Automatically Verified Mechanized Proof of One-Encryption Key Exchange

Bruno Blanchet
blanchet@di.ens.fr

INRIA, École Normale Supérieure, CNRS, Paris

January 2012

Motivation

- OEKE (One-Encryption Key Exchange) [Bresson, Chevassut, Pointcheval, CCS'03]:
    - Variant of 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.

Proofs by sequences of games

Proofs in the computational model are typically proofs by sequences of games [Shoup, Bellare&Rogaway]:

- 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. The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property is obvious from the form of the game.
  (The advantage of the adversary is 0 for this game.)

Game 0		Game 1				Game n
Protocol to prove	$\longleftrightarrow$ p_1 negligible		$\longleftrightarrow$ p_2 negligible	$\cdots$	$\longleftrightarrow$ p_n negligible	Property obvious

CryptoVerif background: Indistinguishability

- The game G interacting with an adversary (evaluation context) C is denoted $C[G]$.
- $C[G]$ may execute events, collected in a sequence $\mathcal{E}$.
- A distinguisher D takes as input $\mathcal{E}$ and returns **true** or **false**.
  - Example: $D_e(\mathcal{E}) = \textbf{true}$ if and only if $e \in \mathcal{E}$. D_e is abbreviated e.
- $\Pr[C[G] : D]$ is the probability that $C[G]$ executes $\mathcal{E}$ such that $D(\mathcal{E}) = \textbf{true}$.

Definition (Indistinguishability)

We write $G \approx_p^V G'$ when, for all evaluation contexts C acceptable for G and G' with public variables V and all distinguishers D,

$$|\Pr[C[G] : D] - \Pr[C[G'] : D]| \leq p(C, D).$$

Properties of indistinguishability

Lemma

1. *Reflexivity:* $G \approx_0^V G$.
2. *Symmetry:* $\approx_p^V$ *is symmetric.*
3. *Transitivity: if* $G \approx_p^V G'$ *and* $G' \approx_{p'}^V G''$, *then* $G \approx_{p+p'}^V G''$.
4. *Application of context: if* $G \approx_p^V G'$ *and* C *is an evaluation context acceptable for* G *and* G' *with public variables* V, *then* $C[G] \approx_{p'}^{V'} C[G']$, *where* $p'(C', D) = p(C'[C[\,]], D)$ *and* $V' \subseteq V \cup \mathrm{var}(C)$.

OEKE

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

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 $a, b \in \mathbb{Z}_q$.

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

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

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

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

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 OEKE 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.

Extending the formalization of CDH 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**

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

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 $OCDHa$))

Extending the formalization of CDH in CryptoVerif

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

Extending the formalization of CDH in CryptoVerif

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

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

$!^{ib \leq Nb}$ **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** $x4 : Z;$

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

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

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.

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

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.
  - extend the language of equivalences used for specifying assumptions on primitives.
- 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**.

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

Goal: bound $\Pr[C[G_o] : e_0]$.

G_0	
$\updownarrow$ probability p	
G_n	
$\updownarrow$ $\Pr[C[G_{n+1}] : e]$	
G_{n+1}	event e
$\updownarrow$ probability p'	
$G_{n'}$	events e_0 and e never executed

$$\begin{aligned}\Pr[C[G_0] : e_0] &\leq p + \Pr[C[G_{n+1}] : e] + p' \\ &\leq p + p' + p' \\ &\leq p + 2p'\end{aligned}$$

Improved version of Shoup's lemma

Goal: bound $\Pr[C[G_0] : e_0]$.

G_0

$\updownarrow$ probability p

G_n

$\updownarrow$ differ only when e is executed

G_{n+1} event e

$\updownarrow$ probability p'

$G_{n'}$ events e_0 and e never executed

$$\begin{aligned}
\Pr[C[G_0] : e_0] &\leq p + \Pr[C[G_n] : e_0] \\
&\leq p + \Pr[C[G_{n+1}] : e_0 \vee e] \\
&\leq p + p' + \Pr[C[G_{n'}] : e_0 \vee e] \\
&\leq p + p'
\end{aligned}$$

Improved Shoup's lemma

Lemma

Let C be a context acceptable for G and G' with public variables V.

