

CryptoVerif: A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet

CNRS, École Normale Supérieure, INRIA, Paris

May 2010

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**:

- proves **secrecy** and **correspondence** properties.
- provides a **generic** method for specifying properties of **cryptographic primitives** which handles symmetric encryption, MACs, public-key encryption, signatures, hash functions, CDH, DDH, ...
- 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

As in 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.

Initial game

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

- adapted from the pi calculus.
- **purely probabilistic** semantics (no non-determinism).
- all processes run in **polynomial time**.

Example: hashed El Gamal in the random oracle model (Avik Chaudhuri)
 Cyclic group G of cardinal q , with generator γ .

- key generation: secret key $x \xleftarrow{R} \mathbb{Z}_q^*$, public key $\alpha \leftarrow \gamma^x$
- encryption:
 $\mathcal{E}_\alpha(m) = y \xleftarrow{R} \mathbb{Z}_q^*; \beta \leftarrow \gamma^y; \delta \leftarrow \alpha^y; h \leftarrow H(\delta); \mathbf{return}(\beta, h \oplus m)$
- decryption: $\mathcal{D}_x(\beta, c) = \delta \leftarrow \beta^x; h \leftarrow H(\delta); \mathbf{return}(h \oplus c)$

Prove semantic security.

Initial game (1): key generation

- key generation: secret key $x \xleftarrow{R} \mathbb{Z}_q^*$, public key $\alpha \leftarrow \gamma^x$

start();

new $x : Z$;

let $alpha = exp(g, x)$ **in**

$\overline{CPK} \langle alpha \rangle$

Initial game (2): semantic security test

- The adversary chooses two messages m_0, m_1 .
- It should be unable to distinguish $\mathcal{E}_\alpha(m_0)$ from $\mathcal{E}_\alpha(m_1)$
- That is, we choose a random bit b , output $\mathcal{E}_\alpha(m_b)$, and the adversary should be unable to guess b .

$c_E(m_0 : D, m_1 : D);$

new $b : \text{bool};$

let $m = \text{choose}(b, m_0, m_1)$ **in**

$\overline{c_{Eret}} \langle \mathcal{E}_\alpha(m) \rangle$

Initial game (2): semantic security test

- encryption:

$$\mathcal{E}_\alpha(m) = y \stackrel{R}{\leftarrow} \mathbb{Z}_q^*; \beta \leftarrow \gamma^y; \delta \leftarrow \alpha^y; h \leftarrow H(\delta); \mathbf{return}(\beta, h \oplus m)$$

$c_E(m_0 : D, m_1 : D);$

new $b : \mathit{bool};$

let $m = \mathit{choose}(b, m_0, m_1)$ **in**

new $y : Z;$

let $\mathit{beta} = \mathit{exp}(g, y)$ **in**

let $\mathit{delta} = \mathit{exp}(\mathit{alpha}, y)$ **in**

let $h = H(\mathit{delta})$ **in**

let $v = \mathit{xor}(h, m)$ **in**

$\overline{c_{Eret}}(\mathit{beta}, v)$

Initial game (3): hash oracle

- The adversary also has access to a hash oracle:
given x , this oracle returns $H(x)$.
- Let n_H be the maximum number of calls to this oracle.

hashoracle = $!^{i_H \leq n_H}$
 $c_H(x : G);$
 $\overline{c}_H \langle H(x) \rangle$

Initial game: full game

```
start();
new  $x : Z$ ;
let  $alpha = exp(g, x)$  in
 $\overline{c_{PK}}$  $\langle alpha \rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ;
  new  $b : bool$ ;
  let  $m = choose(b, m_0, m_1)$  in
  new  $y : Z$ ;
  let  $beta = exp(g, y)$  in
  let  $delta = exp(alpha, y)$  in
  let  $h = H(delta)$  in
  let  $v = xor(h, m)$  in
   $\overline{c_{Eret}}$  $\langle beta, v \rangle$ 
| hashoracle)
```

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:

- xor is **exclusive or**.
One-time pad
Algebraic properties
- H is a hash function in the **random oracle model**.
- exp is modular exponentiation, with the **CDH assumption** (computational Diffie-Hellman)

Security property to prove

In the example:

- **Secrecy** of b : b is indistinguishable from a random boolean.

