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

**Client** $U$  

- $x \leftarrow [1, q - 1]$  
- $X \leftarrow g^x$  
- $Y \leftarrow D_{pw}(Y^*)$  
- $K_U \leftarrow Y^x$  
- $Auth \leftarrow H_1(U \parallel S \parallel X \parallel Y \parallel K_U)$  
- $sk_U \leftarrow H_0(U \parallel S \parallel X \parallel Y \parallel K_U)$  

**Server** $S$  

- $y \leftarrow [1, q - 1]$  
- $Y \leftarrow g^y$  
- $Y^* \leftarrow E_{pw}(Y)$  

$K_S \leftarrow X^y$

if $Auth = H_1(U \parallel S \parallel X \parallel Y \parallel K_S)$ then

$sk_S \leftarrow H_0(U \parallel S \parallel X \parallel Y \parallel K_S)$
The proof relies on the **Computational Diffie-Hellman** assumption and on the **Ideal Cipher Model**.

- ⇒ 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.
- ⇒ 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**.

- ⇒ 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 $a, b \in \mathbb{Z}_q$. 
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

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

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

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

$$\forall i' \leq N' \; \text{OCDH}(z : G) := \text{false}$$
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 \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, \text{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) := \text{false})
$$

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

\[ \begin{align*}
!i & \leq N_a \new a : Z; (OA() := \exp(g, a), Oa() := a), \\
!i & \leq Na \new a : Z; (OCDH_a(m : G, j \leq N_b) := m = \exp(g, \text{mult}(b[j], a))), \\
!i & \leq N_b \new b : Z; (OB() := \exp(g, b), Ob() := b), \\
!i & \leq Nb \new b : Z; (OCDH_b(m : G, j \leq N_a) := m = \exp(g, \text{mult}(a[j], b))), \\
\end{align*} \]

\[ \approx \]

\[ \begin{align*}
!i & \leq N_a \new a : Z; (OA() := \exp(g, a), Oa() := \text{let } ka = \text{mark } \text{in } a, \\
!i & \leq Na \new a : Z; (OCDH_a(m : G, j \leq N_b) := \\
\text{find } u \leq n_b \text{ such that defined } (kb[u], b[u]) \land b[j] = b[u] \text{ then} \\
m = \exp(g, \text{mult}(b[j], a)) \\
\text{else if defined } (ka) \text{ then } m = \exp(g, \text{mult}(b[j], a)) \text{ else false),} \\
!i & \leq N_b \new b : Z; (OB() := \exp(g, b), Ob() := \text{let } kb = \text{mark } \text{in } b, \\
!i & \leq Nb \new b : Z; (OCDH_b(m : G, j \leq N_a) := (\text{symmetric of } OCDH_a)) \end{align*} \]
Computational Diffie-Hellman assumption in CryptoVerif

\[ \forall a \leq Na \quad \textbf{new} \quad a : Z; (OA()) := \exp(g, a), Oa()[3] := a, \]
\[ \forall a \leq naCDH \quad \text{OCDHa}(m : G, j \leq Nb)[\text{required}] := m = \exp(g, \text{mult}(b[j]),) \]
\[ \forall b \leq Nb \quad \textbf{new} \quad b : Z; (OB()) := \exp(g, b), Ob()[3] := b, \]
\[ \forall b \leq nbCDH \quad \text{OCDHb}(m : G, j \leq Na) := m = \exp(g, \text{mult}(a[j], b))) \]
\[ \approx (\#OCDHa + \#OCDHb) \times \max(1, e^2 \#Oa) \times \max(1, e^2 \#Ob) \times \text{pCDH(time + (na + nb + \#OCDHa + \#OCDHb) \times \text{time(exp)}))} \]

\[ \forall a \leq Na \quad \textbf{new} \quad a : Z; (OA()) := \exp'(g, a), Oa() := \text{let} \quad ka = \text{mark in} \quad a, \]
\[ \forall a \leq naCDH \quad \text{OCDHa}(m : G, j \leq Nb) := \]
\[ \text{find} \quad u \leq nb \quad \text{such that} \quad \text{defined}(kb[u], b[u]) \land b[j] = b[u] \quad \text{then} \]
\[ m = \exp(g, \text{mult}(b[j], a)) \]
\[ \text{else if} \quad \text{defined}(ka) \quad \text{then} \quad m = \exp'(g, \text{mult}(b[j], a)) \quad \text{else} \quad \text{false}), \]
\[ \forall b \leq Nb \quad \textbf{new} \quad b : Z; (OB()) := \exp'(g, b), Ob() := \text{let} \quad kb = \text{mark in} \quad b, \]
\[ \forall b \leq nbCDH \quad \text{OCDHb}(m : G, j \leq Na) := (\text{symmetric of OCDHa})) \]
Other declarations for Diffie-Hellman (1)

\[ g : G \]
\[ \exp(G, Z) : G \]
\[ \text{mult}(Z, Z) : Z \text{ commutative} \]
\[ \exp(\exp(z, a), b) = \exp(z, \text{mult}(a, b)) \]
\[
(g^a)^b = g^{ab} \text{ and } (g^b)^a = g^{ba}, \text{ equal by commutativity of mult}
\]
\[ (\exp(g, x) = \exp(g, y)) = (x = y) \]
\[ (\exp'(g, x) = \exp'(g, y)) = (x = y) \]

