Automated Security Proofs with Sequences of Games

Bruno Blanchet and David Pointcheval
CNRS, Département d'Informatique,
École Normale Supérieure

October 2006

Proofs of cryptographic protocols

There are two main frameworks for analyzing security protocols:

- The Dolev-Yao model: a formal, abstract model.

  The cryptographic primitives are ideal blackboxes.
  The adversary uses only those primitives.

  Proofs can be done automatically.

- The computational model: a realistic model.

  The cryptographic primitives are functions on bit-strings.
  The adversary is a polynomial-time Turing machine.

  Proofs are done manually.

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

An automatic prover

We have implemented an automatic prover sound in the computational model:

- proves secrecy properties and that events can be executed only with negligible probability.
- handles various cryptographic primitives: MACs (message authentication codes), stream and block ciphers, public-key encryption, signatures, hash functions, ...
- works for N sessions with an active adversary.
- gives a bound on the probability of an attack (exact security).

Produced proofs

As in Shoup's or Bellare and Rogaway's method, the proof is a sequence of games:

- In the first game, the adversary plays against the real protocol.
- One goes from one game to the next by syntactic transformations or by applying security assumptions on cryptographic primitives.

  The difference of probability between consecutive games is bounded.
- The last game is "ideal": the security property can be read directly on it.

  (The advantage of the adversary is 0 for this game.)

Games are formalized in a process calculus.

Case study: Full Domain Hash signature scheme

hash hash function (in the random oracle model)
$f(pk, m)$ one-way trapdoor permutation, with inverse $invf(sk, m)$

We define a signature scheme as follows:

- signature: $sign(m, sk) = invf(sk, hash(m))$
- verification: $verify(m, pk, s) = (f(pk, s) = hash(m))$

Our goal is to show that this signature scheme is UF-CMA (satisfies unforgeability under chosen message attacks).

Formalizing the security of a signature scheme (1)

Key generation oracle:

$$Ogen() := r \stackrel{R}{\leftarrow} seed; pk \leftarrow pkgen(r); sk \leftarrow skgen(r); \textbf{return}(pk)$$

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 oracle:

$$OS(m : bitstring) := \textbf{return}(sign(sk, m))$$

Formalizing the security of a signature scheme (2)

Signature oracle:

$$OS(m : bitstring) := \mathbf{return}(sign(sk, m))$$

This oracle can be called at most q_S times:

$$\mathbf{foreach}\ i_S \leq q_S\ \mathbf{do}\ OS(m : bitstring) := \mathbf{return}(sign(sk, m))$$

Formalizing the security of a signature scheme (2)

Signature oracle:

$$OS(m : bitstring) := \textbf{return}(sign(sk, m))$$

This oracle can be called at most q_S times:

$$\textbf{foreach}\ i_S \leq q_S\ \textbf{do}\ OS(m : bitstring) := \textbf{return}(sign(sk, m))$$

In fact, this is an abbreviation for:

$$\textbf{foreach}\ i_S \leq q_S\ \textbf{do}\ OS[i_S](m[i_S] : bitstring) := \textbf{return}(sign(sk, m[i_S]))$$

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.

Test oracle:

$$OT(m' : bitstring, s : D) := \textbf{if } verify(m', pk, s) \textbf{ then}$$
$$\textbf{find } u \leq q_S \textbf{ suchthat } (\textbf{defined}(m[u]) \wedge m' = m[u])$$
$$\textbf{then end else event } forge$$

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.

$$Ogen() := r \stackrel{R}{\leftarrow} seed; pk \leftarrow pkgen(r); sk \leftarrow skgen(r); \mathbf{return}(pk);$$

$$(\mathbf{foreach}\ i_S \leq q_S\ \mathbf{do}\ OS(m : bitstring) := \mathbf{return}(sign(sk, m))$$

$$\mid OT(m' : bitstring, s : D) := \mathbf{if}\ verify(m', pk, s)\ \mathbf{then}$$

$$\quad \mathbf{find}\ u \leq q_S\ \mathbf{suchthat}\ (\mathbf{defined}(m[u]) \wedge m' = m[u])$$

$$\quad \mathbf{then\ end\ else\ event}\ forge)$$

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).

$$\textbf{foreach } i_H \leq q_H \textbf{ do } OH(x : \mathit{bitstring}) := \textbf{return}(\mathit{hash}(x))$$

$$\mid \mathit{Ogen}() := r \stackrel{R}{\leftarrow} \mathit{seed}; \mathit{pk} \leftarrow \mathit{pkgen}(r); \mathit{sk} \leftarrow \mathit{skgen}(r); \textbf{return}(\mathit{pk});$$

$$(\textbf{foreach } i_S \leq q_S \textbf{ do } OS(m : \mathit{bitstring}) := \textbf{return}(\mathit{invf}(\mathit{sk}, \mathit{hash}(m)))$$

$$\mid OT(m' : \mathit{bitstring}, s : D) := \textbf{if } f(\mathit{pk}, s) = \mathit{hash}(m') \textbf{ then}$$

$$\textbf{find } u \leq q_S \textbf{ suchthat } (\textbf{defined}(m[u]) \wedge m' = m[u])$$

$$\textbf{then end else event } \mathit{forge})$$

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.

