The automatic protocol verifier CryptoVerif

Bruno Blanchet

INRIA Paris-Rocquencourt
Bruno.Blanchet@inria.fr

April 2015

Introduction

Two approaches for the automatic proof of cryptographic protocols in a computational model:

- Indirect approach:

  1) Make a Dolev-Yao proof.
  2) Use a theorem that shows the soundness of the Dolev-Yao approach with respect to the computational model.

  Pioneered by Abadi and Rogaway; pursued by many others.

- Direct approach:

  Design automatic tools for proving protocols in a computational model.

  Approach pioneered by Laud.

Advantages and drawbacks

The indirect approach allows more reuse of previous work, but it has limitations:

- Hypotheses have to be added to make sure that the computational and Dolev-Yao models coincide.
- The allowed cryptographic primitives are often limited, and only ideal, not very practical primitives can be used.
- Using the Dolev-Yao model is actually a (big) detour;
  The computational definitions of primitives fit the computational security properties to prove.
  They do not fit the Dolev-Yao model.

We decided to focus on the direct approach.

CryptoVerif, http://cryptoverif.inria.fr/

CryptoVerif is an automatic prover that:

- generates proofs by sequences of games.
- proves secrecy and correspondence properties.
- provides a generic method for specifying properties of cryptographic primitives which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, Diffie-Hellman key agreements, ...
- works for N sessions (polynomial in the security parameter), with an active adversary.
- gives a bound on the probability of an attack (exact security).

Proofs by sequences of games

CryptoVerif produces proofs by sequences of games, like those of cryptographers [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	$\longleftrightarrow$	Game 1	$\longleftrightarrow$		$\longleftrightarrow$	Game n
Protocol to prove	p_1 negligible		p_2 negligible	$\cdots$	p_n negligible	Property obvious

Input and output of the tool

1. Prepare the input file containing
   - the specification of the protocol to study (initial game),
   - the security assumptions on the cryptographic primitives,
   - the security properties to prove.
2. Run CryptoVerif
3. CryptoVerif outputs
   - the sequence of games that leads to the proof,
   - a succinct explanation of the transformations performed between games,
   - an upper bound of the probability of success of an attack.

Process calculus for games

Games are formalized in a process calculus:

- It is adapted from the pi calculus.
- The semantics is purely probabilistic (no non-determinism).
- The runtime of processes is bounded:
  - bounded number of copies of processes,
  - bounded length of messages on channels.
- Extension to arrays.

Process calculus for games: terms

Terms represent computations on messages (bitstrings).

$M ::=$	terms
$\quad x, y, z, x[M_1, \dots, M_n]$	variable
$\quad f(M_1, \dots, M_n)$	function application

Function symbols f correspond to functions computable by deterministic Turing machines that always terminate.

Process calculus for games: processes

$Q ::=$	input process
0	end
$Q \parallel Q'$	parallel composition
$!^{i \leq N} Q$	replication N times
newChannel $c; Q$	restriction for channels
in$(c, (x_1 : T_1, \dots, x_m : T_m)); P$	input
$P ::=$	output process
yield	end
out$(c, (M_1, \dots, M_m)); Q$	output
event $e(M_1, \dots, M_m); P$	event
new $x : T; P$	random number generation (uniform)
let $x : T = M$ **in** P	assignment
if M **then** P **else** P'	conditional
find $j \leq N$ **suchthat defined**$(x[j], \dots) \wedge M$ **then** P **else** P'	array lookup

Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

Definition (Symmetric encryption scheme SE)

- (Randomized) key generation function $kgen$.
- (Randomized) encryption function $enc(m, k, r')$ takes as input a message m, a key k, and random coins r'.
- Decryption function $dec(c, k)$ such that

$$dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$$

The decryption returns a bitstring or $\perp$:

- $\perp$ when decryption fails,
- the cleartext when decryption succeeds.

The injection $i_\perp$ maps a bitstring to the same bitstring in bitstring $\cup \{\perp\}$.

Example: 2. MAC

Definition (Message Authentication Code scheme MAC)

- (Randomized) key generation function $mkgen$.
- MAC function $mac(m, k)$ takes as input a message m and a key k.
- Verification function $verify(m, k, t)$ such that

$$verify(m, k, mac(m, k)) = true.$$

A MAC is essentially a keyed hash function.

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC.

Example: 3. encrypt-then-MAC

We define an authenticated encryption scheme by the encrypt-then-MAC construction:

$$enc'(m, (k, mk), r'') = e, mac(e, mk) \text{ where } e = enc(m, k, r'').$$

A basic example of protocol using encrypt-then-MAC:

- A and B initially share an encryption key k and a MAC key mk.
- A sends to B a fresh key k' encrypted under authenticated encryption, implemented as encrypt-then-MAC.

$$A \rightarrow B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

k' should remain secret.

