Automatic verification of security protocols: the tools ProVerif and CryptoVerif

Bruno Blanchet

INRIA, École Normale Supérieure, CNRS, Paris

October 2011

Outline

1. Introduction to security protocols
2. Verification of protocols in the formal model: ProVerif
3. Verification of protocols in the computational model: CryptoVerif
4. Conclusion and future work

Some cryptographic primitives

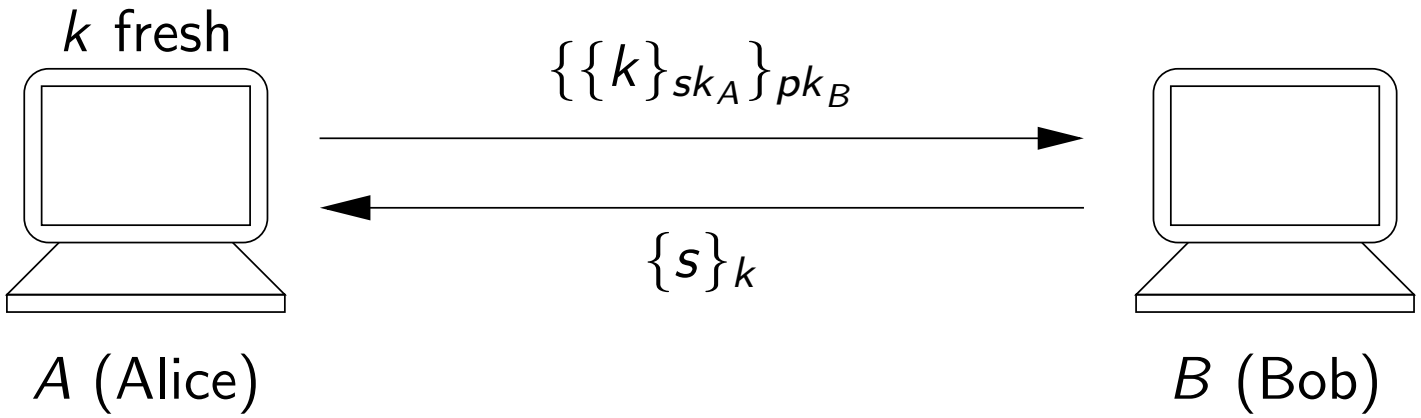
- Encryption: $\{m\}_k$ is the encryption of message m under key k. When you have the decryption key, you can get m from $\{m\}_k$.

  Shared-key encryption: the decryption key is equal to the encryption key.

  Public-key encryption: the decryption key (secret key sk) is different from the encryption key (public key pk).
- Signature: one signs with the secret key sk ($\{m\}_{sk}$), and checks the signature with the public key pk.

Example

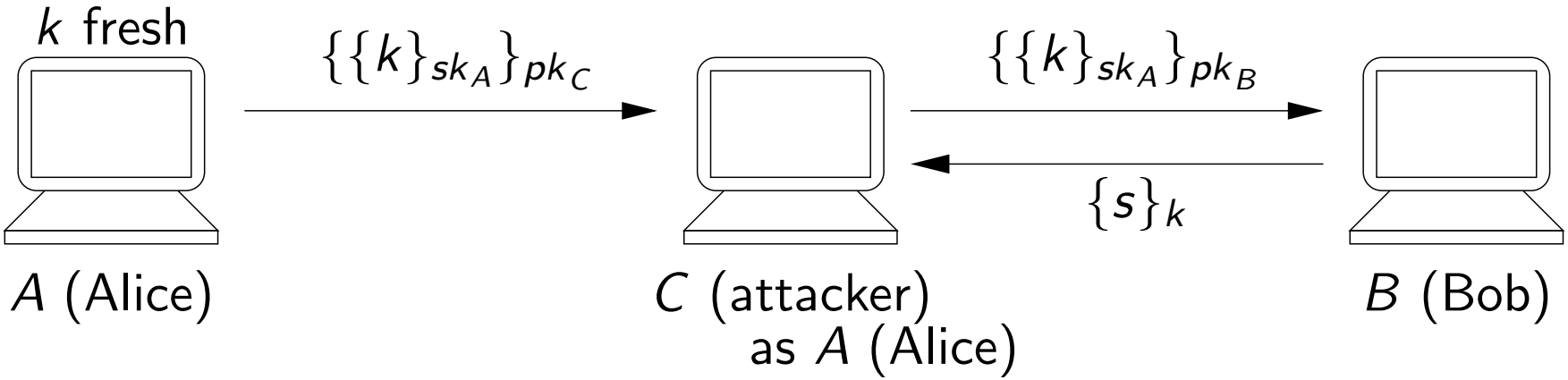
Denning-Sacco key distribution protocol [Denning, Sacco, Comm. ACM, 1981] (simplified)



The goal of the protocol is that the key k should be a secret key, shared between A and B. So s should remain secret.