Indistinguishability as observational equivalence

Two processes (games) Q_1, Q_2 are observationally equivalent up to probability p when an adversary running in time t has probability at most $p(t)$ of distinguishing them:

$$Q_1 \approx_p Q_2$$

The adversary is represented by an acceptable evaluation context C (essentially a process put in parallel with the considered games).

- Observational equivalence is reflexive and symmetric.
- If $Q_1 \approx_p Q_2$ and $Q_2 \approx_{p'} Q_3$ then $Q_1 \approx_{p+p'} Q_3$.
- It is contextual: $Q_1 \approx_p Q_2$ implies $C[Q_1] \approx_{p'} C[Q_2]$ where C is any acceptable evaluation context running in time t_C and $p'(t) = p(t + t_C)$.

Proof technique

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

- observational equivalences $L \approx_p R$ given as axioms and that come from security properties of primitives. These equivalences are used inside a context:
$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$
- syntactic transformations: simplification, expansion of assignments, ...

We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx_{p_2} \ldots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1+\ldots+p_m} G_m$.

If **event** *forge* cannot be executed in G_m, it can be executed with probability at most $p_1 + \ldots + p_m$ in G_0.

One-wayness

The adversary inverts f when, given the public key $pk = pkgen(r_0)$ 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, say at most probability $\mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t)$ when the adversary runs in time t.

$$LR = Ogen() := r_0 \stackrel{R}{\leftarrow} seed; x_0 \stackrel{R}{\leftarrow} D; \textbf{return}(pkgen(r_0), f(pkgen(r_0), x_0));$$
$$Oeq(x' : D) := \textbf{if } x' = x_0 \textbf{ then event } invert$$

LR executes **event** *invert* with probability at most $\mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t)$ in the presence of a context that runs in time t.
(The event *invert* is executed when the adversary inverts f.)

One-wayness as an observational equivalence

Security assumptions have to be given as equivalences. The following equivalence formalizes one-wayness:

$$\begin{array}{l}
\textbf{foreach } i_k \leq n_k \textbf{ do } r \xleftarrow{R} seed; (Opk() := \textbf{return}(pkgen(r)) \\
\quad \mid \textbf{foreach } i_f \leq n_f \textbf{ do } x \xleftarrow{R} D; (Oy() := \textbf{return}(f(pkgen(r), x)) \\
\quad\quad \mid \textbf{foreach } i_1 \leq n_1 \textbf{ do } Oeq(x' : D) := \textbf{return}(x' = x) \\
\quad\quad \mid Ox() := \textbf{return}(x)))
\end{array}$$

$$\approx_{n_k \times n_f \times \mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t + (n_k n_f - 1) t_f + (n_k - 1) t_{pkgen})}$$

$$\begin{array}{l}
\textbf{foreach } i_k \leq n_k \textbf{ do } r \xleftarrow{R} seed; (Opk() := \textbf{return}(pkgen'(r)) \\
\quad \mid \textbf{foreach } i_f \leq n_f \textbf{ do } x \xleftarrow{R} D; (Oy() := \textbf{return}(f'(pkgen'(r), x)) \\
\quad\quad \mid \textbf{foreach } i_1 \leq n_1 \textbf{ do } Oeq(x' : D) := \\
\quad\quad\quad \textbf{if defined}(k) \textbf{ then return}(x' = x) \textbf{ else return}(false) \\
\quad\quad \mid Ox() := k \leftarrow mark; \textbf{return}(x)))
\end{array}$$

This equivalence is proved manually, once for one-way trapdoor permutations. It can be reused in the proof of many schemes.

Other properties of one-way trapdoor permutations (1)

- $x \mapsto f(pkgen(r), x)$ and $x \mapsto invf(skgen(r), x)$ are inverse permutations:

$$invf(skgen(r), f(pkgen(r), x)) = x$$

- $x \mapsto f(pk, x)$ is injective:

$$(f(pk, x) = f(pk, x')) = (x = x')$$

Other properties of one-way trapdoor permutations (2)

When x is a uniformly distributed random number, we can replace x with $f(pk, x)$ without changing the probability distribution ($x \mapsto f(pk, x)$ is a permutation).

