CryptoVerif: A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet

CNRS, École Normale Supérieure, INRIA, Paris

June 2009

Introduction

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 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.

Our goal: achieve automatic provability under the realistic computational assumptions.

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 and Bellare&Rogaway'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. 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.)

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).
- All processes run in polynomial time:
  - polynomial number of copies of processes,
  - length of messages on channels bounded by polynomials.

This calculus is inspired by:

- the calculus of [Lincoln, Mitchell, Mitchell, Scedrov, 1998],
- the calculus of [Laud, 2005].

Example

$$A \to B : e = \{x'_k\}_{x_k}, mac(e, x_{mk}) \qquad x'_k \text{ fresh}$$

A sends to B a fresh key x'_k encrypted under authenticated encryption, implemented as encrypt-then-MAC.

x'_k should remain secret.

Example (initialization)

$$A \to B : e = \{x'_k\}_{x_k}, mac(e, x_{mk}) \qquad x'_k \text{ fresh}$$

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

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, x_k and x_{mk} respectively, using the key generation algorithms *kgen* and *mkgen*.
3. Q_0 returns control to the adversary by the output $\overline{c}\langle\rangle$. Q_A and Q_B represent the actions of A and B (see next slides).

Example (role of A)

$$A \to B : e = \{x'_k\}_{x_k}, mac(e, x_{mk}) \qquad x'_k \text{ fresh}$$

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

Role of A:

1. $!^{i \leq n}$ represents n copies, indexes 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 x'_k and sends the message on channel c_A.

Example (role of B)

$$A \rightarrow B : e = \{x'_k\}_{x_k}, mac(e, x_{mk}) \qquad x'_k \text{ fresh}$$

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

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 x''_k.

Example (summary)

$$A \to B : e = \{x'_k\}_{x_k}, mac(e, x_{mk}) \qquad x'_k \text{ fresh}$$

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

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

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

Arrays

Arrays replace lists often used in cryptographic proofs.

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

A variable defined under a replication is implicitly an array:

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

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** ... **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:

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

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.

Indistinguishability as observational equivalence

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

$$Q_1 \approx Q_2$$

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.

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 assumptions on 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.

MACs: security definition

A MAC scheme:

- (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 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 *verify* has a negligible probability to forge a MAC (UF-CMA):

$$\max_{\mathcal{A}} \Pr[verify(m, k, t) \mid k \xleftarrow{R} mkgen; (m, t) \leftarrow \mathcal{A}^{mac(.,k),verify(.,k,.)}]$$

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

MACs: intuitive implementation

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- so when verifying a MAC with $verify(m, k, t)$ and $k \xleftarrow{R} mkgen$ is used only for generating and verifying MACs, the verification 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 call to *verify* with an array lookup: if the call to *mac* is $mac(x, k)$, we replace $verify(m, k, t)$ with

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

MACs: formal implementation

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

$$!^{N''}\textbf{new}\ r : mkeyseed; (!^{N}(x : bitstring) \rightarrow mac(x, mkgen(r)), !^{N'}(m : bitstring, t : macstring) \rightarrow verify(m, mkgen(r), t))$$

$$\approx$$

$$!^{N''}\textbf{new}\ r : mkeyseed; (!^{N}(x : bitstring) \rightarrow mac(x, mkgen(r)), !^{N'}(m : bitsting, t : macstring) \rightarrow \textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j]) \wedge (m = x[j]) \wedge verify(m, mkgen(r), t)\ \textbf{then true else false})$$

The prover understands such specifications of primitives.
They can be reused in the proof of many protocols.

MACs: formal implementation

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

$$!^{N''}\textbf{new}\ r : mkeyseed; (!^{N}(x : bitstring) \rightarrow mac(x, mkgen(r)), !^{N'}(m : bitstring, t : macstring) \rightarrow verify(m, mkgen(r), t))$$

$$\approx$$

$$!^{N''}\textbf{new}\ r : mkeyseed; (!^{N}(x : bitstring) \rightarrow mac'(x, mkgen'(r)), !^{N'}(m : bitsting, t : macstring) \rightarrow \textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j]) \wedge (m = x[j]) \wedge verify'(m, mkgen'(r), t)\ \textbf{then true else false})$$

The prover understands such specifications of primitives.
They can be reused in the proof of many protocols.

MACs: formal implementation

The prover applies the previous rule automatically in any (polynomial-time) 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*).

IND-CPA symmetric encryption

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

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

$$!^{N'}\mathbf{new}\ r : keyseed; !^{N}(x : bitstring) \rightarrow \mathbf{new}\ r' : coins; enc(x, kgen(r), r')$$
$$\approx$$
$$!^{N'}\mathbf{new}\ r : keyseed; !^{N}(x : bitstring) \rightarrow \mathbf{new}\ r' : coins; enc'(Z(x), kgen'(r), r')$$

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

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)

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

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

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

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

Proof of the example: remove assignments x_{mk}

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

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

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

Proof of the example: security of the MAC

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

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

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

Proof of the example: simplify

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

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

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

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

Proof of the example: remove assignments x_k

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

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

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

Proof of the example: security of the encryption

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

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

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

Proof of the example: simplify

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

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

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

$$Z(\mathit{k2b}(x'_k)) = Z_k$$

Proof of the example: secrecy

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

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

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

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)

- 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.
- A detailed case study of Kerberos V, with and without its public-key extension PKINIT (AsiaCCS'08, with with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- Starts being used by others: Verification of F# implementations, including TLS, by Microsoft Research and the MSR-INRIA lab.

Conclusion

Hopefully a promising approach.

Future extensions:

- Extension to other cryptographic primitives, in particular Diffie-Hellman, full support of XOR.
- More game transformations.
- 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.