A Computationally Sound Mechanized Prover for Cryptographic Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris

May 2006

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; currently attracts much attention.

- 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), stream and block ciphers, 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
$x, y, z, x[M_1, \ldots, M_n]$	variable
$f(M_1, \ldots, M_n)$	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
$\mathbf{newChannel}\ c; Q$	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
$\mathbf{new}\ x : T; P$	random number generation (uniform)
$\mathbf{let}\ x : T = M\ \mathbf{in}\ P$	assignment
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	conditional
$\mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j], \ldots) \wedge M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	array lookup

Arrays

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

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

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.
  ($\mathbf{if}\ \ldots\ \mathbf{then\ let}\ x = M\ \mathbf{in}\ P\ \mathbf{else\ let}\ x = M'\ \mathbf{in}\ P'$ is ok)

So each array cell can be assigned at most once.

Arrays

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

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

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.
  ($\mathbf{if}\ \ldots\ \mathbf{then\ let}\ x = M\ \mathbf{in}\ P\ \mathbf{else\ let}\ x = M'\ \mathbf{in}\ P'$ is ok)

So each array cell can be assigned at most once.

Arrays (continued)

find performs an array lookup:

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

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.

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 $C ::= [\,] \quad C \mid Q \quad Q \mid C \quad \mathbf{newChannel}\ c; C$.

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):

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

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

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

MACs: formal implementation

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

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

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).

Stream ciphers

Similarly, the security of stream ciphers is expressed as follows:

$$dec(enc(m, kgen(r), r'), kgen(r)) = m$$

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

A stream cipher is non-deterministic, length-revealing, satisfies INDistinguishability under Chosen Plaintext Attacks (IND-CPA).

$Z(x)$ 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 $\mathbf{new}\ x : T$ under a $\mathbf{if}$ or a $\mathbf{find}$ 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: $\mathbf{let}\ x = M\ \mathbf{in}\ P$ implies that $x = M$ in P
- Tests: $\mathbf{if}\ M = N\ \mathbf{then}\ P$ implies that $M = N$ in P
- Definitions of cryptographic primitives
- When a $\mathbf{find}$ 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 $\mathbf{new}\ x : T$, $x[i] = x[j]$ 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 $\mathbf{new}\ x[i] : T$ 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.

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 David Pointcheval)
– Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)

Shared-key encryption is implemented as encrypt-then-mac, using a IND-CPA stream cipher for encryption.
(For Otway-Rees and Woo-Lam shared-key, we also considered a SPRP block cipher.)
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.
  – 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.

Otway-Rees

M, N_a, N_b fresh nonces; K_{ab} fresh key created by the server.

1	$A \to B$	$M, A, B, e_1 = \{N_a, M, A, B\}_{K_{as}}$
2	$B \to S$	$M, A, B, e_1, \{N_b, M, A, B\}_{K_{bs}}$
3	$S \to B$	$M, e_2 = \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$
4	$B \to A$	M, e_2

Encryption implemented as encrypt-then-mac:
$\{M\}_k$ is in fact $\mathbf{new}\ r : coins; e = enc(M, k, r); e, mac(e, mk)$.

A, B, and S may also talk to dishonest participants.

Proof of Otway-Rees (1)

Simplify; Remove useless assignments

Remove assignments to mK_{bs}
Single assignment renaming of $Rmkey$ (mac key in the key table)
Remove assignments $Rmkey1$, $Rmkey2$, $Rmkey3$
Security of mac for mK_{bs}
Simplify; Remove useless assignments

Remove assignments to mK_{as}
Security of mac for mK_{as}
Simplify; Remove useless assignments

Proof of Otway-Rees (2)

Remove assignments to K_{bs}

Single assignment renaming of $Rkey$ (encryption key in the key table)

Remove assignments $Rkey1$, $Rkey2$, $Rkey3$

Security of enc for K_{bs}

Simplify; Remove useless assignments

Remove assignments to K_{as}

Security of enc for K_{as}

Simplify; Remove useless assignments

Single assignment renaming of K_{ab}

Simplify

Success!

Conclusion

Hopefully a promising approach.

Future extensions:

- Extension to other cryptographic primitives, in particular Diffie-Hellman.

- Extension to other security properties: semantic security of the key, . . .

More information: http://www.di.ens.fr/~blanchet/cryptoc-eng.html

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.

Thank you for your attention.

Questions?