Automatically Verified Mechanized Proof of One-Encryption Key Exchange

Bruno Blanchet
blanchet@di.ens.fr

Joint work with David Pointcheval

École Normale Supérieure, CNRS, INRIA, Paris

December 2010
Motivation

- **EKE (Encrypted Key Exchange):**
  - A password-based key exchange protocol.
  - A non-trivial protocol.
  - It took some time before getting a proper computational proof of this protocol.

- **Our goal:**
  - Mechanize, and automate as far as possible, its proof using the computational protocol verifier *CryptoVerif*.
  - This is an opportunity for several interesting extensions of CryptoVerif.
The goal of CryptoVerif

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 most often 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.

CryptoVerif achieves **mechanized provability** under the realistic computational assumptions.
CryptoVerif

CryptoVerif is a prover:

- sound in the computational model.
- performs automatic or manually guided proofs.
- 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.)

Games are formalized in a *process calculus*. 
Indistinguishability as observational equivalence

Two processes (games) $Q_1$, $Q_2$ are **observationally equivalent** when the adversary has a negligible probability of distinguishing them:

$$Q_1 \approx Q_2$$

The adversary is represented by an acceptable evaluation context $C$ (essentially, a process put in parallel with the considered games).

- 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 properties of 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** $G_0 \approx G_1 \approx \ldots \approx G_m$, 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 $G_0$. 
Encryption

\[ \mathcal{E}_k(\text{cleartext}) = \text{ciphertext} \]
\[ \mathcal{D}_k(\text{ciphertext}) = \text{cleartext} \]

- Informally, one needs the key to recover the cleartext from the ciphertext.
- **Ideal Cipher Model**: for each key, encryption is a random permutation, independent of the key. Decryption is the inverse permutation.
Hash functions

- A hash function maps a bitstring (of any length) to a small, fixed-length bitstring:
  \[ H(m) = h \]

  Examples: MD5, SHA1.

- It is difficult to find two messages \( m_1, m_2 \) with the same hash \( H(m_1) = H(m_2) \) (collision resistance), . . .

- Random Oracle Model: a hash function is a random function. It maps each distinct message to an independent random number. \( H(m) \) always returns the same result for the same \( m \).
Diffie-Hellman key exchange

- Consider a multiplicative cyclic group $G$ of order $q$, with generator $g$.
  
  **Message 1.** $A \rightarrow B : g^a$ $a \in [1, q - 1]$ fresh
  
  **Message 2.** $B \rightarrow A : g^b$ $b \in [1, q - 1]$ fresh
  
  $A$ computes $k = (g^b)^a$, $B$ computes $k = (g^a)^b$.
  
  These quantities are equal:
  
  $$(g^a)^b = g^{ab} = (g^b)^a$$

- **Computational Diffie-Hellman assumption**: A probabilistic polynomial-time adversary has a negligible probability of computing $g^{ab}$ from $g, g^a, g^b$, for random $a, b \in [1, q - 1]$. 
We consider OEKE, the variant of EKE of [Bresson, Chevassut, Pointcheval, CCS’03].

<table>
<thead>
<tr>
<th>Client $U$</th>
<th>Server $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow [1, q - 1]$</td>
<td>$y \leftarrow [1, q - 1]$</td>
</tr>
<tr>
<td>$X \leftarrow g^x$</td>
<td>$Y \leftarrow g^y$</td>
</tr>
<tr>
<td>$Y \leftarrow D_{pw}(Y^*)$</td>
<td>$Y^* \leftarrow E_{pw}(Y)$</td>
</tr>
<tr>
<td>$K_U \leftarrow Y^x$</td>
<td>$K_S \leftarrow X^y$</td>
</tr>
<tr>
<td>$Auth \leftarrow H_1(U</td>
<td></td>
</tr>
<tr>
<td>$sk_U \leftarrow H_0(U</td>
<td></td>
</tr>
<tr>
<td>if $Auth = H_1(U</td>
<td></td>
</tr>
</tbody>
</table>
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 $a, b \in [1, q - 1]$. 
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 $a, b \in [1, q - 1]$.

In CryptoVerif, this can be written

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

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

$$\approx$$

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

$$\forall i' \leq n' \text{ OCDH}(z : G) := false$$
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 $a, b \in [1, q - 1]$.

