CryptoVerif: A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, INRIA, Paris

October 2007

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.

An automatic prover

We have implemented an automatic prover:

- 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, ...
- 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 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.
  The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property can be read directly on it.
  (The advantage of the adversary is 0 for this game.)

Process calculus for games

A game is formalized in a process calculus, essentially an extension of the pi calculus.

This calculus is inspired by:
– the calculus of [Lincoln, Mitchell, Mitchell, Scedrov]
– the calculus of [Laud, CCS'05]

The semantics is purely probabilistic (no non-determinism).

The runtime of processes is polynomial in the security parameter:
– polynomial number of copies of processes
– length of messages on channels bounded by polynomials

Extension to arrays.

Process calculus for games: terms

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

Function symbols f correspond to functions computable by polynomial-time deterministic Turing machines.

Process calculus for games: processes

$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq N} Q$	replication N times
newChannel $c; Q$	restriction for channels
$c(x_1 : T_1, \ldots, x_m : T_m); P$	input

$P ::=$	output process
$\overline{c}\langle M_1, \ldots, M_m \rangle; Q$	output
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**$(x[j], \ldots) \wedge M$ **then** P **else** P'	array lookup

Example

$$A \rightarrow B : \{x'_k\}_{x_k, x_{mk}}$$

where symmetric encryption is coded as encrypt-then-MAC.

$$\begin{aligned}
Q_0 = {} & \mathit{start}(); \textbf{new } x_r : \mathit{keyseed}; \textbf{let } x_k : \mathit{key} = \mathit{kgen}(x_r) \textbf{ in} \\
& \textbf{new } x'_r : \mathit{mkeyseed}; \textbf{let } x_{mk} : \mathit{mkey} = \mathit{mkgen}(x'_r) \textbf{ in } \overline{c}\langle\rangle; (Q_A \mid Q_B)
\end{aligned}$$

$$\begin{aligned}
Q_A = {} & !^{i \leq n} c_A(); \textbf{new } x'_k : \mathit{key}; \textbf{new } x''_r : \mathit{coins}; \\
& \textbf{let } x_m : \mathit{bitstring} = \mathit{enc}(k2b(x'_k), x_k, x''_r) \textbf{ in} \\
& \overline{c_A}\langle x_m, \mathit{mac}(x_m, x_{mk})\rangle
\end{aligned}$$

$$\begin{aligned}
Q_B = {} & !^{i' \leq n} c_B(x'_m : \mathit{bitstring}, x_{ma} : \mathit{macstring}); \\
& \textbf{if } \mathit{check}(x'_m, x_{mk}, x_{ma}) \textbf{ then} \\
& \textbf{let } i_\perp(k2b(x''_k)) = \mathit{dec}(x'_m, x_k) \textbf{ in } \overline{c_B}\langle\rangle
\end{aligned}$$

Arrays

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

$$!^{i \leq N} \ldots \textbf{let } x \quad = M \textbf{ in } \ldots$$

in fact defines $x[i]$, for i in $1, \ldots, N$.
Under $!^{i \leq N}$, we write x for $x[i]$.

Requirements:

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

So each array cell can be assigned at most once.

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

$$!^{i \leq N} \ldots \textbf{let } x[i] = M \textbf{ in } \ldots$$

in fact defines $x[i]$, for i in $1, \ldots, N$.
Under $!^{i \leq N}$, we write x for $x[i]$.

Requirements:

- Only variables with the current indexes can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indexes.
  (**if** $\ldots$ **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 (continued)

find performs an array lookup:

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

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.

Main notion of security: observational equivalence

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

$$Q_1 \approx Q_2$$

In the formal definition, the adversary is represented by an acceptable evaluation context
$C ::= [\,] \quad C \mid Q \quad Q \mid C \quad \textbf{newChannel}\ c; C.$

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.