Example: initialization

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_0 = {} & \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{let}\ k : key = kgen(r)\ \mathbf{in} \\ & \mathbf{new}\ r' : mkeyseed; \mathbf{let}\ mk : mkey = mkgen(r')\ \mathbf{in}\ \mathbf{out}(c, ()); \\ & (Q_A \parallel Q_B) \end{aligned}$$

Initialization of keys:

1. The process Q_0 waits for a message on channel *start* to start running.
   The adversary triggers this process.
2. Q_0 generates encryption and MAC keys, k and mk respectively, using the key generation algorithms *kgen* and *mkgen*.
3. Q_0 returns control to the adversary by the output $\mathbf{out}(c, ())$.
   Q_A and Q_B represent the actions of A and B (see next slides).

Example: role of A

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_A = {!}^{i \leq n}\, &\mathbf{in}(c_A, ()); \mathbf{new}\ k' : key; \mathbf{new}\ r'' : coins; \\ &\mathbf{let}\ e : bitstring = enc(k2b(k'), k, r'')\ \mathbf{in} \\ &\mathbf{out}(c_A, (e, mac(e, mk))) \end{aligned}$$

Role of A:

1. ${!}^{i \leq n}$ represents n copies, indexed by $i \in [1, n]$
   The protocol can be run n times (polynomial in the security parameter).
2. The process is triggered when a message is sent on c_A by the adversary.
3. The process chooses a fresh key k' and sends the message on channel c_A.

Example: role of B

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_B = {!}^{i' \leq n}\, &\mathbf{in}(c_B, (e' : bitstring, ma : macstring)); \\ &\mathbf{if}\ verify(e', mk, ma)\ \mathbf{then} \\ &\mathbf{let}\ i_\perp(k2b(k'')) = dec(e', k)\ \mathbf{in}\ \mathbf{out}(c_B, ()) \end{aligned}$$

Role of B:

1. n copies, as for Q_A.
2. The process Q_B waits for the message on channel c_B.
3. It verifies the MAC, decrypts, and stores the key in k''.

Example: summary of the initial game

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned}Q_0 = {} & \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{let}\ k : key = kgen(r)\ \mathbf{in} \\ & \mathbf{new}\ r' : mkeyseed; \mathbf{let}\ mk : mkey = mkgen(r')\ \mathbf{in}\ \mathbf{out}(c, ()); \\ & (Q_A \parallel Q_B)\end{aligned}$$

$$\begin{aligned}Q_A = {} & !^{i \leq n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : key; \mathbf{new}\ r'' : coins; \\ & \mathbf{let}\ e : bitstring = enc(k2b(k'), k, r'')\ \mathbf{in} \\ & \mathbf{out}(c_A, (e, mac(e, mk)))\end{aligned}$$

$$\begin{aligned}Q_B = {} & !^{i' \leq n}\, \mathbf{in}(c_B, (e' : bitstring, ma : macstring)); \\ & \mathbf{if}\ verify(e', mk, ma)\ \mathbf{then} \\ & \mathbf{let}\ i_\perp(k2b(k'')) = dec(e', k)\ \mathbf{in}\ \mathbf{out}(c_B, ())\end{aligned}$$

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is IND-CPA (indistinguishable under chosen plaintext attacks).
  An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is IND-CPA (indistinguishable under chosen plaintext attacks).
  An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.
- All keys have the same length: **forall** $y : key; Z(k2b(y)) = Z_k$.

Security properties to prove

In the example:

- One-session secrecy of k'': each k'' is indistinguishable from a random number.
- Secrecy of k'': the k'' are indistinguishable from independent random numbers.

Demo

- CryptoVerif input file: enc-then-MAC.cv
- library of primitives
- run CryptoVerif
- output

Arrays

A variable defined under a replication is implicitly an array:

$$
\begin{aligned}
Q_A = {}!^{i \le n}\, &\mathbf{in}(c_A, ()); \mathbf{new}\ k'[i] : \mathit{key}; \mathbf{new}\ r''[i] : \mathit{coins};\\
&\mathbf{let}\ e[i] : \mathit{bitstring} = \mathit{enc}(k2b(k'[i]), k, r''[i])\ \mathbf{in}\\
&\mathbf{out}(c_A, (e[i], \mathit{mac}(e[i], \mathit{mk})))
\end{aligned}
$$

Requirements:

- Only variables with the current indices can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indices.
  (**if** … **then let** $x = M$ **in** P **else let** $x = M'$ **in** P' is ok)

So each array cell can be assigned at most once.