1. *Improved Shoup's lemma:*
   If G' differs from G only when G' executes event e, then $\Pr[C[G] : D] \leq \Pr[C[G'] : D \vee e]$.
2. *If $G \approx_p^V G'$, then* $\Pr[C[G] : D] \leq p(C, D) + \Pr[C[G'] : D]$.
3. $\Pr[C[G] : D \vee D'] \leq \Pr[C[G] : D] + \Pr[C[G] : D']$.

Definition of secrecy

Definition (Secrecy)

Let x be a one-dimensional array.

Let R_x be a process that

- chooses a bit b;
- provides test queries that, on input u, return $x[u]$ when $b = 1$ and a random value $y[u]$ when $b = 0$;
- expects a value b' from the adversary and executes event S when $b' = b$.

Let C be a context acceptable for $G \mid R_x$ without public variables that does not contain S.

$$\mathrm{Adv}_G^{\mathrm{secrecy}(x)}(C) = 2\Pr[C[G \mid R_x] : \mathsf{S}] - 1$$

Definition of secrecy

Definition (Secrecy)

Let x be a one-dimensional array.

Let R_x be a process that

- chooses a bit b;
- provides test queries that, on input u, return $x[u]$ when $b = 1$ and a random value $y[u]$ when $b = 0$;
- expects a value b' from the adversary and executes event S when $b' = b$.

Let C be a context acceptable for $G \mid R_x$ without public variables that does not contain S.

$$\mathrm{Adv}_G^{\mathrm{secrecy}(x)}(C) = 2\Pr[C[G \mid R_x] : \mathsf{S}] - 1$$

Proof of secrecy

Goal: secrecy of x in G_0

$G_0 \mid R_x$

$\updownarrow$ probability p

$G_n \mid R_x$ secrecy proved: $\Pr[C[G_n \mid R_x] : \mathsf{S}] = \frac{1}{2}$

$$\begin{aligned}\mathrm{Adv}_{G_0}^{\mathrm{secrecy}(x)}(C) &= 2\Pr[C[G_0 \mid R_x] : \mathsf{S}] - 1 \\ &\leq 2(p + \Pr[C[G_n \mid R_x] : \mathsf{S}]) - 1 \\ &\leq 2p\end{aligned}$$

Proof of secrecy with Shoup's lemma

$G_0 \mid R_x$	goal: secrecy of x in G_0
$\updownarrow$ probability p	
$G_n \mid R_x$	
$\updownarrow$ differ only when e is executed	
$G_{n+1} \mid R_x$	event e
$\updownarrow$ probability p'	
$G_{n'} \mid R_x$	secrecy proved: $\Pr[C[G_{n'} \mid R_x] : \mathsf{S}] = \frac{1}{2}$
$\updownarrow$ probability p''	
$G_{n''} \mid R_x$	event e never executed

$$\begin{aligned}
\mathrm{Adv}^{\mathrm{secrecy}(x)}_{G_0}(C) &\leq 2(p + \Pr[C[G_n \mid R_x] : \mathsf{S}]) - 1 \\
&\leq 2(p + \Pr[C[G_{n+1} \mid R_x] : \mathsf{S} \vee e]) - 1 \\
&\leq 2(p + p' + \Pr[C[G_{n'} \mid R_x] : \mathsf{S} \vee e]) - 1 \\
&\leq 2(p + p' + \Pr[C[G_{n'} \mid R_x] : e]) \leq 2(p + p' + p'')
\end{aligned}$$

Improved proof of secrecy with Shoup's lemma

$G_0 \mid R_x$ — goal: secrecy of x in G_0

$\updownarrow$ probability p

$G_n \mid R_x$

$\updownarrow$ differ only when e is executed

$G_{n+1} \mid R_x$ — event e

$\updownarrow$ probability p'

$G_{n'} \mid R_x$ — secrecy proved: $\Pr[C[G_{n'} \mid R_x] : \mathsf{S}] = \frac{1}{2}$

event e is independent of S

$\updownarrow$ probability p''

