

# CryptoVerif: A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet

CNRS, École Normale Supérieure, INRIA, Paris

September 2009

# Introduction

Two models for security protocols:

- **Computational model:**
  - messages are bitstrings
  - cryptographic primitives are functions from bitstrings to bitstrings
  - the adversary is a probabilistic polynomial-time Turing machine

Proofs are done manually.

- **Formal model** (so-called “Dolev-Yao model”):
  - cryptographic primitives are ideal blackboxes
  - messages are terms built from the cryptographic primitives
  - the adversary is restricted to use only the primitives

Proofs can be done automatically.

Our goal: achieve **automatic provability** under the realistic **computational** assumptions.

# Introduction

Two approaches for the automatic proof of cryptographic protocols in a computational model:

- **Indirect approach:**

- 1) Make a Dolev-Yao proof.

- 2) Use a theorem that shows the soundness of the Dolev-Yao approach with respect to the computational model.

Pioneered by Abadi and Rogaway; pursued by many others.

- **Direct approach:**

Design automatic tools for proving protocols in a computational model.

Approach pioneered by Laud.

# Advantages and drawbacks

The indirect approach allows more reuse of previous work, but it has limitations:

- **Hypotheses** have to be added to make sure that the computational and Dolev-Yao models coincide.
- The **allowed cryptographic primitives** are often limited, and only ideal, not very practical primitives can be used.
- Using the Dolev-Yao model is actually a (big) **detour**;  
The computational definitions of primitives fit the computational security properties to prove.  
They do not fit the Dolev-Yao model.

We decided to focus on the direct approach.

# An automatic prover

We have implemented an **automatic prover** CryptoVerif:

- proves **secrecy** and **correspondence** (including authentication) properties.
- provides a **generic** method for specifying properties of **cryptographic primitives** which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, ...
- works for  **$N$  sessions** (polynomial in the security parameter), with an **active adversary**.
- gives a bound on the **probability** of an attack (exact security).

# Produced proofs

We use Shoup's and Bellare&Rogaway's **game hopping** method.

The proof is a **sequence of games**:

- The first game is the **real protocol**.
- One goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. Between consecutive games, the difference of probability of success of an attack is negligible.
- The last game is **"ideal"**: the security property is obvious from the form of the game.  
(The advantage of the adversary is typically 0 for this game.)

# Input and output of the tool

- 1 Prepare the input file containing
  - the specification of the **protocol** to study (initial game),
  - the **security assumptions** on the cryptographic primitives,
  - the **security properties** to prove.
- 2 Run CryptoVerif
- 3 CryptoVerif outputs
  - the **sequence of games** that leads to the proof,
  - a **succinct explanation** of the transformations performed between games,
  - an upper bound of the **probability** of success of an attack.

# Process calculus for games

Games are formalized in a **process calculus**:

- It is adapted from the pi calculus.
- The semantics is **purely probabilistic** (no non-determinism).
- All processes run in **polynomial time**:
  - polynomial number of copies of processes,
  - length of messages on channels bounded by polynomials.

This calculus is inspired by:

- the calculus of [Lincoln, Mitchell, Mitchell, Scedrov, 1998],
- the calculus of [Laud, 2005].

# Example

$$A \rightarrow B : e = \{k'\}_k, \text{mac}(e, mk) \quad k' \text{ fresh}$$

$A$  sends to  $B$  a fresh key  $k'$  encrypted under authenticated encryption, implemented as encrypt-then-MAC.

$k'$  should remain secret.

# Example (initialization)

$$A \rightarrow B : e = \{k'\}_k, \text{mac}(e, mk) \quad k' \text{ fresh}$$

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$   
 $\mathbf{new} \ r' : \text{mkeyseed}; \mathbf{let} \ mk : \text{mkey} = \text{mkgen}(r') \mathbf{in} \ \bar{c}\langle \rangle; (Q_A \mid Q_B)$

Initialization of keys:

- 1 The process  $Q_0$  waits for a message on channel  $\text{start}$  to start running. The adversary triggers this process.
- 2  $Q_0$  **generates encryption and MAC keys**,  $k$  and  $mk$  respectively, using the key generation algorithms  $\text{kgen}$  and  $\text{mkgen}$ .
- 3  $Q_0$  returns control to the adversary by the output  $\bar{c}\langle \rangle$ .  
 $Q_A$  and  $Q_B$  represent the actions of  $A$  and  $B$  (see next slides).

# Example (role of A)

$$A \rightarrow B : e = \{k'\}_k, \text{mac}(e, mk) \quad k' \text{ fresh}$$

$$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$$

$$\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$$

$$\overline{c_A} \langle e, \text{mac}(e, mk) \rangle$$

Role of A:

- ①  $!^{i \leq n}$  represents  $n$  copies, indexed by  $i \in [1, n]$   
The protocol can be run  $n$  times (polynomial in the security parameter).
- ② The process is triggered when a message is sent on  $c_A$  by the adversary.
- ③ The process **chooses a fresh key  $k'$  and sends the message** on channel  $c_A$ .

# Example (role of $B$ )

$$A \rightarrow B : e = \{k'\}_k, \text{mac}(e, mk) \quad k' \text{ fresh}$$

$$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$$

**if**  $\text{verify}(e', mk, ma)$  **then**

**let**  $i_{\perp}(k2b(k'')) = \text{dec}(e', k)$  **in**  $\overline{c_B} \langle \rangle$

Role of  $B$ :

- ①  $n$  copies, as for  $Q_A$ .
- ② The process  $Q_B$  waits for the message on channel  $c_B$ .
- ③ It verifies the MAC, decrypts, and stores the key in  $k''$ .

# Example (summary)

$$A \rightarrow B : e = \{k'\}_k, \text{mac}(e, mk) \quad k' \text{ fresh}$$

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$   
 $\mathbf{new} \ r' : \text{mkeyseed}; \mathbf{let} \ mk : \text{mkey} = \text{mkgen}(r') \mathbf{in} \ \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$   
 $\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$   
 $\overline{c}_A\langle e, \text{mac}(e, mk) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
 $\mathbf{if} \ \text{verify}(e', mk, ma) \mathbf{then}$   
 $\mathbf{let} \ i_{\perp}(k2b(k'')) = \text{dec}(e', k) \mathbf{in} \ \overline{c}_B\langle \rangle$

# Security assumptions on primitives

The most frequent cryptographic primitives are **already specified in a library**. The user can use them without redefining them.