Injectivity

\[ \text{new } x_1 : Z; \text{new } x_2 : Z; \text{new } x_3 : Z; \text{new } x_4 : Z; \]
\[ \text{mult}(x_1, x_2) = \text{mult}(x_3, x_4) \overset{\approx_{1/|Z|}}{=} \text{false} \]

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} \( x : Z ; (OX() := \exp(g, x) , i' \leq N' \) \( OXm(m : Z) := \exp(g, \text{mult}(x, m)) ) \)

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 \( m = \exp(g, \text{mult}(b, a)) \) typically occurs inside the condition of a \textbf{find}.
  - This \textbf{find} comes from the transformation of a hash function in the Random Oracle Model.

After transformation, we obtain a \textbf{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

\[ \uparrow \text{probability } p \]

Game \( n \)

\[ \uparrow \Pr[\text{event } e \text{ in game } n + 1] \]

Game \( n + 1 \) event \( e \)

\[ \uparrow \text{probability } p' \]

Game \( n' \) event \( e \) never executed

no attack

\[ \Pr[\text{attack in game } 0] \leq \Pr[\text{dist. } 0/n] + \Pr[\text{dist. } n/n + 1] + \Pr[\text{dist. } n + 1/n'] \]
\[ \leq \Pr[\text{dist. } 0/n] + \Pr[\text{event } e \text{ in game } n + 1] + \Pr[\text{dist. } n + 1/n'] \]
\[ \leq \Pr[\text{dist. } 0/n] + \Pr[\text{dist. } n + 1/n'] + \Pr[\text{dist. } n + 1/n'] \]
\[ \leq p + 2p' \]
Improved version with sets of traces

Game 0

\[ \uparrow p \]

Game \( n \)

\[ \uparrow p \quad \downarrow e \]

Game \( n + 1 \) event \( e \)

\[ \uparrow p \quad p' \quad \text{no event \( e \)} \quad \text{no attack} \]

Game \( n' \) event \( e \) never executed

no attack

\[ \text{Tr(attack in game 0)} \]
\[ \subseteq \text{Tr(dist. 0/}n) \cup \text{Tr(dist. }n/n + 1) + \text{Tr(dist. }n + 1/n') \]
\[ \subseteq \text{Tr(dist. 0/}n) \cup \text{Tr(event }e \text{ in game }n + 1) \cup \text{Tr(dist. }n + 1/n') \]
\[ \subseteq \text{Tr(dist. 0/}n) \cup \text{Tr(dist. }n + 1/n') \cup \text{Tr(dist. }n + 1/n') \]

So \( \text{Pr[attack in game 0]} \leq p + p' \).
Impact on EKE

- The proof of [Bresson et al, CCS’03] uses the standard Shoup lemma. Probability of an attack:
  \[
  3 \times \frac{q_s}{N} + 8q_h \times \text{Succ}_{cdh}^G(t') + \text{collision terms}
  \]
  - \(q_s\) interactions with the parties
  - \(q_h\) hash queries
  - dictionary size \(N\)

- With the previous remark and the same proof, we obtain instead:
  \[
  \frac{q_s}{N} + q_h \times \text{Succ}_{cdh}^G(t') + \text{collision terms}
  \]
  - The adversary can test one password per interaction with the parties.

This remark is \textit{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)
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.
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 **finds** 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 **finds** on these variables.
  - The branches of these **finds** 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: \( c = Y^* \land k = pw \), 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: \( p = Y \land k = pw \), in the server.

- We eliminate collisions between the password \( pw \) and other keys.

- The difference of probability can be evaluated in two ways:
  - \( \frac{q_E + q_D}{N} \)
    - The password is compared with keys \( k \) from \( q_E \) encryption queries and \( q_D \) decryption queries.
    - Dictionary size \( N \).
  - \( \frac{N_U + N_S}{N} \)
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: $c = Y^* \land k = pw$, 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: $p = Y \land k = pw$, 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: $c = Y^* \land k = pw$, 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: $p = Y \land k = pw$, 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.
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.