Automatic, computational proof of EKE using CryptoVerif
(Work in progress)

Bruno Blanchet
blanchet@di.ens.fr

Joint work with David Pointcheval

CNRS, École Normale Supérieure, INRIA, Paris

April 2010

Motivation

- EKE (Encrypted Key Exchange):
  - A password-based key exchange protocol.
  - A non-trivial protocol.
  - It took some time before getting a computational proof of this protocol.
- Our goal:
  - Mechanize, and automate as far as possible, its proof using the automatic computational protocol verifier CryptoVerif.
  - This is an opportunity for several interesting extensions of CryptoVerif.

This work is still in progress.

EKE

We consider the variant of EKE of [Bresson, Chevassut, Pointcheval, CCS'03].

$$
\begin{array}{rcl}
\text{Client } U & & \text{Server } S \\
\hline
\multicolumn{3}{c}{\text{shared } pw} \\
x \stackrel{R}{\leftarrow} [1, q-1] & & \\
X \leftarrow g^x & \xrightarrow{U,X} & y \stackrel{R}{\leftarrow} [1, q-1] \\
& & Y \leftarrow g^y \\
Y \leftarrow \mathcal{D}_{pw}(Y^*) & \xleftarrow{S,Y^*} & Y^* \leftarrow \mathcal{E}_{pw}(Y) \\
K_U \leftarrow Y^x & & \\
Auth \leftarrow \mathcal{H}_1(U||S||X||Y||K_U) & & \\
sk_U \leftarrow \mathcal{H}_0(U||S||X||Y||K_U) & \xrightarrow{Auth} & K_s \leftarrow X^y \\
& & \text{if } Auth = \mathcal{H}_1(U||S||X||Y||K_S) \text{ then} \\
& & sk_S \leftarrow \mathcal{H}_0(U||S||X||Y||K_S)
\end{array}
$$

EKE

- The proof relies on the Computational Diffie-Hellman assumption and on the Ideal Cipher Model.
  - $\Rightarrow$ Model these assumptions in CryptoVerif.
- The proof uses Shoup's lemma:
  - Insert an event and later prove that the probability of this event is negligible.
  - $\Rightarrow$ Implement this reasoning technique in CryptoVerif.
- The probability of success of an attack must be precisely evaluated as a function of the size of the password space.
  - $\Rightarrow$ Optimize the computation of probabilities in CryptoVerif.

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

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

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

$$
\begin{aligned}
&!^{i \leq N}\ \textbf{new}\ a : Z; \textbf{new}\ b : Z; (OA() := exp(g, a), OB() := exp(g, b),\\
&\qquad !^{i' \leq N'} OCDH(z : G) := z = exp(g, mult(a, b)))\\
&\approx\\
&!^{i \leq N}\ \textbf{new}\ a : Z; \textbf{new}\ b : Z; (OA() := exp(g, a), OB() := exp(g, b),\\
&\qquad !^{i' \leq N'} OCDH(z : G) := false)
\end{aligned}
$$

Application: semantic security of hashed El Gamal in the random oracle model (A. Chaudhuri).

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

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

Computational Diffie-Hellman assumption in CryptoVerif

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

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

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

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

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

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

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

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

$m = exp(g, mult(b[j], a))$

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

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

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

Other declarations for Diffie-Hellman (1)

$g : G$	generator of G
$exp(G, Z) : G$	exponentiation
$mult(Z, Z) : Z$ commutative	product in $\mathbb{Z}_q$
$exp(exp(z, a), b) = exp(z, mult(a, b))$	$(z^a)^b = z^{ab}$

$(g^a)^b = g^{ab}$ and $(g^b)^a = g^{ba}$, equal by commutativity of *mult*

$(exp(g, x) = exp(g, y)) = (x = y)$
$(exp'(g, x) = exp'(g, y)) = (x = y)$

Injectivity

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

Collision between products

Other declarations for Diffie-Hellman (2)

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

This equivalence is very general, apply it only manually.

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

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

$$!^{i \leq N} \textbf{new } x : Z; OX() := exp(g, x)$$
$$\approx_0 !^{i \leq N} \textbf{new } X : G; OX() := X$$

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 $m = exp(g, mult(b, a))$ 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.