In CryptoVerif, this can be written

\[
\begin{align*}
!^{i \leq n} & \textbf{new} \ a : Z; \textbf{new} \ b : Z; (OA() := \exp(g, a), OB() := \exp(g, b), \\
&!^{i' \leq n'} \ OCDH(z : G) := z = \exp(g, \text{mult}(a, b))) \\
\approx \\
!^{i \leq n} & \textbf{new} \ a : Z; \textbf{new} \ b : Z; (OA() := \exp(g, a), OB() := \exp(g, b), \\
&!^{i' \leq n'} \ OCDH(z : G) := false)
\end{align*}
\]

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

\[ \text{!}^{i_a \leq n_a} \textbf{new} \ a : Z; (OA() := \exp(g, a), Oa() := a,} \]
\[ \text{!}^{i_a \text{CDH} \leq n_a \text{CDH}} \ OCDHa(m : G, j \leq nb) := m = \exp(g, \text{mult}(b[j], a))),} \]
\[ \text{!}^{i_b \leq n_b} \textbf{new} \ b : Z; (OB() := \exp(g, b), Ob() := b,} \]
\[ \text{!}^{i_b \text{CDH} \leq n_b \text{CDH}} \ OCDHb(m : G, j \leq na) := m = \exp(g, \text{mult}(a[j], b))) \]
\[ \approx \]
\[ \text{if Ob[j] or Oa has been called then} \]
\[ m = \exp(g, \text{mult}(b[j], a)) \]
\[ \text{else false),} \]
\[ \text{!}^{i_b \leq n_b} \textbf{new} \ b : Z; (OB() := \exp(g, b), Ob() := b,} \]
\[ \text{!}^{i_b \text{CDH} \leq n_b \text{CDH}} \ OCDHb(m : G, j \leq na) := (\text{symmetric of OCDHa}) \]
Computational Diffie-Hellman assumption in CryptoVerif

\[ \text{new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := a, \text{\small \text{\} \text{new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := a,} \]

\[ \text{\small \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := a, \text{\text{\{iaCDH} \leq \text{naCDH}} \text{OCDHa}(m : G, j \leq nb) := m = \text{exp}(g, \text{mult}(b[j], a)))}, \]

\[ \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := b, \text{\small \text{\{ibCDH} \leq \text{nbCDH}} \text{OCDHb}(m : G, j \leq na) := m = \text{exp}(g, \text{mult}(a[j], b))) \]

\[ \approx \]

\[ \text{\small \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a, \text{\small \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a,} \]

\[ \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a, \text{\text{\{iaCDH} \leq \text{naCDH}} \text{OCDHa}(m : G, j \leq nb) :=} \]

\[ \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a, \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a,} \]

\[ \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a, \text{\text{\{iaCDH} \leq \text{naCDH}} \text{OCDHa}(m : G, j \leq nb) :=} \]

\[ \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a, \text{\small \text{\} \text{\} \text{\} \text{\} new } a : \mathbb{Z}; \text{(OA) := exp}(g, a), \text{Oa}() := \text{let } ka = \text{mark } \text{in } a,} \]

\[ \text{\text{\{iaCDH} \leq \text{naCDH}} \text{OCDHa}(m : G, j \leq nb) := \text{\small \begin{array}{l} \text{find } u \leq nb \text{ such that defined}(kb[u], b[u]) \land b[j] = b[u] \text{ then} \smallstrut \cr m = \text{exp}(g, \text{mult}(b[j], a)) \smallstrut \cr \text{else if defined}(ka) \text{ then } m = \text{exp}(g, \text{mult}(b[j], a)) \text{ else } false, \smallstrut \cr \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := \text{let } kb = \text{mark } \text{in } b, \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := \text{let } kb = \text{mark } \text{in } b,} \]

\[ \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := \text{let } kb = \text{mark } \text{in } b, \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := \text{let } kb = \text{mark } \text{in } b,} \]

\[ \text{\small \text{\} \text{\} new } b : \mathbb{Z}; \text{(OB) := exp}(g, b), \text{Ob}() := \text{let } kb = \text{mark } \text{in } b, \text{\text{\{ibCDH} \leq \text{nbCDH}} \text{OCDHb}(m : G, j \leq na) := \text{\small (symmetric of OCDHa))} \]

