

Automated Security Proofs with Sequences of Games

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

March 2008

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, that **events** can be executed only with negligible probability, and that the execution of certain events implies the execution of other events.
- 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).

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:

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

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:

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

Formalizing the security of a signature scheme (2)

Signature oracle:

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

This oracle can be called at most q_S times:

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

Formalizing the security of a signature scheme (2)

Signature oracle:

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

This oracle can be called at most q_S times:

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

In fact, this is an abbreviation for:

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

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 .

Formalizing the security of a signature scheme (3)

Test oracle:

```
 $OT(m' : \text{bitstring}, s : D) :=$  if  $\text{verify}(m', pk, s)$  then  
  find  $u \leq q_S$  suchthat  $(\text{defined}(m[u]) \wedge m' = m[u])$   
  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( $r$ ); return( $pk$ );  
(foreach  $i_S \leq q_S$  do OS( $m$  : bitstring) := return(sign( $sk$ ,  $m$ ))  
| OT( $m'$  : bitstring,  $s$  :  $D$ ) := if verify( $m'$ ,  $pk$ ,  $s$ ) then  
  find  $u \leq q_S$  suchthat (defined( $m[u]$ )  $\wedge$   $m' = m[u]$ )  
  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).

```
foreach  $i_H \leq q_H$  do  $OH(x : \text{bitstring}) := \text{return}(\text{hash}(x))$   
|  $Ogen() := r \xleftarrow{R} \text{seed}; pk \leftarrow pkgen(r); sk \leftarrow skgen(r); \text{return}(pk);$   
  (foreach  $i_S \leq q_S$  do  $OS(m : \text{bitstring}) := \text{return}(\text{invf}(sk, \text{hash}(m)))$ )  
|  $OT(m' : \text{bitstring}, s : D) := \text{if } f(pk, s) = \text{hash}(m') \text{ then}$   
  find  $u \leq q_S$  suchthat ( $\text{defined}(m[u]) \wedge m' = m[u]$ )  
  then end 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.

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 $p'(t) = p(t + t_C)$.

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** $G_0 \approx_{p_1} G_1 \approx_{p_2} \dots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1 + \dots + p_m} G_m$.

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

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 $\text{Succ}_{\mathcal{P}}^{\text{ow}}(t)$ when the adversary runs in time t .

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

LR executes **event** *invert* with probability at most $\text{Succ}_{\mathcal{P}}^{\text{ow}}(t)$ 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:

$$\begin{aligned} & \text{foreach } i_k \leq n_k \text{ do } r \stackrel{R}{\leftarrow} \text{seed}; (Opk() := \text{return}(pkgen(r))) \\ & \quad | \text{foreach } i_f \leq n_f \text{ do } x \stackrel{R}{\leftarrow} D; (Oy() := \text{return}(f(pkgen(r), x))) \\ & \quad \quad | \text{foreach } i_1 \leq n_1 \text{ do } Oeq(x' : D) := \text{return}(x' = x) \\ & \quad \quad | OX() := \text{return}(x))) \\ & \approx_{n_k \times n_f \times \text{ Succ}_P^{\text{ow}}(t + (n_k n_f - 1)t_f + (n_k - 1)t_{pkgen}} \\ & \text{foreach } i_k \leq n_k \text{ do } r \stackrel{R}{\leftarrow} \text{seed}; (Opk() := \text{return}(pkgen'(r))) \\ & \quad | \text{foreach } i_f \leq n_f \text{ do } x \stackrel{R}{\leftarrow} D; (Oy() := \text{return}(f'(pkgen'(r), x))) \\ & \quad \quad | \text{foreach } i_1 \leq n_1 \text{ do } Oeq(x' : D) := \\ & \quad \quad \quad \text{if defined}(k) \text{ then return}(x' = x) \text{ else return}(false) \\ & \quad \quad | OX() := k \leftarrow \text{mark}; \text{return}(x))) \end{aligned}$$

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:

```
foreach  $i_k \leq n_k$  do  $r \stackrel{R}{\leftarrow} \text{seed}$ ; ( $\text{Opk}() := \text{return}(pkgen(r))$ )  
  | foreach  $i_f \leq n_f$  do  $x \stackrel{R}{\leftarrow} D$ ; ( $\text{Oant}() := \text{return}(invf(skgen(r), x))$ )  
    | ( $\text{Oim}() := \text{return}(x)$ )  
 $\approx_0$  foreach  $i_k \leq n_k$  do  $r \stackrel{R}{\leftarrow} \text{seed}$ ; ( $\text{Opk}() := \text{return}(pkgen(r))$ )  
  | foreach  $i_f \leq n_f$  do  $x \stackrel{R}{\leftarrow} D$ ; ( $\text{Oant}() := \text{return}(x)$ )  
    | ( $\text{Oim}() := \text{return}(f(pkgen(r), x))$ )
```

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:

```
foreach  $i_h \leq n_h$  do  $OH(x : \text{bitstring}) := \mathbf{return}(hash(x))$   
 $\approx_0$  foreach  $i_h \leq n_h$  do  $OH(x : \text{bitstring}) :=$   
  find  $u \leq n_h$  suchthat  $(\mathbf{defined}(x[u], r[u]) \wedge x = x[u])$   
  then return}(r[u]) else  $r \stackrel{R}{\leftarrow} D; \mathbf{return}(r)$ 
```

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

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

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.

- 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 steps for the proof of FDH

- Apply security of the hash function.
- Simplify.
- Remove assignments to sk , substituting $skgen(r)$ for sk .
(This transformation is advised by the next transformation.)
- Apply invariance of uniform distributions by permutations.
- Simplify.
- Apply one-wayness.
- Simplify.
- Success: the obtained game never executes **event** *forge*.

Other examples: Encryption schemes

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

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

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.cryptoverif.ens.fr/>