Arrays allow one to remember the values of all variables during the whole execution

Arrays (continued)

find performs an array lookup:

$$!^{i \leq N} \ldots \textbf{let } x = M \textbf{ in } P$$
$$\| \; !^{i' \leq N'} \textbf{in}(c, y : T); \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge y = x[j] \textbf{ then } \ldots$$

Note that **find** is here used outside the scope of x.

This is the only way of getting access to values of variables in other sessions.

When several array elements satisfy the condition of the **find**, the returned index is chosen randomly, with uniform probability.

Arrays (continued)

find performs an array lookup:

$$
\begin{array}{l}
\;\; !^{i \leq N} \ldots \textbf{let } x[i] = M \textbf{ in } P \\
\| \; !^{i' \leq N'} \textbf{in}(c, y : T); \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge y = x[j] \textbf{ then } \ldots
\end{array}
$$

Note that **find** is here used outside the scope of x.

This is the only way of getting access to values of variables in other sessions.

When several array elements satisfy the condition of the **find**, the returned index is chosen randomly, with uniform probability.

Arrays versus lists

Arrays replace lists often used in cryptographic proofs.

$$
\begin{aligned}
&!^{i \leq N} \ldots \textbf{let } x = M \textbf{ in let } y = M' \textbf{ in } P \\
\| \ &!^{i' \leq N'} \textbf{in}(c, x' : T); \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge x' = x[j] \textbf{ then} \\
&\qquad P'(y[j])
\end{aligned}
$$

might be written by cryptographers

$$
\begin{aligned}
&!^{i \leq N} \ldots \textbf{let } x = M \textbf{ in let } y = M' \textbf{ in insert } (x, y) \textbf{ in } L; P \\
\| \ &!^{i' \leq N'} \textbf{in}(c, x' : T); \textbf{get } (x, y) \textbf{ in } L \textbf{ suchthat } x' = x; P'(y)
\end{aligned}
$$

Arrays avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.

Arrays versus lists

Arrays replace lists often used in cryptographic proofs.

$$!^{i \leq N} \ldots \textbf{let } x[i] = M \textbf{ in let } y[i] = M' \textbf{ in } P$$
$$\| \ !^{i' \leq N'} \textbf{in}(c, x' : T); \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge x' = x[j] \textbf{ then } P'(y[j])$$

might be written by cryptographers

$$!^{i \leq N} \ldots \textbf{let } x = M \textbf{ in let } y = M' \textbf{ in insert } (x, y) \textbf{ in } L; P$$
$$\| \ !^{i' \leq N'} \textbf{in}(c, x' : T); \textbf{get } (x, y) \textbf{ in } L \textbf{ suchthat } x' = x; P'(y)$$

Arrays avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.

Indistinguishability

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

$$Q_1 \approx Q_2$$

More precisely,

$$Q \approx_p^V Q'$$

when an adversary that has direct access to variables in V has probability at most p of distinguishing Q from Q'.

Indistinguishability

Lemma

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

Proof technique

We transform a game G_0 into an indistinguishable one using:

- indistinguishability properties $L \approx_p R$ given as axioms and that come from security assumptions on primitives. These equivalences are used inside a context:
$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$
- syntactic transformations: simplification, expansion of assignments, ...

We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx \ldots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1+\cdots+p_m} G_m$.

If some trace property holds up to probability p in G_m, then it holds up to probability $p + p_1 + \cdots + p_m$ in G_0.

MAC: definition of security (UF-CMA)

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC.
More formally, $\mathrm{Succ}_{\mathrm{MAC}}^{\mathrm{uf-cma}}(t, q_m, q_v, l)$ is negligible if t is polynomial in the security parameter:

Definition (UnForgeability under Chosen Message Attacks, UF-CMA)

$$\mathrm{Succ}_{\mathrm{MAC}}^{\mathrm{uf-cma}}(t, q_m, q_v, l) =$$

$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l} k \stackrel{R}{\leftarrow} \mathit{mkgen}; (m, s) \leftarrow \mathcal{A}^{\mathit{mac}(.,k), \mathit{verify}(.,k,.)} : \mathit{verify}(m, k, s) \wedge \\ m \text{ was never queried to the oracle } \mathit{mac}(., k) \end{array}\right]$$

where $\mathcal{A}$ runs in time at most t,
calls $\mathit{mac}(., k)$ at most q_m times with messages of length at most l,
calls $\mathit{verify}(., k, .)$ at most q_v times with messages of length at most l.