This is again expressed by an observational equivalence:

$$\begin{aligned}
&\textbf{foreach } i_k \leq n_k \textbf{ do } r \xleftarrow{R} seed; (Opk() := \textbf{return}(pkgen(r)) \\
&\qquad | \textbf{ foreach } i_f \leq n_f \textbf{ do } x \xleftarrow{R} D; (Oant() := \textbf{return}(invf(skgen(r), x)) \\
&\qquad\qquad\qquad | \; Oim() := \textbf{return}(x))) \\
\approx_0 &\textbf{foreach } i_k \leq n_k \textbf{ do } r \xleftarrow{R} seed; (Opk() := \textbf{return}(pkgen(r)) \\
&\qquad | \textbf{ foreach } i_f \leq n_f \textbf{ do } x \xleftarrow{R} D; (Oant() := \textbf{return}(x) \\
&\qquad\qquad\qquad | \; Oim() := \textbf{return}(f(pkgen(r), x))))
\end{aligned}$$

which allows to perform the previous replacement only when x is used in calls to $invf(skgen(r), x)$, where r is a random number such that r occurs only in $pkgen(r)$ and $invf(skgen(r), x)$ for some random numbers x.

Hash function in the random oracle model

A random oracle is formalized by the following equivalence:

$$
\begin{aligned}
& \textbf{foreach } i_{\mathrm{h}} \leq n_{\mathrm{h}} \textbf{ do } OH(x : bitstring) := \textbf{return}(hash(x)) \\
\approx_0\ & \textbf{foreach } i_{\mathrm{h}} \leq n_{\mathrm{h}} \textbf{ do } OH(x : bitstring) := \\
& \quad \textbf{find } u \leq n_{\mathrm{h}} \textbf{ suchthat } (\textbf{defined}(x[u], r[u]) \wedge x = x[u]) \\
& \quad \textbf{then return}(r[u]) \textbf{ else } r \stackrel{R}{\leftarrow} D; \textbf{return}(r)
\end{aligned}
$$

The hash function is equivalent to a function that looks up the argument x in the array of previous arguments given to OH.

- If x is not found, it returns a fresh random number r.
- If x is found, it returns the same result as in the previous call.

Automatic proof of FDH: result

Our prover is given as input

- the security assumptions of one-way trapdoor permutations and of the hash function,
- the initial game that formalizes the security of FDH.

It automatically produces a proof that this game executes **event** *forge* with probability at most

$$(q_H + q_S + 1)\mathrm{Succ}_{\mathcal{P}}^{\mathrm{ow}}(t + (q_H + q_S)t_f)$$

where t_f is the time of one evaluation of f (ignoring the time of bit-string comparisons).

This is the standard upper-bound [Bellare, Rogaway, CCS'93].

The prover also outputs

- the sequence of games that leads to this proof,
- a succinct explanation of the transformations performed between games.

Proof strategy

- The prover tries to apply all equivalences given as axioms, which represent security assumptions.

  It transforms the left-hand side into the right-hand side of the equivalence.
- If such a transformation succeeds, the obtained game is then simplified, using in particular equations given as axioms.
- When these transformations fail, they may return syntactic transformations to apply in order to make them succeed, called advised transformations.

  The prover then applies the advised transformations, and retries the initial transformation.

Proof of FDH: Initial game

$($

$\quad$ **foreach** $i_H \leq q_H$ **do**

$\quad$ $OH(x : bitstring) := \textbf{return}(hash(x))$

$|$

$\quad$ $Ogen() := r \xleftarrow{R} seed; pk \leftarrow pkgen(r); sk \leftarrow skgen(r); \textbf{return}(pk);$

$\quad$ ((* signature oracle *)

$\quad\quad$ **foreach** $i_S \leq q_S$ **do**

$\quad\quad$ $OS(m : bitstring) := \textbf{return}(invf(sk, hash(m)))$

$\quad$ | (* forged signature? *)

$\quad\quad$ $OT(m' : bitstring, s : D) :=$

$\quad\quad$ **if** $f(pk, s) = hash(m')$ **then**

$\quad\quad$ **find** $u \leq q_S$ **suchthat defined**$(m[u]) \wedge (m' = m[u])$

$\quad\quad$ **then end else event** *forge*

$\quad$)

$)$

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

Replace each occurrence of $hash(M)$ with a lookup in the arguments of previous calls to $hash$.
– If M is found, return the same result as the previous result.
– Otherwise, pick a new random number and return it.