MACs: security definition

A MAC takes as input a message and a secret key $mac(m, k)$.
It comes with a checking function *check* such that

$$check(m, k, mac(m, k)) = true$$

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

More formally, an adversary $\mathcal{A}$ that has oracle access to *mac* and *check* has a negligible probability to forge a MAC (UF-CMA):

$$\Pr[check(m, k, t) \mid k \stackrel{R}{\leftarrow} kgen; (m, t) \leftarrow \mathcal{A}^{mac(.,k),check(.,k,.)}] \text{ is negligible}$$

when the adversary $\mathcal{A}$ has not called the *mac* oracle on message m.

MACs: intuitive implementation

By the previous definition, the adversary has a negligible probability of forging a correct MAC.

So when checking a MAC with $check(m, k, t)$ and k is secret, the check can succeed only if m is in the list (array) of messages whose mac has been computed by the protocol.

So we can replace a check with an array lookup:
if the call to mac is $mac(x, k)$, we replace $check(m, k, t)$ with

$$\begin{array}{l} \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge \\ \qquad (m = x[j]) \wedge check(m, k, t) \textbf{ then true else false} \end{array}$$

Furthermore, we use primed function symbols after the transformation, so that it is not done again.

MACs: formal implementation

$$check(m, kgen(r), mac(m, kgen(r))) = \mathbf{true}$$

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

The prover understands such specifications of primitives.

MACs: formal implementation

The prover applies the previous rule automatically in any (polynomial-time) context, perhaps containing several occurrences of *mac* and or *check*:

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

IND-CPA symmetric encryption

We consider a non-deterministic, length-revealing encryption scheme that satisfies INDistinguishability under Chosen Plaintext Attacks (IND-CPA).

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

The IND-CPA property is expressed as follows:

$$\begin{array}{l} !^{N'} \textbf{new } r : keyseed; !^{N}(x : bitstring) \rightarrow \\ \qquad\qquad \textbf{new } r' : coins; enc(x, kgen(r), r') \\ \approx \\ !^{N'} \textbf{new } r : keyseed; !^{N}(x : bitstring) \rightarrow \\ \qquad\qquad \textbf{new } r' : coins; enc'(Z(x), kgen'(r), r') \end{array}$$

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

Syntactic transformations

- 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.)
- Expansion of assignments: replacing a variable with its value.
  (Not completely trivial because of array references.)
- Move new: move restrictions downwards in the game as much as possible, when there is no array reference to them.
  (Moving **new** $x : T$ under a **if** or a **find** duplicates it.
  A subsequent single assignment renaming will distinguish cases.)

Simplification and elimination of collisions

Terms are simplified according to 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 indexes 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)

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 of security properties: correspondences

Correspondences: up to negligible probability, if an event has been executed, then some other events have been executed.

Criterion for proving correspondences: See CSF'07.
The conditions of **find** play a key role:
the **defined** conditions guarantee that, if the branch of the **find** is taken, certain variables are defined, so one of their definitions has been executed.

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

$$Q_0 = start(); \textbf{new } x_r : keyseed; \textbf{let } x_k : key = kgen(x_r) \textbf{ in}$$
$$\textbf{new } x'_r : mkeyseed; \textbf{let } x_{mk} : mkey = mkgen(x'_r) \textbf{ in } \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } x'_k : key; \textbf{new } x''_r : coins;$$
$$\textbf{let } x_m : bitstring = enc(k2b(x'_k), x_k, x''_r) \textbf{ in}$$
$$\overline{c_A}\langle x_m, mac(x_m, x_{mk})\rangle$$

$$Q_B = !^{i' \leq n} c_B(x'_m : bitstring, x_{ma} : macstring);$$
$$\textbf{if } check(x'_m, x_{mk}, x_{ma}) \textbf{ then}$$
$$\textbf{let } i_\perp(k2b(x''_k)) = dec(x'_m, x_k) \textbf{ in } \overline{c_B}\langle\rangle$$