CryptoVerif can also verify correspondence assertions (authentication).

Demo

Demo

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 .

Exclusive or

- **One-time pad**: if a is random and not used elsewhere, $a \oplus x$ is indistinguishable from a random number.

$$!i \leq N \text{ new } a : D; O_{xor}(x : D) := xor(a, x)$$

$$\approx_0$$

$$!i \leq N \text{ new } a : D; O_{xor}(x : D) := a$$

CryptoVerif replaces $xor(a, x)$ with a .

- **Algebraic properties**:

$xor(D, D) : D$ commutative

$xor(x, xor(x, y)) = y$

$(xor(x, z) = xor(y, z)) = (x = y)$

xor self-cancels

injectivity

Some algebraic properties (associativity) cannot be taken into account by CryptoVerif.

Hash function in the 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,
- returns the same result when it is called on the same argument.

Cryptographers usually code such a function by storing the previous arguments and results in a **list L** :

$$H(x)$$

becomes

- 1 Look for an element (x', r') in L with $x' = x$.
- 2 If it is found, return r' .
- 3 Otherwise, pick a fresh r , add (x, r) to L , and return r .

Hash function in the 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,
- returns the same result when it is called on the same argument.

CryptoVerif uses **arrays** instead of lists:

$$!^{i_H \leq n_H} c_H(x[i_H] : G); \overline{c}_H \langle H(x[i_H]) \rangle$$

becomes

$$!^{i_H \leq n_H} c_H(x[i_H] : G);$$

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

then $\overline{c}_H \langle r[j] \rangle$

else new $r[i_H] : D; \overline{c}_H \langle r[i_H] \rangle$

Hash function in the 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,
- returns the same result when it is called on the same argument.

Arrays are implicit:

$$!^{i_H \leq n_H} c_H(x : G); \overline{c_H} \langle H(x) \rangle$$

becomes

$$!^{i_H \leq n_H} c_H(x : G);$$

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

then $\overline{c_H} \langle r[j] \rangle$

else new $r : D; \overline{c_H} \langle r \rangle$

Hash function in the random oracle model

Hence the equivalence:

$$\forall i_H \leq n_H \quad Ohash(x : G) := H(x) \text{ [all]}$$

$$\approx_0$$

$$\forall i_H \leq n_H \quad Ohash(x : G) :=$$

```

find  $j \leq n_H$  suchthat  $\text{defined}(x[j], r[j]) \wedge (x = x[j])$ 
then  $r[j]$ 
else new  $r : D; r$ 

```

Hash function in the random oracle model

With an optimization for hash verification:

$$!^{i_H \leq n_H} \text{Ohash}(x : G) := H(x) \text{ [all]},$$

$$!^{i_{eq} \leq n_{eq}} \text{Oeq}(x' : G, r' : D) := r' = H(x') \text{ [all]}$$

$$\approx_{n_{eq}/|D|}$$

$$!^{i_H \leq n_H} \text{Ohash}(x : G) :=$$

find [unique] $j \leq n_H$ **suchthat** $\text{defined}(x[j], r[j]) \wedge (x = x[j])$

then $r[j]$

else new $r : D; r,$

$$!^{i_{eq} \leq n_{eq}} \text{Oeq}(x' : G, r' : D) :=$$

find [unique] $j \leq n_H$ **suchthat** $\text{defined}(x[j], r[j]) \wedge (x' = x[j])$

then $r' = r[j]$

else false

Computational Diffie-Hellman assumption

Consider a multiplicative cyclic group G of order q , with generator g . A probabilistic polynomial-time adversary has a negligible probability of **computing** g^{ab} from g , g^a , g^b , for random $a, b \in \mathbb{Z}_q$.

Computational Diffie-Hellman assumption in CryptoVerif

Consider a multiplicative cyclic group G of order q , with generator g . A probabilistic polynomial-time adversary has a negligible probability of **computing** g^{ab} from g, g^a, g^b , for random $a, b \in \mathbb{Z}_q$.

