A Computationally Sound Automatic Prover for Cryptographic Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris

June 2005

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

Work in progress!

We have implemented an automatic prover:

- proves secrecy.

- handles macs (message authentication codes) and stream ciphers (plus a few other variants of symmetric encryption).

- works for N sessions (polynomial in the security parameter), with an active adversary.

Extensions to other security properties and primitives are obviously planned.

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.)

A game is formalized as an extension of the pi calculus with function symbols and arrays.

Process calculus for games: terms

Essentially extends the calculus of [Lincoln, Mitchell, Mitchell, Scedrov] with arrays.

$M ::=$	terms
$x, y, z, x[M_1, \ldots, M_n]$	variable
$f(M_1, \ldots, M_n)$	function application
$M = M'$	equality test
$\mathit{if}\ M\ \mathit{then}\ M_1\ \mathit{else}\ M_2$	test
$\mathit{find}\ j \leq N\ \mathit{suchthat}\ \mathit{defined}(x[j], \ldots)\ \&\&\ M\ \mathit{then}\ M_1\ \mathit{else}\ M_2$	array lookup
$\mathit{let}\ x = M\ \mathit{in}\ M'$	assignment

Process calculus for games: processes

$P ::=$	process
$\quad 0$	nil
$\quad P \mid P'$	parallel composition
$\quad !^{i \leq N} P$	replication N times
$\quad c(x : T); P$	input
$\quad \overline{c}\langle M \rangle; P$	output
$\quad \mathit{new}\ x : T; P$	random number generation (uniform)
$\quad \mathit{let}\ x = M\ \mathit{in}\ P$	assignment
$\quad \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ P'$	conditional

$\quad \mathit{find}\ j \leq N\ \mathit{suchthat}\ \mathit{defined}(x[j], \ldots)\ \&\&\ M\ \mathit{then}\ P\ \mathit{else}\ P'$

array lookup

Arrays

Arrays replace lists often used in cryptographic proofs.

A variable defined under a replication is implicitly an array:

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

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

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.
($if\ \dots\ then\ let\ x = M\ in\ P\ else\ let\ x = M'\ in\ P'$ is ok)

Arrays (continued)

find performs an array lookup:

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

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 sessions satisfy the condition of the *find*,
the returned index is chosen randomly, with uniform probability.

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:

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

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

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

MACs: formal implementation

$$check(m, kgen(r), mac(m, kgen(r))) = true$$

$$\begin{array}{l} new\ r : keyseed; (\\ \quad (x : bitstring) \rightarrow_N mac(x, kgen(r)), \\ \quad (m : bitstring, t : macstring) \rightarrow_{N'} check(m, kgen(r), t)) \end{array}$$

$\approx_{\text{up to negligible probability}}$

$$\begin{array}{l} new\ r : keyseed; (\\ \quad (x : bitstring) \rightarrow_N mac'(x, kgen'(r)), \\ \quad (m : bitsting, t : macstring) \rightarrow_{N'} find\ j \leq N\ suchthat\ defined(x[j])\ \&\& \\ \qquad (m = x[j])\ \&\&\ check'(m, kgen'(r), t)\ then\ true\ else\ false) \end{array}$$

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:

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

$$new\ r : keyseed; (x : bitstring) \rightarrow_N new\ r' : IV; enc(x, kgen(r), r')$$

$\approx_{\text{up to negligible probability}}$

$$new\ r : keyseed; (x : bitstring) \rightarrow_N new\ r' : IV; enc'(Z(x), kgen'(r), r')$$

A stream cipher is non-deterministic, length-revealing, resistant to Chosen Plaintext Attacks (CPA).

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

Syntactic transformations

- Expansion of if/find/let: replace an expression if/find/let with a process, by duplicating the code that follows the test.
  This corresponds to performing a case analysis.

- 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.)

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 array indexes)
- Elimination of collisions: for example, if N is created by new, $N[i] = N[j]$ implies $i = j$, up to negligible probability

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.

Results

For the moment, tested on two protocols:

- Otway-Rees, secrecy of the exchanged key successfully proved (runtime 1.15 s on a Pentium M 1.8 GHz).

- Yahalom: the original version is not proved, because the protocol is not secure, at least using encrypt-then-mac as definition of encryption (runtime 0.62 s).

  There is a confirmation round $\{N_B\}_K$ where K is the exchanged key. This message may reveal some information on K.

  If we remove this confirmation round, the secrecy of K is proved (runtime 0.61 s).

Otway-Rees

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

$$\begin{array}{lll} 1 & A \rightarrow B & M, A, B, e_1 = \{N_a, M, A, B\}_{K_{as}} \\ 2 & B \rightarrow S & M, A, B, e_1, \{N_b, M, A, B\}_{K_{bs}} \\ 3 & S \rightarrow B & M, e_2 = \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}} \\ 4 & B \rightarrow A & M, e_2 \end{array}$$

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

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

Proof of Otway-Rees (1)

Expand if/let/find; Simplify

Remove useless assignments

Remove assignments to mK_{bs}

Single assignment renaming of $Rmkey$ (mac key in the key table)

Remove assignments $Rmkey$1, $Rmkey$2, $Rmkey$3

Security of mac for mK_{bs}

Expand if/let/find; Simplify

Remove useless assignments

Remove assignments to mK_{as}

Security of mac for mK_{as}

Expand if/let/find; 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 $Rkey$1, $Rkey$2, $Rkey$3

Security of enc for K_{bs}

Expand if/let/find; Simplify

Remove useless assignments

Remove assignments to K_{as}

Security of enc for K_{as}

Expand if/let/find; Simplify

Remove useless assignments

Single assignment renaming of K_{ab}

Simplify

Success!

Conclusion

Hopefully a promising approach, but still some work to do:

- Extension to other cryptographic primitives: public-key cryptography (encryption and signatures), hash functions, Diffie-Hellman, xor.
  (small extensions to the format of primitive specifications, improvements to the simplification algorithm)

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

- More experiments.

- Detailed proofs.

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?