For instance, **return**$(hash(x))$ is replaced with
find **suchthat defined**(m', r_{30}) && $(x = m')$ **then**
  return(r_{30})
$\oplus$ $@i_1 \leq q_S$ **suchthat defined**$(m[@i_1], r_{32}[@i_1])$
  && $(x = m[@i_1])$ **then return**$(r_{32}[@i_1])$
$\oplus$ $@i_2 \leq q_H$ **suchthat defined**$(x[@i_2], r_{34}[@i_2])$
  && $(x = x[@i_2])$ **then return**$(r_{34}[@i_2])$
else
  $r_{34} \stackrel{R}{\leftarrow} D$; **return**$(r_{34})$

Proof of FDH step 2: simplify

(* forged signature? *)
$OT(m' : bitstring, s : D) :=$
find **suchthat** **defined**(m', r_{30}) && $(m' = m')$ **then**
  if $f(pk, s) = r_{30}$ **then**
  find $u \leq q_S$ **suchthat** **defined**$(m[u])$ && $(m' = m[u])$ **then** **end** **else** **event** *forge*
$\oplus$ $@i_5 \leq q_S$ **suchthat** **defined**$(m[@i_5], r_{32}[@i_5])$ && $(m' = m[@i_5])$ **then**
  if $f(pk, s) = r_{32}[@i_5]$ **then**
  find $u \leq q_S$ **suchthat** **defined**$(m[u])$ && $(m' = m[u])$ **then** **end** **else** **event** *forge*
$\oplus$ $@i_6 \leq q_H$ **suchthat** **defined**$(x[@i_6], r_{34}[@i_6])$ && $(m' = x[@i_6])$ **then**
  if $f(pk, s) = r_{34}[@i_6]$ **then**
  find $u \leq q_S$ **suchthat** **defined**$(m[u])$ && $(m' = m[u])$ **then** **end** **else** **event** *forge*
else
  $r_{30} \stackrel{R}{\leftarrow} D$;
  if $f(pk, s) = r_{30}$ **then**
  find $u \leq q_S$ **suchthat** **defined**$(m[u])$ && $(m' = m[u])$ **then** **end** **else** **event** *forge*

r_{30} is not defined at the third line, so the first blue part is removed.
The red test always succeeds, so the second blue part becomes **end**.

Proof of FDH step 3: substitute *sk* with its value

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

Proof of FDH step 4: permutation

(* signature oracle *)
foreach $i_S \leq q_S$ **do**
$OS(m : bitstring) :=$
find **suchthat defined**(m', r_{30}) && $(m = m')$ **then**
  return$(mf(skgen(r), r_{30}))$
$\oplus$ $@i_3 \leq q_S$ **suchthat defined**$(m[@i_3], r_{32}[@i_3])$ && $(m = m[@i_3])$ **then**
  return$(mf(skgen(r), r_{32}[@i_3]))$
$\oplus$ $@i_4 \leq q_H$ **suchthat defined**$(x[@i_4], r_{34}[@i_4])$ && $(m = x[@i_4])$ **then**
  return$(mf(skgen(r), r_{34}[@i_4]))$
else
  $r_{32} \stackrel{R}{\leftarrow} D$;
  return$(mf(skgen(r), r_{32}))$

$r_i \stackrel{R}{\leftarrow} D$ becomes $y_i \stackrel{R}{\leftarrow} D$,
$mf(skgen(r), r_i)$ becomes y_i,
r_i becomes $f(pkgen(r), y_i)$

Proof of FDH step 5: simplify

(* forged signature? *)

$OT(m' : bitstring, s : D) :=$

find $@i_5 \leq q_S$ **suchthat defined**$(m[@i_5], r_{32}[@i_5])$ && $(m' = m[@i_5])$ **then end**

$\oplus$ $@i_6 \leq q_H$ **suchthat defined**$(x[@i_6], r_{34}[@i_6])$ && $(m' = x[@i_6])$ **then**

  if $(f(pk, s) = f(pkgen(r), y_{34}[@i_6]))$ **then**

  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*

else

  $y_{30} \stackrel{R}{\leftarrow} D;$

  if $(f(pk, s) = f(pkgen(r), y_{30}))$ **then**

  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*

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

Proof of FDH step 6: one-wayness

(* forged signature? *)
$OT(m' : bitstring, s : D) =$
find $@i_5 \leq q_S$ **suchthat defined**$(m[@i_5], r_{32}[@i_5])$ && $(m' = m[@i_5])$ **then end**
$\oplus$ $@i_6 \leq q_H$ **suchthat defined**$(x[@i_6], r_{34}[@i_6])$ && $(m' = x[@i_6])$ **then**
  if $s = y_{34}[@i_6]$ **then**
  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*
else
  $y_{30} \stackrel{R}{\leftarrow} D;$
  if $s = y_{30}$ **then**
  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*

$s = y_i$ becomes **find** $@j_i \leq q_S$ **suchthat defined**$(k_i[@j_i])$
then $s = y_i$ **else** *false*.