The Ideal Cipher Model

- For all keys, encryption and decryption are two inverse random permutations, independent of the key.
  - Some similarity with SPRP ciphers but, for the ideal cipher model, the key need not be random and secret.
- In CryptoVerif, we replace encryption and decryption with lookups in the previous computations of encryption/decryption:
  - If we find a matching previous encryption/decryption, we return the previous result.
  - Otherwise, we return a fresh random number.
  - We eliminate collisions between these random numbers to obtain permutations.
- No extension of CryptoVerif is needed to represent the Ideal Cipher Model.

Shoup's lemma

Game 0

$\updownarrow$ probability p

Game n

$\updownarrow$ Pr[event e in game $n+1$]

Game $n+1$ event e

$\updownarrow$ probability p'

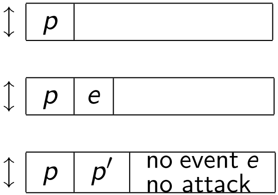
Game n' event e never executed

no attack

$$\begin{aligned}
&\Pr[\text{attack in game } 0] \\
&\quad \le \Pr[\text{dist. } 0/n] + \Pr[\text{dist. } n/n+1] + \Pr[\text{dist. } n+1/n'] \\
&\quad \le \Pr[\text{dist. } 0/n] + \Pr[\text{event } e \text{ in game } n+1] + \Pr[\text{dist. } n+1/n'] \\
&\quad \le \Pr[\text{dist. } 0/n] + \Pr[\text{dist. } n+1/n'] + \Pr[\text{dist. } n+1/n'] \\
&\quad \le p + 2p'
\end{aligned}$$

Improved version with sets of traces

Game 0



Game n

Game $n+1$ event e

Game n' event e never executed
no attack

$$
\begin{aligned}
&\mathrm{Tr}(\text{attack in game } 0)\\
&\quad \subseteq \mathrm{Tr}(\text{dist. } 0/n) \cup \mathrm{Tr}(\text{dist. } n/n+1) + \mathrm{Tr}(\text{dist. } n+1/n')\\
&\quad \subseteq \mathrm{Tr}(\text{dist. } 0/n) \cup \mathrm{Tr}(\text{event } e \text{ in game } n+1) \cup \mathrm{Tr}(\text{dist. } n+1/n')\\
&\quad \subseteq \mathrm{Tr}(\text{dist. } 0/n) \cup \mathrm{Tr}(\text{dist. } n+1/n') \cup \mathrm{Tr}(\text{dist. } n+1/n')
\end{aligned}
$$

So $\Pr[\text{attack in game } 0] \leq p + p'$.

Impact on EKE

- The proof of [Bresson et al, CCS'03] uses the standard Shoup lemma. Probability of an attack:

$$3 \times \frac{q_s}{N} + 8 q_h \times \mathrm{Succ}_G^{\mathrm{cdh}}(t') + \text{collision terms}$$

  - q_s interactions with the parties
  - q_h hash queries
  - dictionary size N

- With the previous remark and the same proof, we obtain instead:

$$\frac{q_s}{N} + q_h \times \mathrm{Succ}_G^{\mathrm{cdh}}(t') + \text{collision terms}$$

- The adversary can test one password per interaction with the parties.

This remark is general: it is not specific to EKE or to CryptoVerif, and can be used in any proof by sequences of games.

CryptoVerif input

CryptoVerif takes as input:

- The assumptions on security primitives: CDH, Ideal Cipher Model, Random Oracle Model.
  - These assumptions are formalized in a library of primitives. The user does not have to redefine them.
- The initial game that represents the protocol EKE:
  - Code for the client
  - Code for the server
  - Code for sessions in which the adversary listens but does not modify messages (passive eavesdroppings)
  - Encryption, decryption, and hash oracles
- The security properties to prove:
  - Secrecy of the keys sk_U and sk_S
  - Authentication of the client to the server
- Manual proof indications (see next slide)

Manual proof indications

- The proof uses two events corresponding to the two cases in which the adversary can guess the password:
  - The adversary impersonates the server by encrypting a Y of its choice under the right password pw, and sending it to the client.
  - The adversary impersonates the client by sending a correct authenticator $Auth$ that it built to the server.
- The manual proof indications consist in manually inserting these two events.