In the example:

- The MAC is **UF-CMA** (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).

# Security assumptions on primitives

The most frequent cryptographic primitives are **already specified in a library**. The user can use them without redefining them.

In the example:

- The MAC is **UF-CMA** (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is **IND-CPA** (indistinguishable under chosen plaintext attacks). An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

# Security assumptions on primitives

The most frequent cryptographic primitives are **already specified in a library**. The user can use them without redefining them.

In the example:

- The MAC is **UF-CMA** (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is **IND-CPA** (indistinguishable under chosen plaintext attacks). An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.
- All keys have the **same length**: **forall**  $y : key; Z(k2b(y)) = Z_k$ .

# Security properties to prove

In the example:

- **One-session secrecy** of  $k''$ : each  $k''$  is indistinguishable from a random number.
- **Secrecy** of  $k''$ : the  $k''$  are indistinguishable from independent random numbers.

# Demo

Demo

# Process calculus for games: terms

$M ::=$	terms
$x, y, z, x[M_1, \dots, M_n]$	variable
$f(M_1, \dots, M_n)$	function application

Function symbols  $f$  correspond to functions computable by polynomial-time deterministic Turing machines.

# Process calculus for games: processes

$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq N} Q$	replication $N$ times
<b>newChannel</b> $c; Q$	restriction for channels
$c(x_1 : T_1, \dots, x_m : T_m); P$	input
$P ::=$	output process
$\bar{c}\langle M_1, \dots, M_m \rangle; Q$	output
<b>new</b> $x : T; P$	random number generation (uniform)
<b>let</b> $x : T = M$ <b>in</b> $P$	assignment
<b>if</b> $M$ <b>then</b> $P$ <b>else</b> $P'$	conditional
<b>find</b> $j \leq N$ <b>suchthat</b> <b>defined</b> ( $x[j], \dots$ ) $\wedge M$ <b>then</b> $P$ <b>else</b> $P'$	array lookup

# Arrays

Arrays replace **lists** often used in cryptographic proofs.

They avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.

A variable defined under a replication is implicitly an **array**:

$$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k'[i] : \mathit{key}; \mathbf{new} \ r''[i] : \mathit{coins};$$

$$\mathbf{let} \ e[i] : \mathit{bitstring} = \mathit{enc}(k2b(k'[i]), k, r''[i]) \mathbf{in}$$

$$\overline{c_A} \langle e[i], \mathit{mac}(e[i], mk) \rangle$$

Requirements:

- Only variables with the current indices can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indices.

(**if** ... **then let**  $x = M$  **in**  $P$  **else let**  $x = M'$  **in**  $P'$  is ok)

So each array cell can be **assigned at most once**.

# Arrays (continued)

**find** performs an **array lookup**:

$$!i \leq N \dots \mathbf{let} \ x = M \ \mathbf{in} \ P$$

$$| !i' \leq N' \ c(y : T) \mathbf{find} \ j \leq N \ \mathbf{suchthat} \ \mathbf{defined}(x[j]) \wedge y = x[j] \ \mathbf{then} \dots$$

Note that **find** is here used outside the scope of  $x$ .

This is the only way of getting access to values of variables in other sessions.

When several array elements satisfy the condition of the **find**, the returned index is chosen randomly, with uniform probability.

# Indistinguishability as observational equivalence

Two processes (games)  $Q_1$ ,  $Q_2$  are **observationally equivalent** when the adversary has a negligible probability of distinguishing them:

$$Q_1 \approx Q_2$$

In the formal definition, the adversary is represented by an acceptable evaluation context  $C ::= [] \quad C \mid Q \quad Q \mid C \quad \mathbf{newChannel} \ c; C$ .

- Observational equivalence is an equivalence relation.
- It is **contextual**:  $Q_1 \approx Q_2$  implies  $C[Q_1] \approx C[Q_2]$  where  $C$  is any acceptable evaluation context.

# Proof technique

We transform a game  $G_0$  into an observationally equivalent one using:

- **observational equivalences**  $L \approx R$  given as **axioms** and that come from security assumptions on primitives. These equivalences are used inside a context:

$$G_1 \approx C[L] \approx C[R] \approx G_2$$

- **syntactic transformations**: simplification, expansion of assignments, ...

We obtain a **sequence of games**  $G_0 \approx G_1 \approx \dots \approx G_m$ , which implies  $G_0 \approx G_m$ .

If some equivalence or trace property holds with overwhelming probability in  $G_m$ , then it also holds with overwhelming probability in  $G_0$ .

# MACs: security definition

A MAC scheme:

- (Randomized) key generation function *mkgen*.
- MAC function *mac(m, k)* takes as input a message *m* and a key *k*.
- Verification function *verify(m, k, t)* such that
 
$$\text{verify}(m, k, \text{mac}(m, k)) = \text{true}.$$

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the mac.

More formally, an adversary  $\mathcal{A}$  that has oracle access to *mac* and *verify* has a negligible probability of forging a MAC (UF-CMA):

$$\Pr[\text{verify}(m, k, t) \mid k \xleftarrow{R} \text{mkgen}; (m, t) \leftarrow \mathcal{A}^{\text{mac}(\cdot, k), \text{verify}(\cdot, k, \cdot)}]$$

is negligible, when the adversary  $\mathcal{A}$  has not called the *mac* oracle on message *m*.

# MACs: intuitive implementation

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- so when verifying a MAC with  $verify(m, k, t)$  and  $k \stackrel{R}{\leftarrow} mkgen$  is used only for generating and verifying MACs, the verification can succeed **only if  $m$  is in the list (array) of messages whose  $mac$  has been computed** by the protocol
- so we can replace a call to  $verify$  with an array lookup:  
if the call to  $mac$  is  $mac(x, k)$ , we replace  $verify(m, k, t)$  with

**find  $j \leq N$  suchthat  $defined(x[j]) \wedge$   
 $(m = x[j]) \wedge verify(m, k, t)$  then true else false**

# MACs: formal implementation

$$\text{verify}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \mathbf{true}$$

$$!^{N''} \mathbf{new} r : \text{mkeyseed}; ($$

$$!^N (x : \text{bitstring}) \rightarrow \text{mac}(x, \text{mkgen}(r)),$$

$$!^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow \text{verify}(m, \text{mkgen}(r), t))$$

$$\approx$$


$$!^{N''} \mathbf{new} r : \text{mkeyseed}; ($$

$$!^N (x : \text{bitstring}) \rightarrow \text{mac}(x, \text{mkgen}(r)),$$

$$!^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow$$

$$\mathbf{find} j \leq N \mathbf{suchthat} \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge \\ \text{verify}(m, \text{mkgen}(r), t) \mathbf{then true else false})$$

The prover understands such specifications of primitives.