$G_{n''} \mid R_x$ — event e never executed

$$\mathrm{Adv}_{G_0}^{\mathrm{secrecy}(x)}(C) \leq 2(p + p' + \Pr[C[G_{n'} \mid R_x] : \mathsf{S} \vee e]) - 1$$

$$\leq 2(p + p' + \frac{1}{2}\Pr[C[G_{n'} \mid R_x] : e]) \leq 2(p + p') + p''$$

Improved proof of secrecy with Shoup's lemma

Lemma

If CryptoVerif proves the secrecy of x in game G, and $e_1, \dots, e_n$ are events introduced by Shoup's lemma in previous steps of the proof, then

$$\Pr[C[G \mid R_x] : \mathsf{S} \vee e_1 \vee \dots \vee e_n] \leq \frac{1}{2} + \frac{1}{2}\Pr[C[G \mid R_x] : e_1 \vee \dots \vee e_n].$$

Events $e_1, \dots, e_n$ are independent of S.

$$\begin{aligned}
&\Pr[C[G] : \mathsf{S} \vee e_1 \vee \dots \vee e_n] \\
&\quad = \Pr[C[G] : \mathsf{S}] + \Pr[C[G] : \neg\mathsf{S} \wedge (e_1 \vee \dots \vee e_n)] \\
&\quad = \frac{1}{2} + \Pr[C[G] : \neg\mathsf{S}] \Pr[C[G] : e_1 \vee \dots \vee e_n] \\
&\quad = \frac{1}{2} + \frac{1}{2}\Pr[C[G] : e_1 \vee \dots \vee e_n]
\end{aligned}$$

Impact on OEKE: Notations

- dictionary size N
- N_U client instances under active attack
- N_S server instances under active attack
- N_P sessions under passive attack
- q_h hash queries

Impact on OEKE: semantic security

- Standard computation of probabilities:

$$\mathrm{Adv}^{\mathrm{ake}}_{G_0}(C) \leq \frac{4N_S + 2N_U}{N} + 8q_h \times \mathrm{Succ}^{\mathrm{cdh}}_{\mathbb{G}}(t') + \text{collision terms}$$

- Improved computation of probabilities:

$$\mathrm{Adv}^{\mathrm{ake}}_{G_0}(C) \leq \frac{N_S + N_U}{N} + q_h \times \mathrm{Succ}^{\mathrm{cdh}}_{G}(t') + \text{collision terms}$$

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

Impact on OEKE: one-way authentication

- Standard computation of probabilities:

$$\mathrm{Adv}_{G_0}^{\mathrm{c-auth}}(C) \leq \frac{2N_S + N_U}{N} + 3q_h \times \mathrm{Succ}_{\mathbb{G}}^{\mathrm{cdh}}(t') + \text{collision terms}$$

- Improved computation of probabilities:

$$\mathrm{Adv}_{G_0}^{\mathrm{c-auth}}(C) \leq \frac{N_S + N_U}{N} + q_h \times \mathrm{Succ}_{G}^{\mathrm{cdh}}(t') + \text{collision terms}$$

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

This remark is general: it is not specific to OEKE 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 OEKE:
  - 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

1. 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 has built to the server.

   First, one uses manual proof indications to manually insert these two events.
   - CryptoVerif cannot guess where events should be inserted.
2. After that, one runs the automatic proof strategy of CryptoVerif.
3. Finally, one uses manual tranformations to eliminate uses of the password.

All manual commands are checked by CryptoVerif, so that an incorrect proof cannot be produced.

Uses of the password after automatic transformations

- Goal: in the final game, 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.
- After the automatic game transformations, the (random) result of some of these encryptions/decryptions is used only in comparisons with previous encryption/decryption queries.
  We remove the corresponding lookups that compare with *pw*, using manual transformations.