  After that, one runs the automatic proof strategy of CryptoVerif.
- All manual commands are checked by CryptoVerif, so that an incorrect proof cannot be produced.
- CryptoVerif cannot guess where events should be inserted.

Missing step

One argument is still missing to complete the proof:

- The goal is to obtain a final game in which the password is not used at all.
- The encryptions/decryptions under the password *pw* are transformed into lookups that compare *pw* to keys used in other encryption/decryption queries.
- The result of some of these encryptions/decryptions becomes useless after some transformations.
  However, CryptoVerif is currently unable to remove the corresponding lookups that compare with *pw*.

A possible solution

- Move the choice of the (random) result of encryption/decryption to the point at which it is used.
    - This point is typically another encryption/decryption query in which we compared with a previous query.
- After simplification, we end up with **find**s that have several branches that execute the same code up to variable names.
- Merge these branches, thus removing the test of the **find**, which included the comparison with pw.
    - This merging is delicate because the code differs by the variable names, and there exist **find**s on these variables.
    - The branches of these **find**s must also be merged simultaneously.

This solution is still to verify and implement.

Final step

Assuming the previous step is implemented:

- We obtain a game in which the only uses of pw are:
  - Comparison between $dec(Y^*, pw)$ and an encryption query $c = enc(p, k)$ of the adversary: $c = Y^* \wedge k = pw$, in the client.
  - Comparison between $Y = dec(Y^*, pw)$ (obtained from $Y^* = enc(Y, pw)$) and a decryption query $p = dec(c, k)$ of the adversary: $p = Y \wedge k = pw$, in the server.
- We eliminate collisions between the password pw and other keys.
- The difference of probability can be evaluated in two ways:
  - $(q_E + q_D)/N$
    - The password is compared with keys k from q_E encryption queries and q_D decryption queries.
    - Dictionary size N.
  - $(N_U + N_S)/N$

Final step

Assuming the previous step is implemented:

- We obtain a game in which the only uses of pw are:
  - Comparison between $dec(Y^*, pw)$ and an encryption query $c = enc(p, k)$ of the adversary: $c = Y^* \wedge k = pw$, in the client.
  - Comparison between $Y = dec(Y^*, pw)$ (obtained from $Y^* = enc(Y, pw)$) and a decryption query $p = dec(c, k)$ of the adversary: $p = Y \wedge k = pw$, in the server.
- We eliminate collisions between the password pw and other keys.
- The difference of probability can be evaluated in two ways:
  - $(q_E + q_D)/N$
  - $(N_U + N_S)/N$
    - In the client, for each Y^*, there is at most one encryption query with $c = Y^*$ so the password is compared with one key for each session of the client.
    - Similar situation for the server.
    - N_U sessions of the client.
    - N_S sessions of the server.
    - Dictionary size N.

Final step

Assuming the previous step is implemented:

- We obtain a game in which the only uses of *pw* are:
  - Comparison between $dec(Y^*, pw)$ and an encryption query $c = enc(p, k)$ of the adversary: $c = Y^* \wedge k = pw$, in the client.
  - Comparison between $Y = dec(Y^*, pw)$ (obtained from $Y^* = enc(Y, pw)$) and a decryption query $p = dec(c, k)$ of the adversary: $p = Y \wedge k = pw$, in the server.
- We eliminate collisions between the password *pw* and other keys.
- The difference of probability can be evaluated in two ways:
  - $(q_E + q_D)/N$
  - $(N_U + N_S)/N$

  The second bound is the best: the adversary can make many encryption/decryption queries without interacting with the protocol.
  - We extended CryptoVerif so that it can find the second bound.
  - We give it the information that the encryption/decryption queries are non-interactive, so that it prefers the second bound.

Conclusion

The case study of EKE is interesting for itself, but it is even more interesting by the extensions it required in CryptoVerif:

- Treatment of the Computational Diffie-Hellman assumption.
- New manual game transformations, in particular for inserting events.
- Optimization of the computation of probabilities for Shoup's lemma.
- Other optimizations of the computation of probabilities in CryptoVerif.

These extensions are of general interest.