They can be reused in the proof of many protocols. 

# MACs: formal implementation


$$\text{verify}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \mathbf{true}$$

$$\begin{aligned} & !^{N''} \mathbf{new} \ r : \text{mkeyseed}; ( \\ & \quad !^N (x : \text{bitstring}) \rightarrow \text{mac}(x, \text{mkgen}(r)), \\ & \quad !^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow \text{verify}(m, \text{mkgen}(r), t)) \end{aligned}$$

$$\approx$$

$$\begin{aligned} & !^{N''} \mathbf{new} \ r : \text{mkeyseed}; ( \\ & \quad !^N (x : \text{bitstring}) \rightarrow \text{mac}'(x, \text{mkgen}'(r)), \\ & \quad !^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow \\ & \quad \quad \mathbf{find} \ j \leq N \ \mathbf{suchthat} \ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge \\ & \quad \quad \text{verify}'(m, \text{mkgen}'(r), t) \ \mathbf{then} \ \mathbf{true} \ \mathbf{else} \ \mathbf{false}) \end{aligned}$$

The prover understands such specifications of primitives.

They can be reused in the proof of many protocols. 

# MACs: formal implementation

The prover applies the previous rule automatically in **any (polynomial-time) context**, perhaps containing **several occurrences** of *mac* and of *verify*:

- Each occurrence of *mac* is replaced with *mac'*.
- Each occurrence of *verify* is replaced with a **find** that looks in all arrays of computed MACs (one array for each occurrence of function *mac*).

# Exercises

## Exercise

A **strongly unforgeable under chosen message attacks (SUF-CMA)** MAC is defined as follows:

$$\Pr[\text{verify}(m, k, t) \mid k \stackrel{R}{\leftarrow} \text{mkgen}; (m, t) \leftarrow \mathcal{A}^{\text{mac}(\cdot, k), \text{verify}(\cdot, k, \cdot)}]$$

is negligible, when  $t$  is not the result of calling the MAC oracle  $\text{mac}(\cdot, k)$  on  $m$ .

Represent it in the CryptoVerif formalism.

# Exercises

## Exercise

A **signature scheme** consists of

- a key generation algorithm  $(pk, sk) \xleftarrow{R} kgen$
- a signature algorithm  $sign(m, sk)$
- a verification algorithm  $verify(m, pk, s)$

such that  $verify(m, pk, sign(m, sk)) = 1$ .

A signature scheme is unforgeable under chosen message attacks (UF-CMA) if and only if

$$\Pr[verify(m, pk, s) \mid (pk, sk) \xleftarrow{R} kgen; (m, s) \leftarrow \mathcal{A}^{sign(\cdot, sk)}(pk)]$$

is negligible, when the adversary  $\mathcal{A}$  has not called the  $sign(\cdot, sk)$  oracle on message  $m$ .

Represent it in the CryptoVerif formalism.

# IND-CPA symmetric encryption

We consider a probabilistic, length-revealing encryption scheme that satisfies INDistinguishability under Chosen Plaintext Attacks (IND-CPA).

$$\text{dec}(\text{enc}(m, \text{kgen}(r), r'), \text{kgen}(r)) = i_{\perp}(m)$$

$$\begin{aligned} & !^N \mathbf{new} \ r : \text{keyseed}; !^N(x : \text{bitstring}) \rightarrow \\ & \quad \mathbf{new} \ r' : \text{coins}; \text{enc}(x, \text{kgen}(r), r') \end{aligned}$$

$$\approx$$

$$\begin{aligned} & !^N \mathbf{new} \ r : \text{keyseed}; !^N(x : \text{bitstring}) \rightarrow \\ & \quad \mathbf{new} \ r' : \text{coins}; \text{enc}'(Z(x), \text{kgen}'(r), r') \end{aligned}$$

$Z(x)$  is the bitstring of the same length as  $x$  containing only zeroes (for all  $x : \text{nonce}$ ,  $Z(x) = Z_{\text{nonce}}, \dots$ ).

# Exercises

## Exercise

A symmetric encryption scheme satisfies ciphertext integrity (INT-CTXT) if and only if

$$\Pr[\text{dec}(c, k) \neq \perp \mid k \stackrel{R}{\leftarrow} \text{kgen}; c \leftarrow \mathcal{A}^{\text{enc}(\cdot, k), \text{dec}(\cdot, k) \neq \perp}]$$

is negligible, when  $c$  is not the result of a call to the  $\text{enc}(\cdot, k)$  oracle. Represent it in the CryptoVerif formalism.

# Exercises

## Exercise

A **public-key encryption scheme** consists of

- a key generation algorithm  $(pk, sk) \xleftarrow{R} kgen$
- a probabilistic encryption algorithm  $enc(m, pk)$
- a decryption algorithm  $dec(m, sk)$

such that  $dec(enc(m, pk), sk) = m$ .

A public-key encryption scheme is indistinguishable under adaptive chosen-ciphertext attacks (IND-CCA2) if and only if

$$2 \Pr \left[ \begin{array}{l} b \xleftarrow{R} \{0, 1\}; (pk, sk) \xleftarrow{R} kgen; \\ b' \leftarrow \mathcal{A}^{enc(LR(\cdot, \cdot, b), pk), dec(\cdot, sk)}(pk) : b' = b \end{array} \right] - 1$$

is negligible, where  $\mathcal{A}$  never calls the  $dec(\cdot, sk)$  oracle on the result of the  $enc(LR(\cdot, \cdot, b), pk)$  oracle. Represent it in the CryptoVerif formalism.

# Syntactic transformations: expansion of assignments

**Expansion of assignments:** replacing a variable with its value.  
(Not completely trivial because of array references.)

