A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris

January 2006

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; currently attracts much attention.

- 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:

- proves secrecy and non-injective correspondence properties.
- handles various cryptographic primitives: macs (message authentication codes), stream and block ciphers, 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 (practical security).

Produced proofs

As in Shoup's 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.
  The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property can be read directly on it.
  (The advantage of the adversary is 0 for this game.)

A game is formalized as an extension of the pi calculus with function symbols and arrays.

Process calculus for games: terms

Inspired by:
– the calculus of [Lincoln, Mitchell, Mitchell, Scedrov]
– the calculus of [Laud, CCS'05]

Extension to arrays.

$M ::=$	terms
$x, y, z, x[M_1, \ldots, M_n]$	variable
$f(M_1, \ldots, M_n)$	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
$\mathbf{newChannel}\ c; Q$	restriction for channels
$c(x_1 : T_1, \ldots, x_m : T_m); P$	input

$P ::=$	output process
$\overline{c}\langle M_1, \ldots, M_m \rangle; Q$	output
$\mathbf{event}\ e(M); P$	event
$\mathbf{new}\ x : T; P$	random number generation (uniform)
$\mathbf{let}\ x : T = M\ \mathbf{in}\ P$	assignment
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	conditional
$\mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j], \ldots)\ \&\&\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	array lookup

Process calculus for games: semantics

The semantics is purely probabilistic (no non-determinism).

The runtime of processes is polynomial in the security parameter:
– polynomial number of copies of processes
– length of messages on channels bounded by polynomials

A semantic configuration contains
– an output process (the process currently being executed)
– a set of input processes (waiting for messages)

When executing the output $\overline{c}\langle M_1, \ldots, M_m\rangle; Q$:
– we randomly chose an input of the same arity on the same channel in the set of waiting input processes $c(x_1 : T_1, \ldots, x_m : T_m); P$,
– store messages $M_1, \ldots, M_m$ in variables $x_1, \ldots, x_m$,
– schedule P for execution and add Q to the set of waiting input processes.

Arrays

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

$$!^{i \leq N} \mathbf{let}\ x = M\ \mathbf{in}\ \ldots$$

in fact defines $x[i]$, for i in $1, \ldots, N$. Under $!^{i \leq N}$, we write x for $x[i]$.

Only variables with the current indexes can be assigned.

Variables may be defined at several places, but only one definition can be executed for the same indexes.
($\mathbf{if}\ \ldots\ \mathbf{then\ let}\ x = M\ \mathbf{in}\ P\ \mathbf{else\ let}\ x = M'\ \mathbf{in}\ P'$ is ok)

Arrays (continued)

find performs an array lookup:

$$!^{i \leq N} \mathbf{let}\ x = M\ \mathbf{in}\ P$$
$$\mid !^{i' \leq N'} c(y : T) \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j])\ \&\&\ y = x[j]\ \mathbf{then}\ \ldots$$

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 sessions satisfy the condition of the **find**,
the returned index is chosen randomly, with uniform probability.

Main notion of security: 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.

MACs: security definition

A mac takes as input a message and a secret key $mac(m, k)$.
It comes with a checking function $check$ such that

$$check(m, k, mac(m, k)) = true$$

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 $check$ has a negligible probability to forge a mac:

$\Pr[check(m, k, t) \mid k \stackrel{R}{\leftarrow} kgen; (m, t) \leftarrow \mathcal{A}^{mac(.,k),check(.,k,.)}]$ is negligible when the adversary $\mathcal{A}$ has not called the mac oracle on message m.

MACs: intuitive implementation

By the previous definition, the adversary has a negligible probability of forging a correct mac.

So when checking a mac with $check(m, k, t)$ and k is secret, the check 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 check with an array lookup:
if the call to mac is $mac(x, k)$, we replace $check(m, k, t)$ with

$$\begin{aligned}&\textbf{find } j \leq N \textbf{ suchthat } \textbf{defined}(x[j]) \;\&\&\\ &\qquad (m = x[j]) \;\&\&\; check(m, k, t) \textbf{ then true else false}\end{aligned}$$

Furthermore, we use primed function symbols after the transformation, so that it is not done again.