In CryptoVerif, this can be written

$$!^{i \leq N} \text{new } a : Z; \text{new } b : Z; (OA() := \text{exp}(g, a), OB() := \text{exp}(g, b), \\ !^{i' \leq N'} \text{OCDH}(z : G) := z = \text{exp}(g, \text{mult}(a, b)))$$

\approx

$$!^{i \leq N} \text{new } a : Z; \text{new } b : Z; (OA() := \text{exp}(g, a), OB() := \text{exp}(g, b), \\ !^{i' \leq N'} \text{OCDH}(z : G) := \text{false})$$

Computational Diffie-Hellman assumption in CryptoVerif

Consider a multiplicative cyclic group G of order q , with generator g . A probabilistic polynomial-time adversary has a negligible probability of **computing** g^{ab} from g, g^a, g^b , for random $a, b \in \mathbb{Z}_q$.

In CryptoVerif, this can be written

$$!^{i \leq N} \text{new } a : Z; \text{new } b : Z; (OA() := \text{exp}(g, a), OB() := \text{exp}(g, b),$$

$$!^{i' \leq N'} \text{OCDH}(z : G) := z = \text{exp}(g, \text{mult}(a, b)))$$

\approx

$$!^{i \leq N} \text{new } a : Z; \text{new } b : Z; (OA() := \text{exp}(g, a), OB() := \text{exp}(g, b),$$

$$!^{i' \leq N'} \text{OCDH}(z : G) := \text{false})$$

This is sufficient for proving **hashed El Gamal in the random oracle model**.

Computational Diffie-Hellman assumption in CryptoVerif

This model is **not sufficient** for EKE and other practical protocols.

- It assumes that a and b are chosen under the same replication.
- In practice, one participant chooses a , another chooses b , so these choices are made under different replications.

Computational Diffie-Hellman assumption in CryptoVerif

$$\begin{aligned}
 & !^{ia \leq Na} \text{ new } a : Z; (OA() := \exp(g, a), Oa() := a, \\
 & \quad !^{iaCDH \leq naCDH} OCDHa(m : G, j \leq Nb) := m = \exp(g, \text{mult}(b[j], a))), \\
 & !^{ib \leq Nb} \text{ new } b : Z; (OB() := \exp(g, b), Ob() := b, \\
 & \quad !^{ibCDH \leq nbCDH} OCDHb(m : G, j \leq Na) := m = \exp(g, \text{mult}(a[j], b))) \\
 & \approx \\
 & !^{ia \leq Na} \text{ new } a : Z; (OA() := \exp(g, a), Oa() := \text{let } ka = \text{mark in } a, \\
 & \quad !^{iaCDH \leq naCDH} OCDHa(m : G, j \leq Nb) := \\
 & \quad \quad \text{find } u \leq nb \text{ suchthat defined}(kb[u], b[u]) \wedge b[j] = b[u] \text{ then} \\
 & \quad \quad \quad m = \exp(g, \text{mult}(b[j], a)) \\
 & \quad \quad \text{else if defined}(ka) \text{ then } m = \exp(g, \text{mult}(b[j], a)) \text{ else false}), \\
 & !^{ib \leq Nb} \text{ new } b : Z; (OB() := \exp(g, b), Ob() := \text{let } kb = \text{mark in } b, \\
 & \quad !^{ibCDH \leq nbCDH} OCDHb(m : G, j \leq Na) := (\text{symmetric of } OCDHa))
 \end{aligned}$$

Computational Diffie-Hellman assumption in CryptoVerif

$\!|^{ia \leq Na}$ **new** $a : Z$; ($OA() := \exp(g, a)$, $Oa()[3] := a$,

$\!|^{iaCDH \leq naCDH}$ $OCDHa(m : G, j \leq Nb)$ [required] $:= m = \exp(g, \text{mult}(b[j],$

$\!|^{ib \leq Nb}$ **new** $b : Z$; ($OB() := \exp(g, b)$, $Ob()[3] := b$,

$\!|^{ibCDH \leq nbCDH}$ $OCDHb(m : G, j \leq Na) := m = \exp(g, \text{mult}(a[j], b)))$