In hash oracle, $f(pkgen(r), y_i)$ becomes $f'(pkgen'(r), y_i)$.

In signature oracle, y_i becomes $k_i : bitstring \leftarrow mark; y_i$.

Difference of probability: $(q_H + q_S + 1)\mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t + (q_H + q_S)t_f)$.

Proof of FDH step 7: simplify (1)

(* forged signature? *)
$OT(m' : bitstring, s : D) :=$
find $@i_5 \leq q_S$ **suchthat defined**$(m[@i_5], r_{32}[@i_5])$ && $(m' = m[@i_5])$ **then end**
$\oplus$ $@i_6 \leq q_H$ **suchthat defined**$(x[@i_6], r_{34}[@i_6])$ && $(m' = x[@i_6])$ **then**
  find $@j_{34} \leq q_S$ **suchthat defined**$(k_{34}[@j_{34}])$ && $(@i_4[@j_{34}] = @i_6)$ **then**
  if $s = y_{34}[@i_6]$ **then**
  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*
else
  $y_{30} \stackrel{R}{\leftarrow} D;$
  find $@j_{30} \leq q_S$ **suchthat defined**$(k_{30}[@j_{30}])$ **then**
  if $s = y_{30}$ **then**
  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end 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? *)
$OT(m' : bitstring, s : D) :=$
...
$\oplus$ $@i_6 \leq q_H$ **suchthat defined**$(x[@i_6], r_{34}[@i_6])$ && $(m' = x[@i_6])$ **then**
find $@j_{34} \leq q_S$ **suchthat defined**$(k_{34}[@j_{34}])$ && $(@i_4[@j_{34}] = @i_6)$ **then**
if $s = y_{34}[@i_6]$ **then**
find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*

Definition of k_{34}:

foreach $i_S \leq q_S$ **do**
$OS(m : bitstring) :=$
...
$\oplus$ $@i_4 \leq q_H$ **suchthat defined**$(x[@i_4], y_{34}[@i_4])$ && $(m = x[@i_4])$ **then**
let $k_{34} : bitstring = mark$ **in** ...

When $k_{34}[@j_{34}]$ is defined, $m[@j_{34}]$ is defined and
$m[@j_{34}] = x[@i_4[@j_{34}]] = x[@i_6] = m'$
so the red test succeeds with $j = @j_{34}$.

Proof of FDH step 7: simplify (3)

(* forged signature? *)
$OT(m' : bitstring, s : D) :=$
...
  $y_{30} \xleftarrow{R} D;$
  find $@j_{30} \leq q_S$ **suchthat defined**$(k_{30}[@j_{30}])$ **then**
  if $s = y_{30}$ **then**
  find $u \leq q_S$ **suchthat defined**$(m[u])$ && $(m' = m[u])$ **then end else event** *forge*

Definition of k_{30}:

foreach $i_S \leq q_S$ **do**
$OS(m : bitstring) :=$
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 red test succeeds with $j = @j_{30}$.

Other examples: Encryption schemes

Our prover has successfully proved many protocols and the following properties of encryption schemes:

- $\mathcal{E}(m, r) = f(r) \| hash(r) \; xor \; m$ is IND-CPA.
- $\mathcal{E}(m, r) = f(r) \| hash(r) \; xor \; m \| hash'(hash(r) \; xor \; m, r)$ is IND-CCA2.
- With an improved treatment of the equational theory of *xor*, we believe that it could also show that $\mathcal{E}(m, r) = f(r) \| hash(r) \; xor \; m \| hash'(m, r)$ is IND-CCA2.

The proofs of these encryption schemes use a manual mode of the prover, in which the user indicates the main game transformations to perform.

Conclusion

We have presented a new tool that can prove automatically the security of cryptographic primitives and protocols.

- The security assumptions are given as observational equivalences.
  The manual proof of these equivalences is done once for all proofs, and has already been done for many primitives.
- The protocol or scheme to prove is specified using a syntax close to the notations classically used in cryptography.
- The prover provides a sequence of indistinguishable games which leads to a final game in which the adversary has advantage 0.
- The user is allowed (but does not have) to interact with the prover to make it follow a specific sequence of games.

Details at `http://www.di.ens.fr/~blanchet/cryptoc/FDH/`