MACs: formal implementation

$check(m, kgen(r), mac(m, kgen(r))) = \mathbf{true}$

$!^{N''}\mathbf{new}\ r : keyseed; ($
$\quad !^{N}(x : bitstring) \rightarrow mac(x, kgen(r)),$
$\quad !^{N'}(m : bitstring, t : macstring) \rightarrow check(m, kgen(r), t))$

$\approx_{N'' \times Pmac(N,N',\mathbf{time})}$
$!^{N''}\mathbf{new}\ r : keyseed; ($
$\quad !^{N}(x : bitstring) \rightarrow mac'(x, kgen'(r)),$
$\quad !^{N'}(m : bitsting, t : macstring) \rightarrow \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j])\ \&\&$
$\qquad (m = x[j])\ \&\&\ check'(m, kgen'(r), t)\ \mathbf{then\ true\ else\ false})$

The prover understands such specifications of primitives.

MACs: formal implementation

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

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

Stream ciphers

Similarly, the security of stream ciphers is expressed as follows:

$$dec(enc(m, kgen(r), r'), kgen(r)) = m$$

$$!^{N'}\mathbf{new}\ r : keyseed; !^{N}(x : bitstring) \rightarrow \mathbf{new}\ r' : IV; enc(x, kgen(r), r')$$

$$\approx_{N' \times Penc(N, \mathbf{time})}$$

$$!^{N'}\mathbf{new}\ r : keyseed; !^{N}(x : bitstring) \rightarrow \mathbf{new}\ r' : IV; enc'(Z(x), kgen'(r), r')$$

A stream cipher is non-deterministic, length-revealing, resistant to Chosen Plaintext Attacks (CPA).

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all $x : nonce$, $Z(x) = Znonce$, ...).

Syntactic transformations

- 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.)

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

- Move new: move restrictions downwards in the game as much as possible, when there is no array reference to them.
  (Moving $\mathbf{new}\ x : T$ under a $\mathbf{if}$ or a $\mathbf{find}$ duplicates it.
  A subsequent single assignment renaming will distinguish cases.)

Simplification and elimination of collisions

Terms are simplified according to equalities that come from:

- Assignments: $\mathbf{let}\ x = M\ \mathbf{in}\ P$ implies that $x = M$ in P
- Tests: $\mathbf{if}\ M = N\ \mathbf{then}\ P$ implies that $M = N$ in P
- Definitions of cryptographic primitives
- When a $\mathbf{find}$ guarantees that $x[j]$ is defined, equalities that hold at definition of x also hold under the find (after substituting j for array indexes)
- Elimination of collisions: for example, if N is created by $\mathbf{new}$, $N[i] = N[j]$ implies $i = j$, up to negligible probability

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:
There is a set S such that
– x is defined by $\mathbf{new}\ x[i] : T$ and $x \in S$, or by $\mathbf{let}\ x[i] : T = y[\ldots]$ where y is defined by $\mathbf{new}\ y[j] : T$ and $y \in S$.
– all accesses to $z \in S$ are of the form $\mathbf{let}\ z'[i] : T = z[\ldots]$ for $z' \in S$.

So only the variables in S depend on the secrets. The output messages and the control-flow do not depend on the secrets.

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 $x[i']$ for $i \neq i'$ come from different restrictions.

Proof of security properties: correspondence assertions

Correspondence: $e_1(M_1) \wedge \ldots \wedge e_n(M_n) \Rightarrow M$ where M is a logical formula formed with conjunctions and disjunctions of events and other logical predicates.

Criterion for proving correspondences: For each occurrence of the events $e_1(\ldots), \ldots, e_n(\ldots)$ in the game, compute facts that must be true at that point, and prove M from these facts.

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.

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
– FDH signature and encryption

Shared-key encryption is implemented as encrypt-then-mac, using a IND-CPA stream cipher for encryption.
(For Otway-Rees and Woo-Lam shared-key, we also considered a SPRP block cipher.)
Public-key encryption is assumed to be IND-CCA2.

We prove secrecy of session keys and non-injective 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.

- The public-key protocols, as well as showing that FDH encryption is IND-CPA, need manual proofs.
  (Give the cryptographic proof steps and sometimes single assignment renaming instructions.)

- Runtime: 13 ms to 48 s, average: 6 s

Results (2)

- Cases in which the prover fails although the property is true:

  – Needham-Schroeder public-key when the exchanged key is the nonce N_A.
  – Needham-Schroeder shared-key with $\{N_B\}_K; \{N_B{-}1\}_K$ (loop).
  – Woo-Lam shared-key with a block cipher.
  – Showing that FDH encryption is IND-CCA2.

  Most problems come from missing simplifications.

Example of the FDH signature

hash hash function (in the random oracle model)
$\mathsf{f}(pk, m)$ one-way trapdoor permutation, with inverse $\mathsf{mf}(sk, m)$.

