{Sign}, \text{Verify}_b}(\text{pk}(r))$

## 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).  
  
equation forall m: message, r: keyseed, r2: sign_seed;  
  verify(m, pkgen(r), sign_r(m, skgen(r), r2)) = true.
```

# The Computational Diffie-Hellman (CDH) Assumption

- computing  $g^{xy}$  from  $g^x$  and  $g^y$  is hard
- a comparison of an adversary-computed value with  $g^{xy}$  is indistinguishable from **false** for the adversary
- using CDH in a game-rewriting step in CryptoVerif, in a simplified single-key version, where the adversary advantage

$$\text{Adv}_G^{\text{CDH}}(\mathcal{A}) = | \Pr [\text{CDH}_0(\mathcal{A}) \Rightarrow 1] - \Pr [\text{CDH}_1(\mathcal{A}) \Rightarrow 1] | \quad \text{is negligible.}$$

CDH<sub>b</sub>

$x, y \xleftarrow{s} Z$

**return**  $\mathcal{A}^{\text{DDH}_b}(g^x, g^y)$

DDH<sub>0</sub>(c)

**return**  $c = g^{xy}$

DDH<sub>1</sub>(c)

**return false**

## Diffie-Hellman Part I

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

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

```
proba PCollKey1.
```

```
proba PCollKey2.
```

CryptoVerif's default library comes with several macros for groups.

We'll use a basic group in which some collision probabilities are negligible.

```
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.  $Y$  *)  
) .
```

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

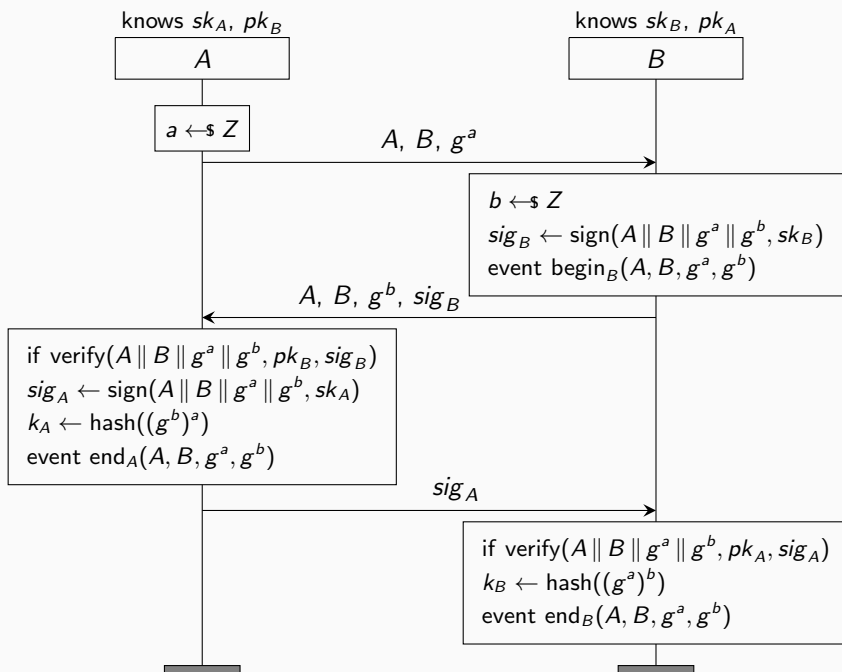
Assumptions like CDH, DDH, GDH, ... must be instantiated with a separate macro. We use CDH, indicating the previously defined group:

```
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 the version presented on the slides.

# Semantics of the Security Queries

---



## 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 $k_B$ ) ...

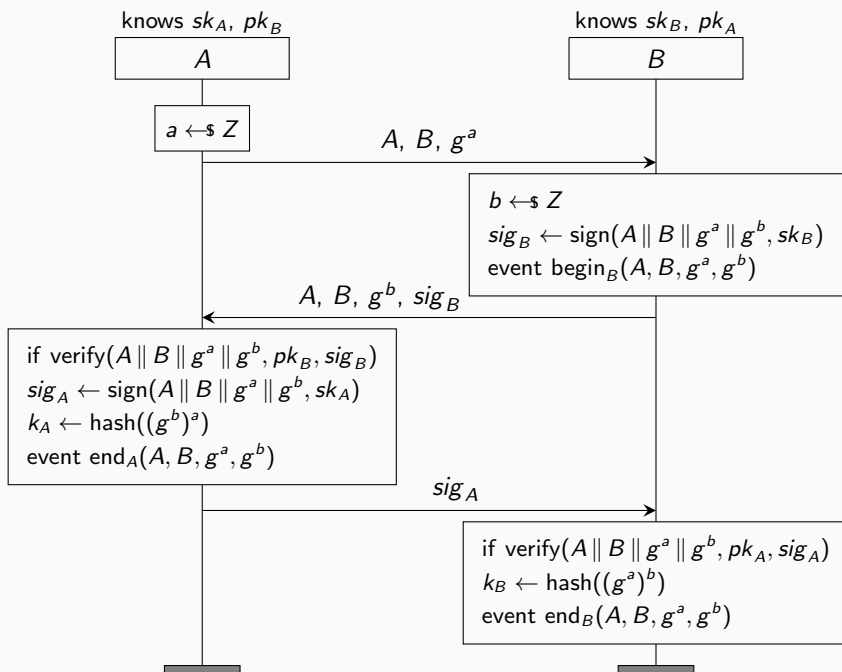
[1]

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

$$\text{Adv}_{\text{signedDH}}^{\text{key-secrecy}, k_A}(\mathcal{A}) = | \Pr[\mathcal{G}_{\text{real}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}_{\text{random}}(\mathcal{A}) \Rightarrow 1] |$$

- where  $\mathcal{G}_{\text{real}}$  is the original game (the initial game modeled in CryptoVerif), and
- in  $\mathcal{G}_{\text{random}}$  (implicitly reasoned about by CryptoVerif), the keys  $k_A$  are replaced by independent uniformly random bitstrings of the same length.

This is different from usual pen-and-paper security notions where there is only one test session; here, all (honest) sessions are test sessions!



## 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:

$$\text{Adv}_{\text{signedDH}}^{\text{auth}, A}(\mathcal{A}) = \Pr \left[ \begin{array}{l} \mathcal{A}^{O_{\text{start}}, O_A, O_B, O_{\text{pk}}, O_H} : \mathcal{A} \text{ produces a sequence of events} \\ \text{such that not every } \text{end}_B(A, B, g^a, g^b) \text{ is preceded} \\ \text{by a distinct } \text{end}_A(A, B, g^a, g^b) \end{array} \right]$$



## Next Exercise Session

(\* It's your turn \*)

You should follow *instructions-practical-session-2.pdf* at:

<https://github.com/charlie-j/summer-school-2023/>

Feel free to refer to the cheatsheet, and to the slides of both sessions, and to ask questions!

## Backup Slides

---

# 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