MAC: intuition behind the CryptoVerif definition

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- so, assuming $k \stackrel{R}{\leftarrow} \mathit{mkgen}$ is used only for generating and verifying MACs, the verification of a MAC with $\mathit{verify}(m, k, t)$ can succeed only if m is in the list (array) of messages whose $\mathit{mac}(\cdot, k)$ has been computed by the protocol
- so we can replace a call to *verify* with an array lookup: if the call to *mac* is $\mathit{mac}(x, k)$, we replace $\mathit{verify}(m, k, t)$ with

$$\textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge (m = x[j]) \wedge \mathit{verify}(m, k, t) \textbf{ then true else false}$$

MAC: CryptoVerif definition

$$verify(m, mkgen(r), mac(m, mkgen(r))) = \mathbf{true}$$

$$\begin{aligned}
&!^{N''}\mathbf{new}\ r : mkeyseed; (\\
&\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),\\
&\quad !^{N'} Overify(m : bitstring, t : macstring) := verify(m, mkgen(r), t))\\
&\approx\\
&!^{N''}\mathbf{new}\ r : mkeyseed; (\\
&\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),\\
&\quad !^{N'} Overify(m : bitstring, t : macstring) :=\\
&\qquad \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge\\
&\qquad\qquad verify(m, mkgen(r), t)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false})
\end{aligned}$$

MAC: CryptoVerif definition

$$verify(m, mkgen(r), mac(m, mkgen(r))) = \mathbf{true}$$

$$!^{N''} \mathbf{new}\ r : mkeyseed; ($$

$$\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),$$

$$\quad !^{N'} Overify(m : bitstring, t : macstring) := verify(m, mkgen(r), t))$$

$$\approx_{N'' \times \mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(\mathbf{time}+(N''-1)(\mathbf{time}(mkgen)+N\,\mathbf{time}(mac,\mathrm{maxl}(x))+ N'\,\mathbf{time}(verify,\mathrm{maxl}(m)),N,N',\max(\mathrm{maxl}(x),\mathrm{maxl}(m)))}$$

$$!^{N''} \mathbf{new}\ r : mkeyseed; ($$

$$\quad !^{N} Omac(x : bitstring) := mac'(x, mkgen'(r)),$$

$$\quad !^{N'} Overify(m : bitstring, t : macstring) :=$$

$$\quad\quad \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge$$

$$\quad\quad\quad verify'(m, mkgen'(r), t)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false})$$

CryptoVerif understands such specifications of primitives.

MAC: using the CryptoVerif definition

CryptoVerif applies the previous rule automatically in any context, perhaps containing several occurrences of *mac* and of *verify*:

- Each occurrence of *mac* is replaced with mac'.
- Each occurrence of *verify* is replaced with a **find** that looks in all arrays of computed MACs (one array for each occurrence of function *mac*).

Symmetric encryption: definition of security (IND-CPA)

An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

Definition (INDistinguishability under Chosen Plaintext Attacks, IND-CPA)

$$\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(t, q_e, l) = \max_{\mathcal{A}} 2\Pr\left[b \stackrel{R}{\leftarrow} \{0,1\}; k \stackrel{R}{\leftarrow} kgen; b' \leftarrow \mathcal{A}^{enc(LR(.,.,b),k)} : b' = b\right] - 1$$

where $\mathcal{A}$ runs in time at most t,
calls $enc(LR(.,.,b),k)$ at most q_e times on messages of length at most l,
$LR(x,y,0) = x$, $LR(x,y,1) = y$, and $LR(x,y,b)$ is defined only when x and y have the same length.

Symmetric encryption: CryptoVerif definition

$$dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$$

$$\begin{aligned} &!^{N'} \textbf{new } r : keyseed; !^{N} Oenc(x : bitstring) := \\ &\qquad \textbf{new } r' : coins; enc(x, kgen(r), r') \\ &\approx \\ &!^{N'} \textbf{new } r : keyseed; !^{N} Oenc(x : bitstring) := \\ &\qquad \textbf{new } r' : coins; enc(Z(x), kgen(r), r') \end{aligned}$$

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all $x : nonce$, $Z(x) = Znonce$, $\ldots$).

Symmetric encryption: CryptoVerif definition

$$dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$$

$$!^{N'} \textbf{new } r : keyseed; !^{N} Oenc(x : bitstring) := \\ \textbf{new } r' : coins; enc(x, kgen(r), r')$$

$$\approx_{N' \times \mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\textbf{time}+(N'-1)(\textbf{time}(kgen)+N\,\textbf{time}(enc,\mathrm{maxl}(x))+N\,\textbf{time}(Z,\mathrm{maxl}(x))), N,\mathrm{maxl}(x))}$$

$$!^{N'} \textbf{new } r : keyseed; !^{N} Oenc(x : bitstring) := \\ \textbf{new } r' : coins; enc'(Z(x), kgen'(r), r')$$

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all $x : nonce$, $Z(x) = Znonce$, $\ldots$).

Syntactic transformations (1)

Expansion of assignments: replacing a variable with its value.
(Not completely trivial because of array references.)