## Example

If  $mk$  is defined by

$$\mathbf{let} \ mk = \mathit{mkgen}(r')$$

and there are no array references to  $mk$ , then  $mk$  is replaced with  $\mathit{mkgen}(r')$  in the game and the definition of  $mf$  is removed.

# Syntactic transformations: single assignment renaming

**Single assignment renaming:** when a variable is assigned at several places, rename it with a distinct name for each assignment.  
(Not completely trivial because of array references.)

## Example

```

start(); new  $r_A : T_r$ ; let  $k_A = kgen(r_A)$  in
  new  $r_B : T_r$ ; let  $k_B = kgen(r_B)$  in  $\bar{c}(\langle \rangle; (Q_K \mid Q_S))$ 
 $Q_K = !^{i \leq n} c(h : T_h, k : T_k)$ 
  if  $h = A$  then let  $k' = k_A$  else
  if  $h = B$  then let  $k' = k_B$  else let  $k' = k$ 
 $Q_S = !^{i' \leq n'} c'(h' : T_h);$ 
  find  $j \leq n$  suchthat  $\text{defined}(h[j], k'[j]) \wedge h' = h[j]$  then  $P_1(k'[j])$ 
  else  $P_2$ 

```

# Syntactic transformations: single assignment renaming

**Single assignment renaming:** when a variable is assigned at several places, rename it with a distinct name for each assignment.  
(Not completely trivial because of array references.)

## Example

*start()*; **new**  $r_A : T_r$ ; **let**  $k_A = kgen(r_A)$  **in**

**new**  $r_B : T_r$ ; **let**  $k_B = kgen(r_B)$  **in**  $\bar{c}(\cdot)$ ; ( $Q_K \mid Q_S$ )

$Q_K = !^{i \leq n} c(h : T_h, k : T_k)$

**if**  $h = A$  **then let**  $k'_1 = k_A$  **else**

**if**  $h = B$  **then let**  $k'_2 = k_B$  **else let**  $k'_3 = k$

$Q_S = !^{i' \leq n'} c'(h' : T_h)$ ;

**find**  $j \leq n$  **suchthat** **defined**( $h[j], k'_1[j]$ )  $\wedge h' = h[j]$  **then**  $P_1(k'_1[j])$

**orfind**  $j \leq n$  **suchthat** **defined**( $h[j], k'_2[j]$ )  $\wedge h' = h[j]$  **then**  $P_1(k'_2[j])$

**orfind**  $j \leq n$  **suchthat** **defined**( $h[j], k'_3[j]$ )  $\wedge h' = h[j]$  **then**  $P_1(k'_3[j])$

**else**  $P_2$

# Syntactic transformations: move new

**Move new:** move restrictions downwards in the game as much as possible, when there is no array reference to them.

(Moving **new**  $x : T$  under a **if** or a **find** duplicates it.

A subsequent single assignment renaming will distinguish cases.)

## Example

**new**  $x : \text{nonce}$ ; **if**  $c$  **then**  $P_1$  **else**  $P_2$

becomes

**if**  $c$  **then** **new**  $x : \text{nonce}$ ;  $P_1$  **else** **new**  $x : \text{nonce}$ ;  $P_2$

# Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - **Assignments:** **let**  $x = M$  **in**  $P$  implies that  $x = M$  in  $P$
  - **Tests:** **if**  $M = N$  **then**  $P$  implies that  $M = N$  in  $P$
  - **Definitions of cryptographic primitives**
  - When a **find** guarantees that  $x[j]$  is **defined**, equalities that hold at definition of  $x$  also hold under the **find** (after substituting  $j$  for the array indices at the definition of  $x$ )
  - **Elimination of collisions:** if  $x$  is created by **new**  $x : T$ ,  $x[i] = x[j]$  implies  $i = j$ , up to negligible probability (when  $T$  is large)
- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly.  
For instance:
  - If  $M$  simplifies to **true**, then **if**  $M$  **then**  $P_1$  **else**  $P_2$  simplifies  $P_1$ .
  - If a condition of **find** simplifies to **false**, then the corresponding branch is removed.

# Proof of security properties: one-session secrecy

**One-session secrecy:** the adversary cannot distinguish any of the secrets from a random number with one test query.

**Criterion for proving one-session secrecy of  $x$ :**

$x$  is defined by **new**  $x[i] : T$  and there is a set of variables  $S$  such that only variables in  $S$  depend on  $x$ .

The output messages and the control-flow do not depend on  $x$ .

# Proof of security properties: secrecy

**Secrecy:** the adversary cannot distinguish the secrets from independent random numbers with several test queries.

**Criterion for proving secrecy of  $x$ :** same as one-session secrecy, plus  $x[i]$  and  $x[i']$  do not come from the same copy of the same restriction when  $i \neq i'$ .

# Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, it tries to analyze why the transformation failed, and **suggests syntactic transformations** that could make it work.
- One tries to execute these syntactic transformations. (If they fail, they may also suggest other syntactic transformations, which are then executed.)
- We retry the cryptographic transformation, and so on.

# Proof of the example: initial game

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$   
 $\mathbf{new} \ r' : \text{mkeyseed}; \mathbf{let} \ mk : \text{mkey} = \text{mkgen}(r') \mathbf{in} \ \bar{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$   
 $\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$   
 $\bar{c}_A\langle e, \text{mac}(e, mk) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
 $\mathbf{if} \ \text{verify}(e', mk, ma) \mathbf{then}$   
 $\mathbf{let} \ i_{\perp}(k2b(k'')) = \text{dec}(e', k) \mathbf{in} \ \bar{c}_B\langle \rangle$

# Proof of the example: remove assignments $mk$

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$   
 $\mathbf{new} \ r' : \text{mkeyseed}; \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$   
 $\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$   
 $\overline{c}_A\langle e, \text{mac}(e, \text{mkgen}(r')) \rangle$

$Q_B = !^{i \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
 $\mathbf{if} \ \text{verify}(e', \text{mkgen}(r'), ma) \mathbf{then}$   
 $\mathbf{let} \ i_{\perp}(k2b(k'')) = \text{dec}(e', k) \mathbf{in} \ \overline{c}_B\langle \rangle$