We define a signature scheme as follows:
– signature $\mathsf{sign}(m, sk) = \mathsf{mf}(sk, \mathsf{hash}(m))$
– verification $\mathsf{check}(m, pk, s) = (\mathsf{f}(pk, s) = \mathsf{hash}(m))$

Our goal is to show that this signature scheme is secure against existential forgery under chosen message attacks.

FDH: security of a hash function

A hash function is equivalent to a "random function":
a function that returns a new random number when it is a called on a new argument, and returns the same result when it is called on the same argument.

$!^N\ (x : bitstring)\ \rightarrow\ \mathsf{hash}(x)\ [all]$

$\approx_0$

$!^N\ (x : bitstring)\ \rightarrow$

$\quad \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j], r[j])\ \&\&\ (x = x[j])$

$\quad \mathbf{then}\ r[j]$

$\quad \mathbf{else\ new}\ r : hasht;\ r$

FDH: permutation

mf is the inverse of f:

$$\forall r : keyseed, x : hasht; \mathsf{mf}(\mathsf{skgen}(r), \mathsf{f}(\mathsf{pkgen}(r), x)) = x$$

f is injective:

$$\forall k : key, x : hasht, x2 : hasht; (\mathsf{f}(k, x) = \mathsf{f}(k, x2)) = (x = x2)$$

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

$!^N$ **new** $r : keyseed$; (
  $!^{N2}$ () $\rightarrow \mathsf{pkgen}(r)$,
  $!^{N3}$ **new** $y : hasht$; ($!^{N4}$ () $\rightarrow \mathsf{mf}(\mathsf{skgen}(r), y)$, $!^{N5}$ () $\rightarrow y$))

$\approx_0$

$!^N$ **new** $r : keyseed$; (
  $!^{N2}$ () $\rightarrow \mathsf{pkgen}(r)$,
  $!^{N3}$ **new** $y' : hasht$; ($!^{N4}$ () $\rightarrow y'$, $!^{N5}$ () $\rightarrow \mathsf{f}(\mathsf{pkgen}(r), y')$))

FDH: one-wayness

$$
\begin{array}{l}
!^{N}\ \mathbf{new}\ r : keyseed;\ (\\
\quad !^{N2}\ ()\ \rightarrow\ \mathsf{pkgen}(r),\\
\quad !^{N3}\ \mathbf{new}\ y : hasht;\ (\\
\qquad !^{N4}\ ()\ \rightarrow\ \mathsf{f}(\mathsf{pkgen}(r), y),\\
\qquad !^{N5}\ (x : hasht)\ \rightarrow\ (x = y),\\
\qquad !^{N6}\ ()\ \rightarrow\ y))
\end{array}
$$

$$
\begin{array}{l}
\approx_{N \times N3 \times P_OW(\mathbf{time})}\\
!^{N}\ \mathbf{new}\ r : keyseed;\ (\\
\quad !^{N2}\ ()\ \rightarrow\ \mathsf{pkgen2}(r),\\
\quad !^{N3}\ \mathbf{new}\ y : hasht;\ (\\
\qquad !^{N4}\ ()\ \rightarrow\ \mathsf{f2}(\mathsf{pkgen2}(r), y),\\
\qquad !^{N5}\ (x : hasht)\ \rightarrow\ \mathbf{find}\ j \leq N6\ \mathbf{suchthat}\ \mathbf{defined}(k[j])\ \&\&\ \mathsf{true}\\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ x = y\ \mathbf{else}\ \mathsf{false},\\
\qquad !^{N6}\ ()\ \rightarrow\ \mathbf{let}\ k : bitstring = \mathsf{mark}\ \mathbf{in}\ y))
\end{array}
$$

FDH: initial game

$start()$; **new** $r : keyseed$;
let $sk : key = \mathsf{skgen}(r)$ **in**
let $pk : key = \mathsf{pkgen}(r)$ **in** $\overline{c9}\langle pk\rangle$;
((* hash oracle *)
  $!^{!_27 \leq N}\ c1[!_27](x : bitstring);\ \overline{c2[!_27]}\langle \mathsf{hash}(x)\rangle$
| (* signature oracle *)
  $!^{!_28 \leq N2}\ c3[!_28](m : bitstring);\ \overline{c4[!_28]}\langle \mathsf{mf}(sk, \mathsf{hash}(m))\rangle$
| (* forged signature? *)
  $c7(m2 : bitstring, s : hasht)$;
  if $(\mathsf{f}(pk, s) = \mathsf{hash}(m2))$ **then**
  find $j \leq N2$ **suchthat defined**$(m[j])$ && $(m2 = m[j])$ **then**
    0
  else
    event bad
)