Proof of the example: remove assignments x_{mk}

$$Q_0 = start(); \textbf{new } x_r : keyseed; \textbf{let } x_k : key = kgen(x_r) \textbf{ in}$$
$$\textbf{new } x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } x'_k : key; \textbf{new } x''_r : coins;$$
$$\textbf{let } x_m : bitstring = enc(k2b(x'_k), x_k, x''_r) \textbf{ in}$$
$$\overline{c_A}\langle x_m, mac(x_m, mkgen(x'_r))\rangle$$

$$Q_B = !^{i' \leq n} c_B(x'_m : bitstring, x_{ma} : macstring);$$
$$\textbf{if } check(x'_m, mkgen(x'_r), x_{ma}) \textbf{ then}$$
$$\textbf{let } i_\perp(k2b(x''_k)) = dec(x'_m, x_k) \textbf{ in } \overline{c_B}\langle\rangle$$

Proof of the example: security of the MAC

$$Q_0 = start(); \textbf{new } x_r : keyseed; \textbf{let } x_k : key = kgen(x_r) \textbf{ in}$$
$$\quad \textbf{new } x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } x'_k : key; \textbf{new } x''_r : coins;$$
$$\quad \textbf{let } x_m : bitstring = enc(k2b(x'_k), x_k, x''_r) \textbf{ in}$$
$$\quad \overline{c_A}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle$$

$$Q_B = !^{i' \leq n} c_B(x'_m : bitstring, x_{ma} : macstring);$$
$$\quad \textbf{find } j \leq n \textbf{ suchthat defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge$$
$$\qquad check'(x'_m, mkgen'(x'_r), x_{ma}) \textbf{ then}$$
$$\quad \textbf{let } i_\perp(k2b(x''_k)) = dec(x'_m, x_k) \textbf{ in } \overline{c_B}\langle\rangle$$

Proof of the example: simplify

$$Q_0 = start(); \textbf{new } x_r : keyseed; \textbf{let } x_k : key = kgen(x_r) \textbf{ in}$$
$$\textbf{new } x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } x'_k : key; \textbf{new } x''_r : coins;$$
$$\textbf{let } x_m : bitstring = enc(k2b(x'_k), x_k, x''_r) \textbf{ in}$$
$$\overline{c_A}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle$$

$$Q_B = !^{i' \leq n} c_B(x'_m : bitstring, x_{ma} : macstring);$$
$$\textbf{find } j \leq n \textbf{ suchthat defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge$$
$$check'(x'_m, mkgen'(x'_r), x_{ma}) \textbf{ then}$$
$$\textbf{let } x''_k = x'_k[j] \textbf{ in } \overline{c_B}\langle\rangle$$

$$dec(x'_m, x_k) = dec(enc(k2b(x'_k[j]), x_k, x''_r[j]), x_k) = i_\perp(k2b(x'_k[j]))$$

Proof of the example: remove assignments x_k

$$Q_0 = start(); \textbf{new } x_r : keyseed; \textbf{new } x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned} Q_A = !^{i \leq n} c_A(); &\textbf{new } x'_k : key; \textbf{new } x''_r : coins; \\ &\textbf{let } x_m : bitstring = enc(k2b(x'_k), {\color{red}kgen(x_r)}, x''_r) \textbf{ in} \\ &\overline{c_A}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle \end{aligned}$$

$$\begin{aligned} Q_B = !^{i' \leq n} c_B(&x'_m : bitstring, x_{ma} : macstring); \\ &\textbf{find } j \leq n \textbf{ suchthat defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge \\ &\quad check'(x'_m, mkgen'(x'_r), x_{ma}) \textbf{ then} \\ &\textbf{let } x''_k = x'_k[j] \textbf{ in } \overline{c_B}\langle\rangle \end{aligned}$$