Example

If mk is defined by

$$\textbf{let } mk = mkgen(r')$$

and there are no array references to mk, then mk is replaced with $mkgen(r')$ in the game and the definition of mk is removed.

Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment.
(Not completely trivial because of array references.)

Example

$$\mathbf{in}(start, ()); \mathbf{new}\ r_A : T_r; \mathbf{let}\ k_A = kgen(r_A)\ \mathbf{in}$$
$$\mathbf{new}\ r_B : T_r; \mathbf{let}\ k_B = kgen(r_B)\ \mathbf{in}\ \mathbf{out}(c, ()); (Q_K \parallel Q_S)$$
$$Q_K = !^{i \leq n}\, \mathbf{in}(c, (h : T_h, k : T_k))$$
$$\mathbf{if}\ h = A\ \mathbf{then}\ \mathbf{let}\ k' = k_A\ \mathbf{else}$$
$$\mathbf{if}\ h = B\ \mathbf{then}\ \mathbf{let}\ k' = k_B\ \mathbf{else}\ \mathbf{let}\ k' = k$$
$$Q_S = !^{i' \leq n'}\, \mathbf{in}(c', h' : T_h);$$
$$\mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(h[j], k'[j]) \wedge h' = h[j]\ \mathbf{then}\ P_1(k'[j])$$
$$\mathbf{else}\ P_2$$

Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment.
(Not completely trivial because of array references.)

Example

$$\mathbf{in}(start, ()); \mathbf{new}\ r_A : T_r; \mathbf{let}\ k_A = kgen(r_A)\ \mathbf{in}$$
$$\mathbf{new}\ r_B : T_r; \mathbf{let}\ k_B = kgen(r_B)\ \mathbf{in}\ \mathbf{out}(c, ()); (Q_K \parallel Q_S)$$

$$Q_K = !^{i \leq n}\mathbf{in}(c, (h : T_h, k : T_k))$$
$$\mathbf{if}\ h = A\ \mathbf{then}\ \mathbf{let}\ k'_1 = k_A\ \mathbf{else}$$
$$\mathbf{if}\ h = B\ \mathbf{then}\ \mathbf{let}\ k'_2 = k_B\ \mathbf{else}\ \mathbf{let}\ k'_3 = k$$

$$Q_S = !^{i' \leq n'}\mathbf{in}(c', h' : T_h);$$
$$\mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(h[j], k'_1[j]) \wedge h' = h[j]\ \mathbf{then}\ P_1(k'_1[j])$$
$$\mathbf{orfind}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(h[j], k'_2[j]) \wedge h' = h[j]\ \mathbf{then}\ P_1(k'_2[j])$$
$$\mathbf{orfind}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(h[j], k'_3[j]) \wedge h' = h[j]\ \mathbf{then}\ P_1(k'_3[j])$$
$$\mathbf{else}\ P_2$$

Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - Assignments: **let** $x = M$ **in** P implies that $x = M$ in P
  - Tests: **if** $M = N$ **then** P implies that $M = N$ in P
  - Definitions of cryptographic primitives
  - When a **find** guarantees that $x[j]$ is defined, equalities that hold at definition of x also hold under the find (after substituting j for the array indices at the definition of x)
  - Elimination of collisions: if x is created by **new** $x : T$, $x[i] = x[j]$ implies $i = j$, up to negligible probability (when T is large)
- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly. For instance:
  - If M simplifies to **true**, then **if** M **then** P_1 **else** P_2 simplifies P_1.
  - If a condition of **find** simplifies to **false**, then the corresponding branch is removed.

Proof of security properties: one-session secrecy

One-session secrecy: the adversary cannot distinguish any of the secrets from a random number with one test query.

Criterion for proving one-session secrecy of x:
x is defined by **new** $x[i] : T$ and there is a set of variables S such that only variables in S depend on x.
The output messages and the control-flow do not depend on x.

Proof of security properties: secrecy

Secrecy: the adversary cannot distinguish the secrets from independent random numbers with several test queries.

Criterion for proving secrecy of x: same as one-session secrecy, plus $x[i]$ and $x[i']$ do not come from the same copy of the same restriction when $i \neq i'$.

Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, it tries to analyze why the transformation failed, and suggests syntactic transformations that could make it work.
- One tries to execute these syntactic transformations. (If they fail, they may also suggest other syntactic transformations, which are then executed.)
- We retry the cryptographic transformation, and so on.