# Proof of the example: security of the MAC

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$   
 $\mathbf{new} \ r' : \text{mkeyseed}; \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i' \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$   
 $\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$   
 $\overline{c}_A\langle e, \text{mac}'(e, \text{mkgen}'(r'')) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
 $\mathbf{find} \ j \leq n \mathbf{suchthat} \ \mathbf{defined}(e[j]) \wedge e' = e[j] \wedge$   
 $\mathbf{verify}'(e', \text{mkgen}'(r'), ma) \mathbf{then}$   
 $\mathbf{let} \ i_{\perp}(k2b(k'')) = \text{dec}(e', k) \mathbf{in} \ \overline{c}_B\langle \rangle$

# Proof of the example: simplify

$$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{let} \ k : \text{key} = \text{kgen}(r) \mathbf{in}$$

$$\mathbf{new} \ r' : \text{mkeyseed}; \overline{c} \langle \rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$$

$$\mathbf{let} \ e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \mathbf{in}$$

$$\overline{c}_A \langle e, \text{mac}'(e, \text{mkgen}'(r')) \rangle$$

$$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, \text{ma} : \text{macstring});$$

$$\mathbf{find} \ j \leq n \mathbf{suchthat} \ \mathbf{defined}(e[j]) \wedge e' = e[j] \wedge$$

$$\text{verify}'(e', \text{mkgen}'(r'), \text{ma}) \mathbf{then}$$

$$\mathbf{let} \ k'' = k'[j] \mathbf{in} \ \overline{c}_B \langle \rangle$$

$$\text{dec}(e', k) = \text{dec}(\text{enc}(k2b(k'[j]), k, r''[j]), k) = i_{\perp}(k2b(k'[j]))$$

# Proof of the example: remove assignments $k$

$Q_0 = \text{start}(); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};$

**let**  $e : \text{bitstring} = \text{enc}(k2b(k'), \text{kgen}(r), r'')$  **in**  
 $\overline{c}_A\langle e, \text{mac}'(e, \text{mkgen}'(r')) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$

**find**  $j \leq n$  **suchthat**  $\text{defined}(e[j]) \wedge e' = e[j] \wedge$   
 $\text{verify}'(e', \text{mkgen}'(r'), ma)$  **then**

**let**  $k'' = k'[j]$  **in**  $\overline{c}_B\langle \rangle$

# Proof of the example: security of the encryption

$Q_0 = \text{start}(); \mathbf{new} \ r : \text{keyseed}; \mathbf{new} \ r' : \text{mkeyseed}; \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ k' : \text{key}; \mathbf{new} \ r'' : \text{coins};$

$\mathbf{let} \ e : \text{bitstring} = \text{enc}'(Z(k2b(k')), \text{kgen}'(r), r'') \mathbf{in}$   
 $\overline{c}_A\langle e, \text{mac}'(e, \text{mkgen}'(r')) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, \text{ma} : \text{macstring});$

$\mathbf{find} \ j \leq n \mathbf{suchthat} \ \mathbf{defined}(e[j]) \wedge e' = e[j] \wedge$   
 $\text{verify}'(e', \text{mkgen}'(r'), \text{ma}) \mathbf{then}$

$\mathbf{let} \ k'' = k'[j] \mathbf{in} \ \overline{c}_B\langle \rangle$

# Proof of the example: simplify

$Q_0 = \text{start}(); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \overline{c}\langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i' \leq n} c_A(); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};$   
**let**  $e : \text{bitstring} = \text{enc}'(Z_k, \text{kgen}'(r), r'')$  **in**  
 $\overline{c}_A\langle e, \text{mac}'(e, \text{mkgen}'(r')) \rangle$

$Q_B = !^{i' \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
**find**  $j \leq n$  **suchthat** **defined** $(e[j]) \wedge e' = e[j] \wedge$   
 $\text{verify}'(e', \text{mkgen}'(r'), ma)$  **then**  
**let**  $k'' = k'[j]$  **in**  $\overline{c}_B\langle \rangle$

$Z(k2b(k')) = Z_k$

# Proof of the example: secrecy

$Q_0 = \text{start}(); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \overline{c} \langle \rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};$   
 $\text{let } e : \text{bitstring} = \text{enc}'(Z_k, \text{kgen}'(r), r'') \text{ in}$   
 $\overline{c}_A \langle e, \text{mac}'(e, \text{mkgen}'(r')) \rangle$

$Q_B = !^{i \leq n} c_B(e' : \text{bitstring}, ma : \text{macstring});$   
 $\text{find } j \leq n \text{ suchthat defined}(e[j]) \wedge e' = e[j] \wedge$   
 $\text{verify}'(e', \text{mkgen}'(r'), ma) \text{ then}$   
 $\text{let } k'' = k'[j] \text{ in } \overline{c}_B \langle \rangle$

Preserves the one-session secrecy of  $k''$  but not its secrecy.

# Example of the FDH signature (joint work with D. Pointcheval)

hash function (in the random oracle model)

$f(pk, m)$  one-way trapdoor permutation, with inverse  $\text{invf}(sk, m)$ .

We define a **signature scheme** as follows:

- signature  $\text{sign}(m, sk) = \text{invf}(sk, \text{hash}(m))$
- verification  $\text{verify}(m, pk, s) = (f(pk, s) = \text{hash}(m))$

Our goal is to show that this signature scheme is UF-CMA (secure against existential forgery under chosen message attacks).

# Formalizing the security of a signature scheme (1)

## Key generation:

$c_{gen}()$ ; **new**  $r : seed$ ; **let**  $sk = skgen(r)$  **in let**  $pk = pkgen(r)$  **in**  $\overline{c_{gen}}\langle pk \rangle$

Chooses a random seed uniformly in the set of bit-strings  $seed$  (consisting of all bit-strings of a certain length), generates a public key  $pk$ , a secret key  $sk$ , and returns the public key.

