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

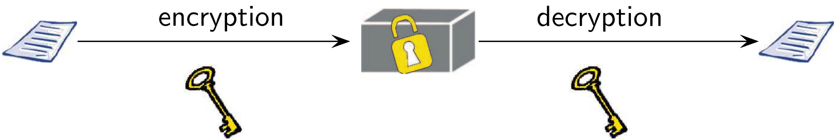
We transform a game G_0 into an observationally equivalent one using:

- 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



$$\mathcal{E}_k(\mathit{cleartext}) = \mathit{ciphertext}$$
$$\mathcal{D}_k(\mathit{ciphertext}) = \mathit{cleartext}$$

- 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:
$$\mathcal{H}(m) = h$$
Examples: MD5, SHA1.
- It is difficult to find two messages m_1, m_2 with the same hash $\mathcal{H}(m_1) = \mathcal{H}(m_2)$ (collision resistance), ...
- Random Oracle Model: a hash function is a random function. It maps each distinct message to an independent random number.

  $\mathcal{H}(m)$ 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.

$$\begin{array}{llll} \text{Message 1.} & A \rightarrow B: & g^a & a \in [1, q-1] \text{ fresh} \\ \text{Message 2.} & B \rightarrow A: & g^b & b \in [1, q-1] \text{ fresh} \end{array}$$

  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 $a, b \in [1, q-1]$.

OEKE

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

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

OEKE

- 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 $a, b \in [1, q-1]$.

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

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

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

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

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

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

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

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

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

$\approx$

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

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

if $Ob[j]$ or Oa has been called **then**

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

else *false*),

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

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

Computational Diffie-Hellman assumption in CryptoVerif

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

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

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

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

$\approx$

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

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

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

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

$\quad\quad$ **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** $b,$

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

Computational Diffie-Hellman assumption in CryptoVerif

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

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

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

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

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

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

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

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

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

$\quad\quad$ **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** $b,$

$\quad !^{ibCDH \leq nbCDH} OCDHb(m : G, j \leq na) :=$ (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

$(mult(x, y) = mult(x, y')) = (y = y')$

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.

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

$\updownarrow$ 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

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

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 $\mathcal{H}_1$ 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)

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.
    - This transformation can in fact be expressed as an equivalence.
      $\Rightarrow$ 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 **find**s 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.

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 **find**s 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.

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 **find**s 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.

Final computation of probabilities

- 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:
  - $(q_E + q_D)/|passwd|$
    - The password is compared with keys k from q_E encryption queries and q_D decryption queries.
    - Dictionary size $|passwd|$.
  - $(N_U + N_S)/|passwd|$

Final computation of probabilities

- 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:
  - $(q_E + q_D)/|passwd|$
  - $(N_U + N_S)/|passwd|$
    - 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 $|passwd|$.

Final computation of probabilities

- 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:
  - $(q_E + q_D)/|passwd|$
  - $(N_U + N_S)/|passwd|$

  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

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

where

- $t' = t + (2qH_0 + 2qH_1 + 2N_U + q_D + 2N_P + N_S)\tau_{\mathrm{exp}}$,
- qH_0 queries to $\mathcal{H}_0$, qH_1 queries to $\mathcal{H}_1$,
- the terms in $negl()$ 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.