Proof of the example: initial game

$$
\begin{aligned}
Q_0 = {} & \mathbf{in}(\mathit{start}, ()); \mathbf{new}\ r : \mathit{keyseed}; \mathbf{let}\ k : \mathit{key} = \mathit{kgen}(r)\ \mathbf{in} \\
& \mathbf{new}\ r' : \mathit{mkeyseed}; \mathbf{let}\ \mathit{mk} : \mathit{mkey} = \mathit{mkgen}(r')\ \mathbf{in}\ \mathbf{out}(c, ()); \\
& (Q_A \parallel Q_B)
\end{aligned}
$$

$$
\begin{aligned}
Q_A = {} & !^{i \le n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : \mathit{key}; \mathbf{new}\ r'' : \mathit{coins}; \\
& \mathbf{let}\ m : \mathit{bitstring} = \mathit{enc}(\mathit{k2b}(k'), k, r'')\ \mathbf{in} \\
& \mathbf{out}(c_A, (m, \mathit{mac}(m, \mathit{mk})))
\end{aligned}
$$

$$
\begin{aligned}
Q_B = {} & !^{i' \le n}\, \mathbf{in}(c_B, (m' : \mathit{bitstring}, \mathit{ma} : \mathit{macstring})); \\
& \mathbf{if}\ \mathit{verify}(m', \mathit{mk}, \mathit{ma})\ \mathbf{then} \\
& \mathbf{let}\ i_\perp(\mathit{k2b}(k'')) = \mathit{dec}(m', k)\ \mathbf{in}\ \mathbf{out}(c_B, ())
\end{aligned}
$$

Proof of the example: remove assignments *mk*

$$Q_0 = \textbf{in}(start, ()); \textbf{new}\ r : keyseed; \textbf{let}\ k : key = kgen(r)\ \textbf{in}$$
$$\textbf{new}\ r' : mkeyseed; \textbf{out}(c, ()); (Q_A \parallel Q_B)$$

$$Q_A = !^{i \le n}\, \textbf{in}(c_A, ()); \textbf{new}\ k' : key; \textbf{new}\ r'' : coins;$$
$$\textbf{let}\ m : bitstring = enc(k2b(k'), k, r'')\ \textbf{in}$$
$$\textbf{out}(c_A, (m, mac(m, mkgen(r'))))$$

$$Q_B = !^{i' \le n}\, \textbf{in}(c_B, (m' : bitstring, ma : macstring));$$
$$\textbf{if}\ verify(m', mkgen(r'), ma)\ \textbf{then}$$
$$\textbf{let}\ i_\perp(k2b(k'')) = dec(m', k)\ \textbf{in}\ \textbf{out}(c_B, ())$$

Proof of the example: security of the MAC

$$Q_0 = \textbf{in}(start, ()); \textbf{new}\ r : keyseed; \textbf{let}\ k : key = kgen(r)\ \textbf{in}$$
$$\textbf{new}\ r' : mkeyseed; \textbf{out}(c, ()); (Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n}\ \textbf{in}(c_A, ()); \textbf{new}\ k' : key; \textbf{new}\ r'' : coins;$$
$$\textbf{let}\ m : bitstring = enc(k2b(k'), k, r'')\ \textbf{in}$$
$$\textbf{out}(c_A, (m, mac'(m, mkgen'(r'))))$$

$$Q_B = !^{i' \leq n}\ \textbf{in}(c_B, (m' : bitstring, ma : macstring));$$
$$\textbf{find}\ j \leq n\ \textbf{suchthat}\ \textbf{defined}(m[j]) \wedge m' = m[j] \wedge$$
$$verify'(m', mkgen'(r'), ma)\ \textbf{then}$$
$$\textbf{let}\ i_{\perp}(k2b(k'')) = dec(m', k)\ \textbf{in}\ \textbf{out}(c_B, ())$$

Probability: $\mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(\textbf{time} + \textbf{time}(kgen) + n\,\textbf{time}(enc, \mathrm{length}(key)) + n\,\textbf{time}(dec, \mathrm{maxl}(m')), n, n, \max(\mathrm{maxl}(m'), \mathrm{maxl}(m)))$.

Proof of the example: simplify

$$Q_0 = \textbf{in}(start, ()); \textbf{new}\ r : keyseed; \textbf{let}\ k : key = kgen(r)\ \textbf{in}$$
$$\textbf{new}\ r' : mkeyseed; \textbf{out}(c, ()); (Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n} \textbf{in}(c_A, ()); \textbf{new}\ k' : key; \textbf{new}\ r'' : coins;$$
$$\textbf{let}\ m : bitstring = enc(k2b(k'), k, r'')\ \textbf{in}$$
$$\textbf{out}(c_A, (m, mac'(m, mkgen'(r'))))$$

$$Q_B = !^{i' \leq n} \textbf{in}(c_B, (m' : bitstring, ma : macstring));$$
$$\textbf{find}\ j \leq n\ \textbf{suchthat}\ \textbf{defined}(m[j]) \land m' = m[j] \land$$
$$verify'(m', mkgen'(r'), ma)\ \textbf{then}$$
$$\textbf{let}\ k'' = k'[j]\ \textbf{in}\ \textbf{out}(c_B, ())$$