\approx $(\#OCDHa + \#OCDHb) \times \max(1, e^2 \#Oa) \times \max(1, e^2 \#Ob) \times$
 $pCDH(\text{time} + (na + nb + \#OCDHa + \#OCDHb) \times \text{time}(\exp))$

$\!|^{ia \leq Na}$ **new** $a : Z$; ($OA() := \exp'(g, a)$, $Oa() := \text{let } ka = \text{mark in } a$,

$\!|^{iaCDH \leq naCDH}$ $OCDHa(m : G, j \leq Nb) :=$

find $u \leq nb$ **suchthat** $\text{defined}(kb[u], b[u]) \wedge b[j] = b[u]$ **then**

$m = \exp(g, \text{mult}(b[j], a))$

else if $\text{defined}(ka)$ **then** $m = \exp'(g, \text{mult}(b[j], a))$ **else false**),

$\!|^{ib \leq Nb}$ **new** $b : Z$; ($OB() := \exp'(g, b)$, $Ob() := \text{let } kb = \text{mark in } b$,

$\!|^{ibCDH \leq nbCDH}$ $OCDHb(m : G, j \leq Na) := (\text{symmetric of } OCDHa)$

Other declarations for Diffie-Hellman (1)

$g : G$	generator of G
$\text{exp}(G, Z) : G$	exponentiation
$\text{mult}(Z, Z) : Z$ commutative	product in \mathbb{Z}_q
$\text{exp}(\text{exp}(z, a), b) = \text{exp}(z, \text{mult}(a, b))$	$(z^a)^b = z^{ab}$
$(g^a)^b = g^{ab}$ and $(g^b)^a = g^{ba}$, equal by commutativity of <i>mult</i>	

$(\text{exp}(g, x) = \text{exp}(g, y)) = (x = y)$
 $(\text{exp}'(g, x) = \text{exp}'(g, y)) = (x = y)$

Injectivity

new $x1 : Z$; **new** $x2 : Z$; **new** $x3 : Z$; **new** $x4 : Z$;
 $\text{mult}(x1, x2) = \text{mult}(x3, x4) \approx_{1/|Z|} \text{false}$

Collision between products

Other declarations for Diffie-Hellman (2)

$$\begin{aligned} & !^{i \leq N} \mathbf{new} X : G; OX() := X \\ \approx_0 \text{ [manual]} & !^{i \leq N} \mathbf{new} x : Z; OX() := \mathit{exp}(g, x) \end{aligned}$$

This equivalence is very general, apply it only manually.

$$\begin{aligned} & !^{i \leq N} \mathbf{new} X : G; (OX() := X, !^{i' \leq N'} OXm(m : Z)[\text{required}] := \mathit{exp}(X, m)) \\ & \approx_0 \\ & !^{i \leq N} \mathbf{new} x : Z; (OX() := \mathit{exp}(g, x), !^{i' \leq N'} OXm(m : Z) := \mathit{exp}(g, \mathit{mult}(x, m))) \end{aligned}$$

This equivalence is a particular case applied only when X is inside exp , and good for automatic proofs.

$$\begin{aligned} & !^{i \leq N} \mathbf{new} x : Z; OX() := \mathit{exp}(g, x) \\ \approx_0 & !^{i \leq N} \mathbf{new} X : G; OX() := X \end{aligned}$$

And the same for exp' .

Extensions for CDH

The implementation of the support for CDH required two extensions of CryptoVerif:

- An **array index j occurs as argument** of a function.
- The equality test $m = \text{exp}(g, \text{mult}(b, a))$ typically occurs inside the condition of a **find**.
 - This **find** comes from the transformation of a hash function in the random oracle model.

After transformation, we obtain a **find inside the condition of a find**.

We added support for these constructs in CryptoVerif.