# Formalizing the security of a signature scheme (2)

Signature:

$$c_S(m : \textit{bitstring}); \overline{c_S} \langle \text{sign}(sk, m) \rangle$$

# Formalizing the security of a signature scheme (2)

**Signature:**

$$c_S(m : \text{bitstring}); \overline{c_S} \langle \text{sign}(sk, m) \rangle$$

This process can be called at most  $q_S$  times:

$$!^{i_S \leq q_S} c_S(m : \text{bitstring}); \overline{c_S} \langle \text{sign}(sk, m) \rangle$$

# Formalizing the security of a signature scheme (2)

**Signature:**

$$c_S(m : \textit{bitstring}); \overline{c_S}\langle \text{sign}(sk, m) \rangle$$

This process can be called at most  $q_S$  times:

$$!^{i_S \leq q_S} c_S(m : \textit{bitstring}); \overline{c_S}\langle \text{sign}(sk, m) \rangle$$

In fact, this is an abbreviation for:

$$!^{i_S \leq q_S} c_S(m[i_S] : \textit{bitstring}); \overline{c_S}\langle \text{sign}(sk, m[i_S]) \rangle$$

The variables in repeated oracles are arrays, with one cell for each call, to remember the values used in each oracle call.

These arrays are indexed with the call number  $i_S$ .

# Formalizing the security of a signature scheme (3)

Test:

```

 $c_T(m' : \text{bitstring}, s : D);$  if  $\text{verify}(m', pk, s)$  then
  find  $j \leq q_S$  suchthat  $\text{defined}(m[j]) \wedge (m' = m[j])$ 
  then yield else event forge

```

If  $s$  is a signature for  $m'$  and the signed message  $m'$  is not contained in the array  $m$  of messages passed to signing oracle, then the signature is a **forgery**, so we execute **event forge**.

# Formalizing the security of a signature scheme (summary)

The signature and test oracles make sense only **after** the key generation oracle has been called, hence a **sequential composition**.

The signature and test oracles are **simultaneously** available, hence a **parallel composition**.

```

 $c_{gen}();$  new  $r : seed;$  let  $sk = skgen(r)$  in let  $pk = pkgen(r)$  in  $\overline{c_{gen}}\langle pk \rangle$ 
  ( $!^{is \leq qs} c_S(m : bitstring); \overline{c_S}\langle sign(sk, m) \rangle$ )
  |  $c_T(m' : bitstring, s : D);$  if  $verify(m', pk, s)$  then
    find  $j \leq qs$  suchthat  $defined(m[j]) \wedge (m' = m[j])$ 
    then yield else event forge)

```

The probability of executing **event** forge in this game is the probability of forging a signature.

# Application to the FDH signature scheme

We add a hash oracle because the adversary must be able to call the random oracle (even though it cannot be implemented).

```

 $c_{gen}()$ ; new  $r : seed$ ; let  $sk = skgen(r)$  in let  $pk = pkgen(r)$  in  $\overline{c_{gen}}\langle pk \rangle$ 
  ( $!^{i_H \leq q_H} c_H(x : bitstring)$ ;  $\overline{c_H}\langle hash(x) \rangle$ )
  ( $!^{i_S \leq q_S} c_S(m : bitstring)$ ;  $\overline{c_S}\langle invf(sk, hash(m)) \rangle$ )
  |  $c_T(m' : bitstring, s : D)$ ; if  $f(pk, s) = hash(m')$  then
    find  $j \leq q_S$  suchthat defined( $m[j]$ )  $\wedge (m' = m[j])$ 
    then yield else event forge)
  
```

Our goal is to **bound the probability that event forge is executed** in this game.

This game is given as input to the prover in the syntax above.

# FDH: security of a hash function (random oracle model)

A hash function is equivalent to a “**random function**”: a function that

- returns a new random number when it is called on a new argument,
- and returns the same result when it is called on the same argument.

$$!^{n_h} (x : \textit{bitstring}) \rightarrow \text{hash}(x) \textit{ [all]}$$

$$\approx_0$$

$$!^{n_h} (x : \textit{bitstring}) \rightarrow$$

**find**  $j \leq N$  **suchthat** **defined**( $x[j], r[j]$ )  $\wedge$  ( $x = x[j]$ )

**then**  $r[j]$

**else new**  $r : D; r$

# FDH: one-wayness

The adversary inverts  $f$  when, given the public key  $pk = \text{pkgen}(r_0)$  and the image of some  $x_0$  by  $f_{pk}$ , it manages to find  $x_0$  (without having the trapdoor).

The function  $f$  is **one-way** when the adversary has negligible probability of inverting  $f$ .

# FDH: one-wayness

```

!nk new r : seed; (
  () → pkgen(r),
  !nf new y : D; (
    () → f(pkgen(r), y),
    !n1 (x : D) → (x = y),
    () → y))

```

$\approx_{n_k \times n_f \times \text{Succ}_P^{\text{OW}}(t + (n_k n_f - 1)t_f + (n_k - 1)t_{\text{pkgen}})}$

```

!nk new r : seed; (
  () → pkgen'(r),
  !nf new y : D; (
    () → f'(pkgen'(r), y),
    !n1 (x : D) → if defined(k) then x = y else false,
    () → let k : bitstring = mark in y))

```

# FDH: other properties of one-way trapdoor permutations

invf is the inverse of f:

$$\forall r : \text{seed}, x : D; \text{invf}(\text{skgen}(r), f(\text{pkgen}(r), x)) = x$$

f is injective:

$$\forall k : \text{key}, x : D, x' : D; (f(k, x) = f(k, x')) = (x = x')$$

We can replace a uniformly distributed random number  $y$  with  $f(\text{pkgen}(r), y')$  where  $y'$  is a uniformly distributed random number:

$$\begin{aligned} & !^{n_k} \text{ new } r : \text{seed}; ( \\ & \quad () \rightarrow \text{pkgen}(r), \\ & \quad !^{n_f} \text{ new } y : D; ((() \rightarrow \text{invf}(\text{skgen}(r), y), () \rightarrow y)) \end{aligned}$$

$$\begin{aligned} & \approx_0 \\ & !^{n_k} \text{ new } r : \text{seed}; ( \\ & \quad () \rightarrow \text{pkgen}(r), \\ & \quad !^{n_f} \text{ new } y' : D; ((() \rightarrow y', () \rightarrow f(\text{pkgen}(r), y'))) \end{aligned}$$

# FDH: initial game

```

 $c_{gen}()$ ; new  $r : seed$ ; let  $sk = skgen(r)$  in let  $pk = pkgen(r)$  in  $\overline{c_{gen}}\langle pk \rangle$ ;
( (* hash oracle *)
   $!^{i_H \leq q_H} c_H(x : bitstring)$ ;  $\overline{c_H}\langle hash(x) \rangle$ 
| (* signature oracle *)
   $!^{i_S \leq q_S} c_S(m : bitstring)$ ;  $\overline{c_S}\langle invf(sk, hash(m)) \rangle$ 
| (* forged signature? *)
   $c_T(m' : bitstring, s : D)$ ;
  if  $(f(pk, s) = hash(m'))$  then
  find  $j \leq q_S$  suchthat  $defined(m[j]) \wedge (m' = m[j])$  then
    yield
  else
    event forge
)

```

# FDH step 1: apply the security of the hash function

Replace each occurrence of  $\text{hash}(M)$  with a lookup in the arguments of previous calls to  $\text{hash}$ .