FDH step 1: apply the security of the hash function

Replace each occurrence of $\mathsf{hash}(M)$ 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, $\overline{c2[!_27]}\langle \mathsf{hash}(x)\rangle$ is replaced with
find suchthat defined$(m2, r_30)$ && $(x = m2)$ **then**
  $\overline{c2[!_27]}\langle r_30\rangle$
orfind @$i1 \leq N2$ **suchthat defined**$(m[@i1], r_32[@i1])$
  && $(x = m[@i1])$ **then** $\overline{c2[!_27]}\langle r_32[@i1]\rangle$
orfind @$i2 \leq N$ **suchthat defined**$(x[@i2], r_34[@i2])$
  && $(x = x[@i2])$ **then** $\overline{c2[!_27]}\langle r_34[@i2]\rangle$
else
  new $r_34 : hasht$; $\overline{c2[!_27]}\langle r_34\rangle$

FDH step 2: simplify

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
find **suchthat** **defined**$(m2, r_30)$ && $(m2 = m2)$ **then**
  if $(\mathsf{f}(pk, s) = r_30)$ **then**
  find $j \leq N2$ **suchthat** **defined**$(m[j])$ && $(m2 = m[j])$ **then** 0 **else event** bad
orfind @$i5 \leq N2$ **suchthat** **defined**$(m[@i5], r_32[@i5])$ && $(m2 = m[@i5])$ **then**
  if $(\mathsf{f}(pk, s) = r_32[@i5])$ **then**
  find $j \leq N2$ **suchthat** **defined**$(m[j])$ && $(m2 = m[j])$ **then** 0 **else event** bad
orfind @$i6 \leq N$ **suchthat** **defined**$(x[@i6], r_34[@i6])$ && $(m2 = x[@i6])$ **then**
  if $(\mathsf{f}(pk, s) = r_34[@i6])$ **then**
  find $j \leq N2$ **suchthat** **defined**$(m[j])$ && $(m2 = m[j])$ **then** 0 **else event** bad
else
  new $r_30 : hasht$;
  if $(\mathsf{f}(pk, s) = r_30)$ **then**
  find $j \leq N2$ **suchthat** **defined**$(m[j])$ && $(m2 = m[j])$ **then** 0 **else event** bad

The red test always succeeds, so the blue part becomes 0.

$(m2 = m2)$ becomes true.

FDH step 3: substitute sk with its value

The variable sk is replaced with $\mathsf{skgen}(r)$, and the assignment $\mathbf{let}\ sk : key = \mathsf{skgen}(r)$ is removed.

This transformation is advised in order to able to apply the permutation property.

FDH step 4: permutation

(* signature oracle *)
$!^{!_28 \leq N2}$
$c3[!_28](m : bitstring)$;
find **suchthat** **defined**$(m2, r_30)$ && $(m = m2)$ **then**
  $\overline{c4[!_28]}\langle \mathsf{mf}(\mathsf{skgen}(r), r_30)\rangle$
orfind $@i3 \leq N2$ **suchthat** **defined**$(m[@i3], r_32[@i3])$ && $(m = m[@i3])$ **then**
  $\overline{c4[!_28]}\langle \mathsf{mf}(\mathsf{skgen}(r), r_32[@i3])\rangle$
orfind $@i4 \leq N$ **suchthat** **defined**$(x[@i4], r_34[@i4])$ && $(m = x[@i4])$ **then**
  $\overline{c4[!_28]}\langle \mathsf{mf}(\mathsf{skgen}(r), r_34[@i4])\rangle$
else
  new $r_32 : hasht$;
  $\overline{c4[!_28]}\langle \mathsf{mf}(\mathsf{skgen}(r), r_32)\rangle$

new $r_i : hasht$ becomes **new** $y_i : hasht$,
$\mathsf{mf}(\mathsf{skgen}(r), r_i)$ becomes y_i,
r_i becomes $\mathsf{f}(\mathsf{pkgen}(r), y_i)$

FDH step 5: simplify

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
find **suchthat** **defined**(y_30) && true **then**
  if (f(pk, s) = f(pkgen(r), y_30))) **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else** **event** bad
orfind @$i5 \leq N2$ **suchthat** **defined**(m[@$i5$], r_32[@$i5$]) && ($m2 = m$[@$i5$]) **then** 0
orfind @$i6 \leq N$ **suchthat** **defined**(x[@$i6$], r_34[@$i6$]) && ($m2 = x$[@$i6$]) **then**
  if (f(pk, s) = f(pkgen(r), y_34[@$i6$])) **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else** **event** bad
else
  new $y_30 : hasht$;
  if (f(pk, s) = f(pkgen(r), y_30)) **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else** **event** bad

f(pk, s) = f(pkgen(r), y_i) becomes $s = y_i$,
knowing $pk = pkgen(r)$ and the injectivity of f:
$\forall k : key, x : hasht, x2 : hasht;$ (f(k, x) = f($k, x2$)) = ($x = x2$)