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 **new** $x : T$ under a **if** or a **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:** **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 indexes 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)

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

```

start(); new  $x : Z$ ; let  $alpha = exp(g, x)$  in  $\overline{c_{PK}}$  $\langle alpha \rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ;
  new  $b : bool$ ; let  $m = choose(b, m_0, m_1)$  in
  new  $y : Z$ ;
  let  $beta = exp(g, y)$  in
  let  $delta = exp(alpha, y)$  in
  let  $h = H(delta)$  in
  let  $v = xor(h, m)$  in
   $\overline{c_{Eret}}$  $\langle beta, v \rangle$ 
  |
   $!^{i_H \leq n_H} c_H(z : G); \overline{c_H}\langle H(z) \rangle$ 
)

```

Proof of the example: simplification

```

start(); new  $x : Z$ ; let  $alpha = exp(g, x)$  in  $\overline{c_{PK}}\langle alpha \rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ;
  new  $b : bool$ ; let  $m = choose(b, m_0, m_1)$  in
  new  $y : Z$ ;
  let  $beta = exp(g, y)$  in
  let  $delta = exp(g, mult(x, y))$  in
  let  $h = H(delta)$  in
  let  $v = xor(h, m)$  in
   $\overline{c_{Eret}}\langle beta, v \rangle$ 
|
   $!_{H \leq n_H} c_H(z : G); \overline{c_H}\langle H(z) \rangle$ 
)

 $exp(exp(z, a), b) = exp(z, mult(a, b))$ 

```

Proof of the example: random oracle

```

start(); new  $x : Z$ ; let  $alpha = exp(g, x)$  in  $\overline{c_{PK}}\langle alpha \rangle$ ;
(  $c_E(m_0 : D, m_1 : D)$ ; new  $b : bool$ ; let  $m = choose(b, m_0, m_1)$  in
  new  $y : Z$ ; let  $beta = exp(g, y)$  in let  $delta = exp(g, mult(x, y))$  in
  find[unique]  $j_1 \leq n_H$  suchthat defined( $z[j_1], r_2[j_1]$ )  $\wedge$  ( $delta = z[j_1]$ ) then
    let  $v : D = xor(r_2[j_1], m)$  in  $\overline{c_{Eret}}\langle beta, v \rangle$ 
  else
    new  $r_1 : D$ ; let  $v : D = xor(r_1, m)$  in  $\overline{c_{Eret}}\langle beta, v \rangle$ 
|
   $!^{j_H \leq n_H} c_H(z : G)$ ;
  find[unique]  $j_2 \leq n_H$  suchthat defined( $z[j_2], r_2[j_2]$ )  $\wedge$  ( $z = z[j_2]$ ) then
     $\overline{c_H}\langle r_2[j_2] \rangle$ 
   $\oplus$  suchthat defined( $delta, b, r_1$ )  $\wedge$  ( $z = delta$ ) then
     $\overline{c_H}\langle r_1 \rangle$ 
  else
    new  $r_2 : D$ ;  $\overline{c_H}\langle r_2 \rangle$ )

```

Proof of the example: remove assignments on *delta*

```

start(); new  $x : Z$ ; let  $alpha = exp(g, x)$  in  $\overline{c_{PK}}\langle alpha \rangle$ ;
( $c_E(m_0 : D, m_1 : D)$ ; new  $b : bool$ ; let  $m = choose(b, m_0, m_1)$  in
new  $y : Z$ ; let  $beta = exp(g, y)$  in
find[unique]  $j_1 \leq n_H$  suchthat  $defined(z[j_1], r_2[j_1]) \wedge$ 
  ( $exp(g, mult(x, y)) = z[j_1]$ ) then
  let  $v : D = xor(r_2[j_1], m)$  in  $\overline{c_{Eret}}\langle beta, v \rangle$ 
else
  new  $r_1 : D$ ; let  $v : D = xor(r_1, m)$  in  $\overline{c_{Eret}}\langle beta, v \rangle$ 
|
 $!^{i_H \leq n_H} c_H(z : G)$ ;
find[unique]  $j_2 \leq n_H$  suchthat  $defined(z[j_2], r_2[j_2]) \wedge (z = z[j_2])$  then
   $\overline{c_H}\langle r_2[j_2] \rangle$ 
 $\oplus$  suchthat  $defined(x, y, b, r_1) \wedge (z = exp(g, mult(x, y)))$  then
   $\overline{c_H}\langle r_1 \rangle$ 
else
  new  $r_2 : D$ ;  $\overline{c_H}\langle r_2 \rangle$ )