- If  $M$  is found, return the same result as the previous result.
- Otherwise, pick a new random number and return it.

For instance,  $\overline{c}_H\langle\text{hash}(x)\rangle$  is replaced with

```
find suchthat defined( $m', r_{30}$ )  $\wedge$  ( $x = m'$ ) then
   $\overline{c}_H\langle r_{30}\rangle$ 
orfind  $@i1 \leq q_S$  suchthat defined( $m[@i1], r_{32}[@i1]$ )
   $\wedge$  ( $x = m[@i1]$ ) then  $\overline{c}_H\langle r_{32}[@i1]\rangle$ 
orfind  $@i2 \leq q_H$  suchthat defined( $x[@i2], r_{34}[@i2]$ )
   $\wedge$  ( $x = x[@i2]$ ) then  $\overline{c}_H\langle r_{34}[@i2]\rangle$ 
else
  new  $r_{34} : D$ ;  $\overline{c}_H\langle r_{34}\rangle$ 
```

## FDH step 2: simplify

```

(* forged signature? *)
c_T(m' : bitstring, s : D);
find suchthat defined(m', r_30)  $\wedge$  (m' = m') then
  if (f(pk, s) = r_30) then
    find j  $\leq$  q_S suchthat defined(m[j])  $\wedge$  (m' = m[j]) then yield else event forge
  orfind @i5  $\leq$  q_S suchthat defined(m[@i5], r_32[@i5])  $\wedge$  (m' = m[@i5]) then
    if (f(pk, s) = r_32[@i5]) then
      find j  $\leq$  q_S suchthat defined(m[j])  $\wedge$  (m' = m[j]) then yield else event forge
    orfind @i6  $\leq$  q_H suchthat defined(x[@i6], r_34[@i6])  $\wedge$  (m' = x[@i6]) then
      if (f(pk, s) = r_34[@i6]) then
        find j  $\leq$  q_S suchthat defined(m[j])  $\wedge$  (m' = m[j]) then yield else event forge
      else
        new r_30 : D;
        if (f(pk, s) = r_30) then
          find j  $\leq$  q_S suchthat defined(m[j])  $\wedge$  (m' = m[j]) then yield else event forge

```

The **red** test always succeeds, so the **blue** part becomes **yield**.

**r\_30** is never defined in the first branch of **find**. 

## FDH step 3: substitute $sk$ with its value

The variable  $sk$  is replaced with  $skgen(r)$ , and the assignment **let**  $sk = skgen(r)$  is removed.  
This transformation is advised in order to be able to apply the permutation property.

# FDH step 4: permutation

(\* signature oracle \*)

```

!i_s ≤ q_s
c_S(m : bitstring);
find suchthat defined(m', r_30) ∧ (m = m') then
   $\overline{c_S}$ ⟨invf(skgen(r), r_30)⟩
orfind @i_3 ≤ q_s suchthat defined(m[@i_3], r_32[@i_3]) ∧ (m = m[@i_3]) then
   $\overline{c_S}$ ⟨invf(skgen(r), r_32[@i_3])⟩
orfind @i_4 ≤ q_H suchthat defined(x[@i_4], r_34[@i_4]) ∧ (m = x[@i_4]) then
   $\overline{c_S}$ ⟨invf(skgen(r), r_34[@i_4])⟩
else
  new r_32 : D;
   $\overline{c_S}$ ⟨invf(skgen(r), r_32)⟩

```

**new** r\_i : D becomes **new** y\_i : D,  
 invf(skgen(r), r\_i) becomes y\_i,  
 r\_i becomes f(pkgen(r), y\_i)

# FDH step 5: simplify

```
(* forged signature *)
c_T(m' : bitstring, s : D);
find @i5 ≤ q_S suchthat defined(m[@i5], r_32[@i5]) ∧ (m' = m[@i5]) then yield
orfind @i6 ≤ q_H suchthat defined(x[@i6], r_34[@i6]) ∧ (m' = x[@i6]) then
  if (f(pk, s) = f(pkgen(r), y_34[@i6])) then
    find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge
else
  new y_30 : D;
  if (f(pk, s) = f(pkgen(r), y_30)) then
    find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge
```

$f(pk, s) = f(pkgen(r), y_i)$  becomes  $s = y_i$ ,  
 knowing  $pk = pkgen(r)$  and the injectivity of  $f$ :  
 $\forall k : key, x : D, x' : D; (f(k, x) = f(k, x')) = (x = x')$

# FDH step 6: one-wayness

(\* forged signature? \*)

$c_T(m' : \text{bitstring}, s : D);$

**find**  $@i5 \leq q_S$  **suchthat**  $\text{defined}(m[@i5], r_{32}[@i5]) \wedge (m' = m[@i5])$  **then yield**

**orfind**  $@i6 \leq q_H$  **suchthat**  $\text{defined}(x[@i6], r_{34}[@i6]) \wedge (m' = x[@i6])$  **then**

**if**  $s = y_{34}[@i6]$  **then**

**find**  $j \leq q_S$  **suchthat**  $\text{defined}(m[j]) \wedge (m' = m[j])$  **then yield else event forge**

**else**

**new**  $y_{30} : D;$

**if**  $s = y_{30}$  **then**

**find**  $j \leq q_S$  **suchthat**  $\text{defined}(m[j]) \wedge (m' = m[j])$  **then yield else event forge**

$s = y_{-i}$  becomes **find**  $@j_{-i} \leq q_H$  **suchthat**  $\text{defined}(k_{-i}[@j_{-i}])$

**then**  $s = y_{-i}$  **else false,**

In **hash oracle**,  $f(\text{pkgen}(r), y_{-i})$  becomes  $f'(\text{pkgen}'(r), y_{-i})$ ,

In **signature oracle**,  $y_{-i}$  becomes **let**  $k_{-i} : \text{bitstring} = \text{mark}$  **in**  $y_{-i}$ .