Bruno Blanchet and David Pointcheval

OEKE in CryptoVerif

December 2010 16 / 29
Computational Diffie-Hellman assumption in CryptoVerif

\[ \text{new } a : Z; (OA()) := \exp(g, a), Oa()[3] := a, \]
\[ \text{new } b : Z; (OB()) := \exp(g, b), Ob()[3] := b, \]
\[ \text{find } u \leq nb \text{ suchthat defined}(kb[u], b[u]) \land b[j] = b[u] \text{ then } m = \exp(g, \text{mult}(b[j], a)) \]
\[ \text{else if defined}(ka) \text{ then } m = \exp'(g, \text{mult}(b[j], a)) \text{ else false}, \]
\[ \text{new } a : Z; (OA()) := \exp'(g, a), Oa() := \text{let } ka = \text{mark} \text{ in } a, \]
\[ \text{new } b : Z; (OB()) := \exp'(g, b), Ob() := \text{let } kb = \text{mark} \text{ in } b, \]
\[ (\text{symmetric of } OCDHa) \]
Other declarations for Diffie-Hellman (1)

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

\[ (\exp(g, x) = \exp(g, y)) = (x = y) \]
\[ (\exp'(g, x) = \exp'(g, y)) = (x = y) \]

**Injectivity**

\[ (\text{mult}(x, y) = \text{mult}(x, y')) = (y = y') \]

new \( x_1 : Z; \) new \( x_2 : Z; \) new \( x_3 : Z; \) new \( x_4 : Z; \)
\[ \text{mult}(x_1, x_2) = \text{mult}(x_3, x_4) \not\approx_{1/|Z|} \text{false} \]

**Collision between products**
Other declarations for Diffie-Hellman (2)

\[ \forall i \leq n \text{new } X : G ; OX() := X \]
\[ \approx_0 [\text{manual}] \forall i \leq n \text{new } x : Z ; OX() := \exp(g, x) \]

This equivalence is very general, apply it only manually.

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

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

\[ \forall i \leq n \text{new } x : Z ; OX() := \exp(g, x) \]
\[ \approx_0 \forall i \leq n \text{new } X : G ; OX() := X \]

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 $m = \text{exp}(g, \text{mult}(b, a))$ 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.
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.
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 slides)
Shoup’s lemma

Game 0

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

Game \(n\)

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

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

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

Game \(n'\) event \(e\) never executed
no attack

\[\text{Pr[attack in game 0]} \leq \text{Pr[dist. } 0/n] + \text{Pr[dist. } n/n+1] + \text{Pr[dist. } n+1/n']\]

\[\leq \text{Pr[dist. } 0/n] + \text{Pr[event } e \text{ in game } n+1] + \text{Pr[dist. } n+1/n']\]

\[\leq \text{Pr[dist. } 0/n] + \text{Pr[dist. } n+1/n'] + \text{Pr[dist. } n+1/n']\]

\[\leq p + 2p'\]
Applying Shoup’s lemma

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.

We use manual proof indications for inserting these two events.

- Before inserting events, we first make the program point appear, at which the event will be inserted.
  In particular, we apply the random oracle assumption on $H_1$ and the ideal cipher assumption.
- All manual commands are checked by CryptoVerif, so that an incorrect proof cannot be produced.
Automatic steps

After inserting events, one runs the automatic proof strategy of CryptoVerif.

- Apply all possible cryptographic transformations (coming from equivalences).
- After each such transformation, the game is simplified.
- When the transformations fail, they advise syntactic transformations that could make them succeed:
  - these transformations are executed,
  - the cryptographic transformation is then retried.

For OEKE, CryptoVerif basically

1. applies the random oracle assumption on $\mathcal{H}_0$,
2. renames some variables and simplifies some terms, and
3. applies the CDH assumption.
Reorganizing random number generations

- 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. We perform some manually guided transformations to remove the corresponding lookups that compare with $pw$. 
Reorganizing random number generations (continued)

Delay 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.
- This transformation can in fact be expressed as an equivalence. ⇒ No need to modify CryptoVerif itself to implement it.
Reorganizing random number generations (continued)

1. **Delay** 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.