```

Proof of the example: CDH

```

start(); new  $x : Z$ ; let  $alpha = exp'(g, x)$  in  $\overline{c_{PK}}$  $\langle alpha \rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ; new  $b : bool$ ; let  $m = choose(b, m_0, m_1)$  in
  new  $y : Z$ ; let  $beta = exp'(g, y)$  in
  new  $r_1 : D$ ; let  $v : D = xor(r_1, m)$  in  $\overline{c_{Eret}}$  $\langle beta, v \rangle$ 
|
   $!^{i_H \leq n_H} c_H(z : G)$ ;
  find[unique]  $j_2 \leq n_H$  suchthat  $defined(z[j_2], r_2[j_2]) \wedge (z = z[j_2])$  then
     $\overline{c_H}\langle r_2[j_2] \rangle$ 
  else
    new  $r_2 : D$ ;  $\overline{c_H}\langle r_2 \rangle$ 
)

```

Proof of the example: random group element

```

start(); new alpha : G;  $\overline{c_{PK}}$  $\langle$ alpha $\rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ; new b : bool; let m = choose(b, m_0, m_1) in
  new beta : G;
  new r_1 : D; let v : D = xor(r_1, m) in  $\overline{c_{Eret}}$  $\langle$ beta, v $\rangle$ 
  |
  ! $i_H \leq n_H$   $c_H(z : G)$ ;
  find[unique] j_2 ≤ n_H suchthat defined(z[j_2], r_2[j_2]) ∧ (z = z[j_2]) then
     $\overline{c_H}$  $\langle$ r_2[j_2] $\rangle$ 
  else
    new r_2 : D;  $\overline{c_H}$  $\langle$ r_2 $\rangle$ 
)

```

Proof of the example: xor is a one-time pad

```

start(); new alpha : G;  $\overline{c_{PK}}$  $\langle$ alpha $\rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ; new b : bool; let m = choose(b, m_0, m_1) in
  new beta : G;
  new r_1 : D;  $\overline{c_{Eret}}$  $\langle$ beta, r_1 $\rangle$ 
  |
  ! $i_H \leq n_H$   $c_H(z : G)$ ;
  find[unique] j_2 ≤ n_H suchthat defined(z[j_2], r_2[j_2]) ∧ (z = z[j_2]) then
     $\overline{c_H}$  $\langle$ r_2[j_2] $\rangle$ 
  else
    new r_2 : D;  $\overline{c_H}$  $\langle$ r_2 $\rangle$ 
)

```

Proof of the example: remove useless assignments

```

start(); new alpha : G;  $\overline{c_{PK}}$  $\langle$ alpha $\rangle$ ;
(
   $c_E(m_0 : D, m_1 : D)$ ; new b : bool;
  new beta : G;
  new r1 : D;  $\overline{c_{Eret}}$  $\langle$ beta, r1 $\rangle$ 
  |
  !iH ≤ nH  $c_H(z : G)$ ;
  find[unique] j2 ≤ nH suchthat defined(z[j2], r2[j2]) ∧ (z = z[j2]) then
     $\overline{c_H}$  $\langle$ r2[j2] $\rangle$ 
  else
    new r2 : D;  $\overline{c_H}$  $\langle$ r2 $\rangle$ 
)

```

b is not used at all, it is secret.

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
- signed Diffie-Hellman

- Full domain hash signature (with D. Pointcheval)
- Encryption schemes of Bellare-Rogaway'93 (with D. Pointcheval)
- El Gamal, hashed El Gamal (A. Chaudhuri)

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 PKINIT
(with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- In progress: EKE, with D. Pointcheval.
- Starts being **used by others**:
 - Verification of F# implementations, including TLS, by Microsoft Research and the MSR-INRIA lab.
 - SSH (see next talk)

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 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?