$$dec(m', k) = dec(enc(k2b(k'[j]), k, r''[j]), k) = i_\perp(k2b(k'[j]))$$

Proof of the example: remove assignments k

$$
\begin{aligned}
Q_0 = {} & \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{new}\ r' : mkeyseed; \mathbf{out}(c, ());\\
& (Q_A \parallel Q_B)
\end{aligned}
$$

$$
\begin{aligned}
Q_A = {} & !^{i \leq n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : key; \mathbf{new}\ r'' : coins;\\
& \mathbf{let}\ m : bitstring = enc(k2b(k'), kgen(r), r'')\ \mathbf{in}\\
& \mathbf{out}(c_A, (m, mac'(m, mkgen'(r'))))
\end{aligned}
$$

$$
\begin{aligned}
Q_B = {} & !^{i' \leq n}\, \mathbf{in}(c_B, (m' : bitstring, ma : macstring));\\
& \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(m[j]) \wedge m' = m[j] \wedge\\
& \quad verify'(m', mkgen'(r'), ma)\ \mathbf{then}\\
& \mathbf{let}\ k'' = k'[j]\ \mathbf{in}\ \mathbf{out}(c_B, ())
\end{aligned}
$$

Proof of the example: security of the encryption

$$Q_0 = \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{new}\ r' : mkeyseed; \mathbf{out}(c, ()); \\ (Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : key; \mathbf{new}\ r'' : coins; \\ \mathbf{let}\ m : bitstring = enc'(Z(k2b(k')), kgen'(r), r'')\ \mathbf{in} \\ \mathbf{out}(c_A, (m, mac'(m, mkgen'(r'))))$$

$$Q_B = !^{i' \leq n}\, \mathbf{in}(c_B, (m' : bitstring, ma : macstring)); \\ \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(m[j]) \wedge m' = m[j] \wedge \\ verify'(m', mkgen'(r'), ma)\ \mathbf{then} \\ \mathbf{let}\ k'' = k'[j]\ \mathbf{in}\ \mathbf{out}(c_B, ())$$

Probability: $\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\mathbf{time} + (n + n^2)\mathbf{time}(mkgen) + n\,\mathbf{time}(mac, \mathrm{maxl}(m)) + n^2\,\mathbf{time}(verify, \mathrm{maxl}(m')) + n^2\,\mathbf{time}(= bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(key))$

Proof of the example: security of the encryption

$$Q_0 = \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{new}\ r' : mkeyseed; \mathbf{out}(c, ()); (Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : key; \mathbf{new}\ r'' : coins; \mathbf{let}\ m : bitstring = enc'(Z(k2b(k')), kgen'(r), r'')\ \mathbf{in}\ \mathbf{out}(c_A, (m, mac'(m, mkgen'(r'))))$$

$$Q_B = !^{i' \leq n}\, \mathbf{in}(c_B, (m' : bitstring, ma : macstring)); \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(m[j]) \wedge m' = m[j] \wedge verify'(m', mkgen'(r'), ma)\ \mathbf{then}\ \mathbf{let}\ k'' = k'[j]\ \mathbf{in}\ \mathbf{out}(c_B, ())$$

Better probability: $\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\mathbf{time} + (n + n^2)\mathbf{time}(mkgen) + n\,\mathbf{time}(mac, \mathrm{maxl}(m)) + n^2\,\mathbf{time}(verify, \mathrm{maxl}(m')) + n^2\,\mathbf{time}(= bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(key))$

Proof of the example: simplify

$$Q_0 = \mathbf{in}(\mathit{start}, ()); \mathbf{new}\ r : \mathit{keyseed}; \mathbf{new}\ r' : \mathit{mkeyseed}; \mathbf{out}(c, ());$$
$$(Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n}\, \mathbf{in}(c_A, ()); \mathbf{new}\ k' : \mathit{key}; \mathbf{new}\ r'' : \mathit{coins};$$
$$\mathbf{let}\ m : \mathit{bitstring} = \mathit{enc}'(Z_k, \mathit{kgen}'(r), r'')\ \mathbf{in}$$
$$\mathbf{out}(c_A, (m, \mathit{mac}'(m, \mathit{mkgen}'(r'))))$$

$$Q_B = !^{i' \leq n}\, \mathbf{in}(c_B, (m' : \mathit{bitstring}, \mathit{ma} : \mathit{macstring}));$$
$$\mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(m[j]) \wedge m' = m[j] \wedge$$
$$\mathit{verify}'(m', \mathit{mkgen}'(r'), \mathit{ma})\ \mathbf{then}$$
$$\mathbf{let}\ k'' = k'[j]\ \mathbf{in}\ \mathbf{out}(c_B, ())$$