Delaying random choices: Y_U (1)

Client U

$\ldots$
$Y_U \leftarrow \mathcal{D}_{pw}(Y_U^*)$
$K_U \leftarrow Y_U^x$
$Auth \leftarrow \mathcal{H}_1(U||S||X||Y_U||K_U)$
$sk_U \leftarrow \mathcal{H}_0(U||S||X||Y_U||K_U)$

Decryption oracle

$(m, kd) \mapsto$ **return** $\mathcal{D}_{kd}(m)$

Delaying random choices: Y_U (2)

Client U

...

find $\mathcal{D}_{pw}(Y_U^*)$ or $\mathcal{E}_{pw}(\cdot) = Y_U^*$ in previous queries **then** ...

else $Y_U \stackrel{R}{\leftarrow} G$; $Auth \stackrel{R}{\leftarrow} H_1$; $sk_U \stackrel{R}{\leftarrow} H_0$

Decryption oracle

$(m, kd) \mapsto$ **find** $\mathcal{D}_{kd}(m)$ or $\mathcal{E}_{kd}(\cdot) = m$ in previous queries **then** ...

else $Y_d \stackrel{R}{\leftarrow} G$; **return** Y_d

$\Rightarrow$ Y_U used only in comparisons with previous queries.

Delaying random choices (3)

- `move array` Y_U: Move the choice of Y_U to the point at which it is used.
  In OEKE, this point is the decryption oracle.
  This oracle can return two randomly chosen values:
  - the one that comes from the delayed choice of Y_U, Y'_U,
  - the one that comes from fresh decryption queries, Y_d.
- After simplification, we have a **find** with several branches that execute the same code up to variable names (Y'_U vs. Y_d).
- Merge these branches, thus removing the test of the **find**, which included the comparison with pw.

Delaying random choices (4)

- `move array` Y_U: Move the choice of Y_U to the point at which it is used.
- After simplification, we have a **find** with several branches that execute the same code up to variable names (Y'_U vs. Y_d).

Client U

$$\begin{array}{l}\textbf{find } \mathcal{D}_{pw}(Y^*_U) \text{ or } \mathcal{E}_{pw}(\cdot) = Y^*_U \text{ in previous queries } \textbf{then } \ldots\\ \textbf{else } Auth \stackrel{R}{\leftarrow} H_1;\ sk_U \stackrel{R}{\leftarrow} H_0\end{array}$$

Decryption oracle

$$\begin{array}{l}(m, kd) \mapsto \textbf{find } \mathcal{D}_{kd}(m) \text{ or } \mathcal{E}_{kd}(\cdot) = m \text{ in previous queries } \textbf{then } \ldots\\ \qquad\qquad \textbf{else find } j \textbf{ suchthat } m = Y^*_U[j] \wedge kd = pw\\ \qquad\qquad\qquad \textbf{then } Y'_U \stackrel{R}{\leftarrow} G; \textbf{return } Y'_U\\ \qquad\qquad\qquad \textbf{else } Y_d \stackrel{R}{\leftarrow} G; \textbf{return } Y_d\end{array}$$

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

Delaying random choices (5)

- `move array` Y_U: Move the choice of Y_U to the point at which it is used.
- After simplification, we have a **find** with several branches that execute the same code up to variable names (Y'_U vs. Y_d).
- Merge these branches, thus removing the test of the **find**, which included the comparison with pw.
  Delicate because the code differs by the variable names (Y'_U vs. Y_d) and there exist **find**s on these variables.
  1. `move binder` $r1$: reorder instructions so that they are in the same order in the branches to merge.
  2. `merge_arrays` Y_d Y'_U: merge the array Y'_U into Y_d.
  3. `merge_branches`: merge the branches of **find** themselves.

Delaying random choices

- `move array`, `merge_arrays`, and `merge_branches` are new game transformations.
- Similar technique for two other random values:
  - Y in the eavesdropped sessions,
  - Y in the server.

Final elimination of collisions with the password

After the previous steps:

- 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 elimination of collisions with the password

After the previous steps:

- 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 client instances under active attack
    - N_S server instances under active attack
    - Dictionary size N.

Final elimination of collisions with the password

After the previous steps:

- 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 OEKE 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 and merging branches of tests.
- 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.