2. After simplification, we end up with **finds** that have **several branches that execute the same code** up to variable names.
   - The result of an encryption/decryption query is either:
     - the standard random choice that previously existed, $X$;
     - the delayed random choice that comes from transformation 1, $Y$. 
Delay 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.
- The result of an encryption/decryption query is either:
  - the standard random choice that previously existed, \( X \);
  - the delayed random choice that comes from transformation 1, \( Y \).

Merge the two arrays \( X \) and \( Y \) into the array \( X \).
- If a find has two branches, one looking up in \( X \) and one in \( Y \), then these two branches are replaced with one branch looking up in \( X \).
Reorganizing random number generations (continued)

1. **Delay** 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.

2. After simplification, we end up with **finds** that have **several branches that execute the same code** up to variable names.
   - The result of an encryption/decryption query is either:
     - the standard random choice that previously existed, $X$;
     - the delayed random choice that comes from transformation 1, $Y$.

3. **Merge the two arrays** $X$ and $Y$ into the array $X$.
   - If a **find** has two branches, one looking up in $X$ and one in $Y$, then these two branches are replaced with one branch looking up in $X$.

4. **Merge the find branches**, thus removing the test of the **find**, which included the comparison with $pw$. 

Bruno Blanchet and David Pointcheval
Final computation of probabilities

- We obtain a game in which the **only uses of pw** are:
  - Comparison between \( \text{dec}(Y^*, pw) \) and an encryption query \( c = \text{enc}(p, k) \) of the adversary: \( c = Y^* \land k = pw \), in the client.
  - Comparison between \( Y = \text{dec}(Y^*, pw) \) (obtained from \( Y^* = \text{enc}(Y, pw) \)) and a decryption query \( p = \text{dec}(c, k) \) 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**:
  - \( (q_E + q_D)/|\text{passwd}| \)
    - The password is compared with keys \( k \) from \( q_E \) encryption queries and \( q_D \) decryption queries.
    - Dictionary size \( |\text{passwd}| \).
  - \( (N_U + N_S)/|\text{passwd}| \)
Final computation of probabilities

- We obtain a game in which the **only uses of pw** are:
  - Comparison between $\text{dec}(Y^*, pw)$ and an encryption query $c = \text{enc}(p, k)$ of the adversary: $c = Y^* \land k = pw$, in the client.
  - Comparison between $Y = \text{dec}(Y^*, pw)$ (obtained from $Y^* = \text{enc}(Y, pw)$) and a decryption query $p = \text{dec}(c, k)$ 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**:
  - $\frac{(q_E + q_D)}{|\text{passwd}|}$
  - $\frac{(N_U + N_S)}{|\text{passwd}|}$

  - 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 $|\text{passwd}|$. 
Final computation of probabilities

- We obtain a game in which the only uses of $pw$ are:
  - Comparison between $\text{dec}(Y^*, pw)$ and an encryption query $c = \text{enc}(p, k)$ of the adversary: $c = Y^* \land k = pw$, in the client.
  - Comparison between $Y = \text{dec}(Y^*, pw)$ (obtained from $Y^* = \text{enc}(Y, pw)$) and a decryption query $p = \text{dec}(c, k)$ 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:
  - $\frac{(q_E + q_D)}{|\text{passwd}|}$
  - $\frac{(N_U + N_S)}{|\text{passwd}|}$

  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.
Obtained result

By summing up all differences of probabilities, the probability of distinguishing the initial game from the final one is

\[ p = 5 \times \frac{N_U + N_S}{|\text{passwd}|} + 8(qH_0 + qH_1)\text{Succ}_{CDH}^G(t') + \text{negl}() \]

where

- \( t' = t + (2qH_0 + 2qH_1 + 2N_U + q_D + 2N_P + N_S)\tau_{\text{exp}} \),
- \( qH_0 \) queries to \( \mathcal{H}_0 \), \( qH_1 \) queries to \( \mathcal{H}_1 \),
- the terms in \( \text{negl}() \) come from elimination of collisions between hashes and between group elements.

So we obtain the following security results:

- **OEKE preserves the secrecy of** \( sk_U \) and \( sk_S \) **up to probability** \( 2p \);
- **OEKE satisfies authentication of the client to the server** **up to probability** \( p \).
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
  - for inserting events,
  - for merging cases.
- Optimizations of the **computation of probabilities** in CryptoVerif.

These extensions are of general interest.