$$Z(k2b(k')) = Z_k$$

Proof of the example: secrecy

$$Q_0 = \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{new}\ r' : mkeyseed; \mathbf{out}(c, ());$$
$$(Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n}\,\mathbf{in}(c_A, ()); \color{red}{\mathbf{new}\ k' : key}; \mathbf{new}\ r'' : coins;$$
$$\mathbf{let}\ m : bitstring = enc'(Z_k, kgen'(r), r'')\ \mathbf{in}$$
$$\mathbf{out}(c_A, (m, mac'(m, mkgen'(r'))))$$

$$Q_B = !^{i' \leq n}\,\mathbf{in}(c_B, (m' : bitstring, ma : macstring));$$
$$\mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(m[j]) \wedge m' = m[j] \wedge$$
$$verify'(m', mkgen'(r'), ma)\ \mathbf{then}$$
$$\mathbf{let}\ \color{red}{k'' = k'[j]}\ \mathbf{in}\ \mathbf{out}(c_B, ())$$

Preserves the one-session secrecy of k'' but not its secrecy.

Final result

Adding the probabilities, we obtain:

Result

The probability that an adversary that runs in time at most t, that executes n sessions of A and B and sends messages of length at most l_{mB} to B breaks the one-session secrecy of k'' is

$$2\mathrm{Succ}_{\mathrm{MAC}}^{\mathrm{uf-cma}}(t'_1, n, n, \max(l_{mB}, l_c)) + 2\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(t'_2, n, l_k)$$

where
$$\begin{aligned}
t'_1 &= t + \mathbf{time}(kgen) + n\,\mathbf{time}(enc, l_k) + n\,\mathbf{time}(dec, l_{mB}) \\
t'_2 &= t + (n + n^2)\mathbf{time}(mkgen) + n\,\mathbf{time}(mac, l_c) + \\
&\quad n^2\,\mathbf{time}(verify, l_{mB}) + n^2\,\mathbf{time}(= bitstring, l_{mB}, l_c)
\end{aligned}$$

l_k is the length of keys, l_c the length of encryptions of keys.

The factor 2 comes from the definition of secrecy.

Experiments

Tested on the following protocols (original and corrected versions):
– Otway-Rees (shared-key)
– Yahalom (shared-key)
– Denning-Sacco (public-key)
– Woo-Lam shared-key and public-key
– Needham-Schroeder shared-key and public-key

Shared-key encryption is implemented as encrypt-then-MAC, using a IND-CPA encryption scheme.

(For Otway-Rees, we also considered a SPRP encryption scheme,
a IND-CPA + INT-CTXT encryption scheme,
a IND-CCA2 + IND-PTXT encryption scheme.)

Public-key encryption is assumed to be IND-CCA2.

We prove secrecy of session keys and correspondence properties.

Results (1)

In most cases, the prover succeeds in proving the desired properties when they hold, and obviously it always fails to prove them when they do not hold.

Only cases in which the prover fails although the property holds:

- Needham-Schroeder public-key when the exchanged key is the nonce N_A.
- Needham-Schroeder shared-key: fails to prove that $N_B[i] \neq N_B[i'] - 1$ with overwhelming probability, where N_B is a nonce

Results (2)

- Some public-key protocols need manual proofs.
  (Give the cryptographic proof steps and single assignment renaming instructions.)
- Runtime: 7 ms to 35 s, average: 5 s on a Pentium M 1.8 GHz.

Other case studies

- Full domain hash signature (with David Pointcheval)
  Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- OEKE (variant of Encrypted Key Exchange, with David Pointcheval).
- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA).
- SSH Transport Layer Protocol (with David Cadé).

Conclusion

CryptoVerif can automatically prove the security of primitives and protocols.

- The security assumptions are given as indistinguishability properties (proved manually once).
- The protocol or scheme to prove is specified in a process calculus.
- The prover provides a sequence of indistinguishable games that lead to the proof and a bound on the probability of an attack.
- The user is allowed (but does not have) to interact with the prover to make it follow a specific sequence of games.

It can also generate OCaml implementations of the protocols it proves.

Future work: CryptoVerif extensions

- Support more primitives:
  - Primitives with internal state
- Improved games transformations.
- Improvements in the proof strategy.
  More precise manual hints?
- More case studies.
  - Will suggest more extensions.
- Combine CryptoVerif with EasyCrypt.
  - Make the easy steps automatically with CryptoVerif and the more difficult steps manually with EasyCrypt.
  - Obtain an additional confidence in the proof by duplicating it in both tools.

Acknowledgments

- I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him.
- Work partly supported by the ANR projects FormaCrypt (ARA SSIA 2005) and ProSe (VERSO 2010).