FDH step 6: one-wayness

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
find **suchthat** **defined**(y_30) && true **then**
  if $s = y_30$ **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad
orfind @$i5 \leq N2$ **suchthat** **defined**(m[@$i5$], r_32[@$i5$]) && ($m2 = m$[@$i5$]) **then** 0
orfind @$i6 \leq N$ **suchthat** **defined**(x[@$i6$], r_34[@$i6$]) && ($m2 = x$[@$i6$]) **then**
  if $s = y_34$[@$i6$] **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad
else
  new $y_30 : hasht$;
  if $s = y_30$ **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad

$s = y_i$ becomes **find** @$j_i \leq N$ **suchthat** **defined**(k_i[@j_i]) && true
then $s = y_i$ **else** false,

In hash oracle, f(pkgen(r), y_i) becomes f2(pkgen2(r), y_i),

In signature oracle, y_i becomes **let** $k_i : bitstring$ = mark **in** y_i.

Difference of probability: $(N + N2 + 1)P_OW(\mathbf{time})$.

FDH step 7: simplify

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
find **suchthat** **defined**(y_30) && true **then**
  find @$j_30 \leq N$ **suchthat** **defined**(k_30[@j_30]) && true **then**
  if $s = y_30$ **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad
orfind @$i5 \leq N2$ **suchthat** **defined**(m[@$i5$], r_32[@$i5$]) && ($m2 = m$[@$i5$]) **then** 0
orfind @$i6 \leq N$ **suchthat** **defined**(x[@$i6$], r_34[@$i6$]) && ($m2 = x$[@$i6$]) **then**
  find @$j_34 \leq N2$ **suchthat** **defined**(k_34[@j_34]) && (@$i4$[@j_34] = @$i6$) **then**
  if $s = y_34$[@$i6$] **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad
else
  new $y_30 : hasht$;
  find @$j_30 \leq N$ **suchthat** **defined**(k_30[@j_30]) && true **then**
  if $s = y_30$ **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else event** bad

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

FDH step 7: simplify (2)

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
find **suchthat** **defined**(y_30) && true **then**
  find @$j_30 \leq N$ **suchthat** **defined**(k_30[@j_30]) && true **then**
  if $s = y_30$ **then**
  find $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) **then** 0 **else** **event** bad

Definition of k_30:

$!^{!_28 \leq N2}$
$c3$[!_28]($m : bitstring$);
find **suchthat** **defined**($m2, y_30$) && ($m = m2$) **then**
  let $k_30 : bitstring$ = mark **in** . . .

When k_30[@j_30] is defined, m[@j_30] is defined and m[@j_30] = $m2$, so the test $j \leq N2$ **suchthat** **defined**($m[j]$) && ($m2 = m[j]$) succeeds with j = @j_30.
Same reasoning for the last red test.

FDH step 7: simplify (3)

(* forged signature? *)
$c7(m2 : bitstring, s : hasht)$;
...
orfind @$i6 \leq N$ **suchthat defined**(x[@$i6$], r_34[@$i6$]) && ($m2 = x$[@$i6$]) **then**
  find @$j_34 \leq N2$ **suchthat defined**(k_34[@j_34]) && (@$i4$[@j_34] = @$i6$) **then**
  if $s = y_34$[@$i6$] **then**
  find $j \leq N2$ **suchthat defined**(m[j]) && ($m2 = m$[j]) **then** 0 **else event** bad

Definition of k_34:

!$^{!_28 \leq N2}$
$c3$[!_28]($m : bitstring$);
...
orfind @$i4 \leq N$ **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[@$i6$] = $m2$
so the red test succeeds with j = @j_34.

Conclusion

Hopefully a promising approach, but still some work to do:

- Extension to other cryptographic primitives, in particular Diffie-Hellman.
- Improvements to the simplification algorithm.
- Extension to other security properties: semantic security of the key, injective correspondences, . . .
- More experiments.

More information: http://www.di.ens.fr/~blanchet/cryptoc-eng.html

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.

Thank you for your attention.

Questions?