Proof of the example: security of the encryption

$$Q_0 = start(); \textbf{new}\ x_r : keyseed; \textbf{new}\ x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}
Q_A = &!^{i \le n} c_A(); \textbf{new}\ x'_k : key; \textbf{new}\ x''_r : coins; \\
&\quad \textbf{let}\ x_m : bitstring = enc'(Z(k2b(x'_k)), kgen'(x_r), x''_r)\ \textbf{in} \\
&\quad \overline{c_A}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle
\end{aligned}$$

$$\begin{aligned}
Q_B = &!^{i' \le n} c_B(x'_m : bitstring, x_{ma} : macstring); \\
&\quad \textbf{find}\ j \le n\ \textbf{suchthat defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge \\
&\quad\quad check'(x'_m, mkgen'(x'_r), x_{ma})\ \textbf{then} \\
&\quad \textbf{let}\ x''_k = x'_k[j]\ \textbf{in}\ \overline{c_B}\langle\rangle
\end{aligned}$$

Proof of the example: simplify

$$Q_0 = \mathit{start}(); \textbf{new } x_r : \mathit{keyseed}; \textbf{new } x'_r : \mathit{mkeyseed}; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}
Q_A = {} & !^{i \leq n} c_A(); \textbf{new } x'_k : \mathit{key}; \textbf{new } x''_r : \mathit{coins}; \\
& \quad \textbf{let } x_m : \mathit{bitstring} = \mathit{enc}'(Z_k, \mathit{kgen}'(x_r), x''_r) \textbf{ in} \\
& \quad \overline{c_A}\langle x_m, \mathit{mac}'(x_m, \mathit{mkgen}'(x'_r))\rangle
\end{aligned}$$

$$\begin{aligned}
Q_B = {} & !^{i' \leq n} c_B(x'_m : \mathit{bitstring}, x_{ma} : \mathit{macstring}); \\
& \quad \textbf{find } j \leq n \textbf{ suchthat defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge \\
& \quad \quad \mathit{check}'(x'_m, \mathit{mkgen}'(x'_r), x_{ma}) \textbf{ then} \\
& \quad \textbf{let } x''_k = x'_k[j] \textbf{ in } \overline{c_B}\langle\rangle
\end{aligned}$$

$$Z(k2b(x'_k)) = Z_k$$

Proof of the example: secrecy

$Q_0 = start(); \textbf{new}\ x_r : keyseed; \textbf{new}\ x'_r : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$

$Q_A = !^{i \leq n} c_A(); \textbf{new}\ x'_k : key; \textbf{new}\ x''_r : coins;$
$\quad \textbf{let}\ x_m : bitstring = enc'(Z_k, kgen'(x_r), x''_r)\ \textbf{in}$
$\quad \overline{c_A}\langle x_m, mac'(x_m, mkgen'(x'_r))\rangle$

$Q_B = !^{i' \leq n} c_B(x'_m : bitstring, x_{ma} : macstring);$
$\quad \textbf{find}\ j \leq n\ \textbf{suchthat}\ \textbf{defined}(x_m[j]) \wedge x'_m = x_m[j] \wedge$
$\quad\quad check'(x'_m, mkgen'(x'_r), x_{ma})\ \textbf{then}$
$\quad \textbf{let}\ x''_k = x'_k[j]\ \textbf{in}\ \overline{c_B}\langle\rangle$

Preserves the one-session secrecy of x''_k but not its 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
– Full domain hash signature (with D. Pointcheval)
– Encryption schemes of Bellare-Rogaway'93 (with D. Pointcheval)

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
- Showing that the encryption scheme $\mathcal{E}(m, r) = f(r) \| H(r) \oplus m \| H'(m, r)$ is IND-CCA2.

Results (2)

- The 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.

Conclusion

Hopefully a promising approach.

Future extensions:

- Extension to other cryptographic primitives, in particular Diffie-Hellman.
- Improvements in the proof strategy.
- More case studies.

More information: http://www.cryptoverif.ens.fr/

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.
This work was partly supported by the ANR project ARA SSIA FormaCrypt.