Difference of probability:  $(q_H + q_S + 1)P_{OW}(\text{time})$ .

# FDH step 7: simplify

```

(* forged signature? *)
c_T(m' : bitstring, s : D);
find @i5 ≤ q_S suchthat defined(m[@i5], r_32[@i5]) ∧ (m' = m[@i5]) then yield
orfind @i6 ≤ q_H suchthat defined(x[@i6], r_34[@i6]) ∧ (m' = x[@i6]) then
  find @j_34 ≤ q_S suchthat defined(k_34[@j_34]) ∧ (@i4[@j_34] = @i6) then
    if s = y_34[@i6] then
      find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge
    else
      new y_30 : D;
      find @j_30 ≤ q_H suchthat defined(k_30[@j_30]) then
        if s = y_30 then
          find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge

```

The tests in **red** always succeed, so **event** forge disappears, which proves the desired property.

## FDH step 7: simplify (2)

```

(* forged signature? *)
c_T(m' : bitstring, s : D);
...
  new y_30 : D;
  find @j_30 ≤ q_H suchthat defined(k_30[@j_30]) then
    if s = y_30 then
      find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge

```

Definition of  $k_{30}$ :

```

!i s ≤ q_S
c_S(m : bitstring);
find suchthat defined(m', y_30) ∧ (m = m') then
  let k_30 : bitstring = mark in ...

```

When  $k_{30}[@j_{30}]$  is defined,  $m[@j_{30}]$  is defined and  $m[@j_{30}] = m'$ , so the test  $j \leq q_S$  suchthat defined( $m[j]$ ) ∧ ( $m' = m[j]$ ) succeeds with  $j = @j_{30}$ .

## FDH step 7: simplify (3)

```
(* forged signature? *)
c_T(m' : bitstring, s : D);
...
orfind @i6 ≤ q_H suchthat defined(x[@i6], r_34[@i6]) ∧ (m' = x[@i6]) then
  find @j_34 ≤ q_S suchthat defined(k_34[@j_34]) ∧ (@i4[@j_34] = @i6) then
    if s = y_34[@i6] then
      find j ≤ q_S suchthat defined(m[j]) ∧ (m' = m[j]) then yield else event forge
```

Definition of  $k_{34}$ :

```
!i_s ≤ q_S
c_S(m : bitstring);
...
orfind @i4 ≤ q_H suchthat defined(x[@i4], y_34[@i4]) ∧ (m = x[@i4]) then
  let k_34 : bitstring = mark in ...
```

When  $k_{34}[@j_{34}]$  is defined,  $m[@j_{34}]$  is defined and

$m[@j_{34}] = x[@i4[@j_{34}]] = x[@i6] = m'$

so the **red** test succeeds with  $j = @j_{34}$ .

# Experiments

Tested on the following protocols (original and corrected versions):

- Otway-Rees (shared-key)
- Yahalom (shared-key)
- Denning-Sacco (public-key)
- Woo-Lam shared-key and public-key
- Needham-Schroeder shared-key and public-key
- Full domain hash signature (with D. Pointcheval)
- Encryption schemes of Bellare-Rogaway'93 (with D. Pointcheval)

Shared-key encryption is implemented as encrypt-then-MAC, using a IND-CPA encryption scheme.

(For Otway-Rees, we also considered a SPRP encryption scheme, a IND-CPA + INT-CTXT encryption scheme, a IND-CCA2 + IND-PTXT encryption scheme.)

Public-key encryption is assumed to be IND-CCA2.

We prove secrecy of session keys and correspondence properties.

# Results (1)

In most cases, the prover succeeds in proving the desired properties when they hold, and obviously it always fails to prove them when they do not hold.

Only cases in which the prover fails although the property holds:

- Needham-Schroeder public-key when the exchanged key is the nonce  $N_A$ .
- Needham-Schroeder shared-key: fails to prove that  $N_B[i] \neq N_B[i'] - 1$  with overwhelming probability, where  $N_B$  is a nonce
- Showing that the encryption scheme  $\mathcal{E}(m, r) = f(r) \| H(r) \oplus m \| H'(m, r)$  is IND-CCA2.

## Results (2)

- Some public-key protocols need **manual proofs**.  
(Give the cryptographic proof steps and single assignment renaming instructions.)
- **Runtime**: 7 ms to 35 s, average: 5 s on a Pentium M 1.8 GHz.
- A detailed case study of **Kerberos V**, with and without its public-key extension PKINIT (AsiaCCS'08, with with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- Starts being **used by others**: Verification of F# implementations, including TLS, by Microsoft Research and the MSR-INRIA lab.

# Conclusion

CryptoVerif can automatically prove the security of primitives and protocols.

- The **security assumptions** are given as **observational equivalences** (proved manually **once**).
- The **protocol or scheme** to prove is specified in a process calculus.
- The prover provides a **sequence of indistinguishable games** that lead to the proof and a bound on the **probability of an attack**.
- The user is allowed (but does not have) to interact with the prover to make it follow a specific sequence of games.

Future extensions:

- Extension to **other cryptographic primitives**, in particular Diffie-Hellman, full support of XOR.
- More **game transformations**.
- More **case studies**.

More information: <http://www.cryptoverif.ens.fr/>

# Related work

Proof tools and techniques in the computational model:

- Automatic prover by Tšahhrov and Laud [TGC'07].  
Similar ideas to CryptoVerif, but uses a different game representation (dependency graph).
- Interactive provers: CertiCrypt, produces Coq proofs [Barthe et al, POPL'09, FAST'08, IEEE S&P'09].
- Logics:
  - Computational PCL [Datta et al, ICALP'05, CSFW'06]
  - CIL [Barthe et al, FCC'09]
- Type systems [Backes and Laud, CCS'06].

# Acknowledgments

I warmly thank **David Pointcheval** for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him.

This work was partly supported by the ANR project ARA SSIA FormaCrypt.

Questions?