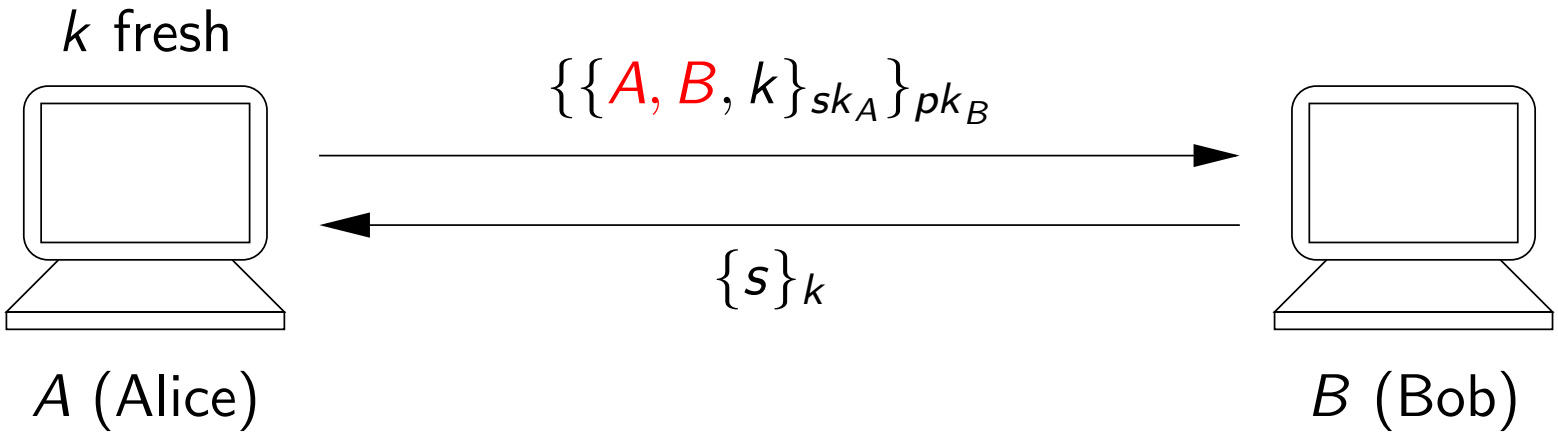
The attack

The (well-known) attack against this protocol.



The attacker C impersonates A and obtains the secret s.

The corrected protocol



Now C cannot impersonate A because in the previous attack, the first message is $\{\{A, C, k\}_{sk_A}\}_{pk_B}$, which is not accepted by B.

Why verify security protocols ?

The verification of security protocols has been and is still a very active research area.

- Their design is error prone.
- Errors are not detected by testing:
  they appear only in the presence of an adversary.
- Errors can have serious consequences.

Model of protocols

Active attacker:

- the attacker can intercept all messages sent on the network
- he can compute messages
- he can send messages on the network

Model of protocols: the formal model

The formal model or "Dolev-Yao model" is due to Needham and Schroeder (1978) and Dolev and Yao (1983).

- The cryptographic primitives are blackboxes.
- The messages are terms on these primitives.
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption

One can add equations between primitives, but in any case, one makes the hypothesis that the only equalities are those given by these equations.

The formal model facilitates automatic proofs.

Models of protocols: the computational model

The computational model has been developped at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

- The messages are bitstrings.
- The cryptographic primitives are functions on bitstrings.
- The attacker is any probabilistic polynomial-time Turing machine.

This model is much more realistic than the formal model, but until recently proofs were only manual.

Security goals

- Secrecy:
  - Formal model:
    - Syntactic secrecy: the attacker cannot have a message s.
    - Strong secrecy: the attacker cannot distinguish when the value of the secret changes.
  - Computational model: the attacker can distinguish the secret from a random number only with negligible probability.
- Authentication: If B thinks he talks to A then A thinks she talks to B.
- Key exchange, fair contract signing, ...

Protocol verification in the formal model

Cryptographic protocols are infinite state:

- The attacker can create messages of unbounded size.
- Unbounded number of sessions of the protocol.

Solutions:

- Bound the state space arbitrarily:
  exhaustive exploration (model-checking, ...);
  find attacks but not prove security.
- Bound the number of sessions:
  the insecurity is NP-complete (with reasonable assumptions).
- Unbounded case:
  the problem is undecidable.

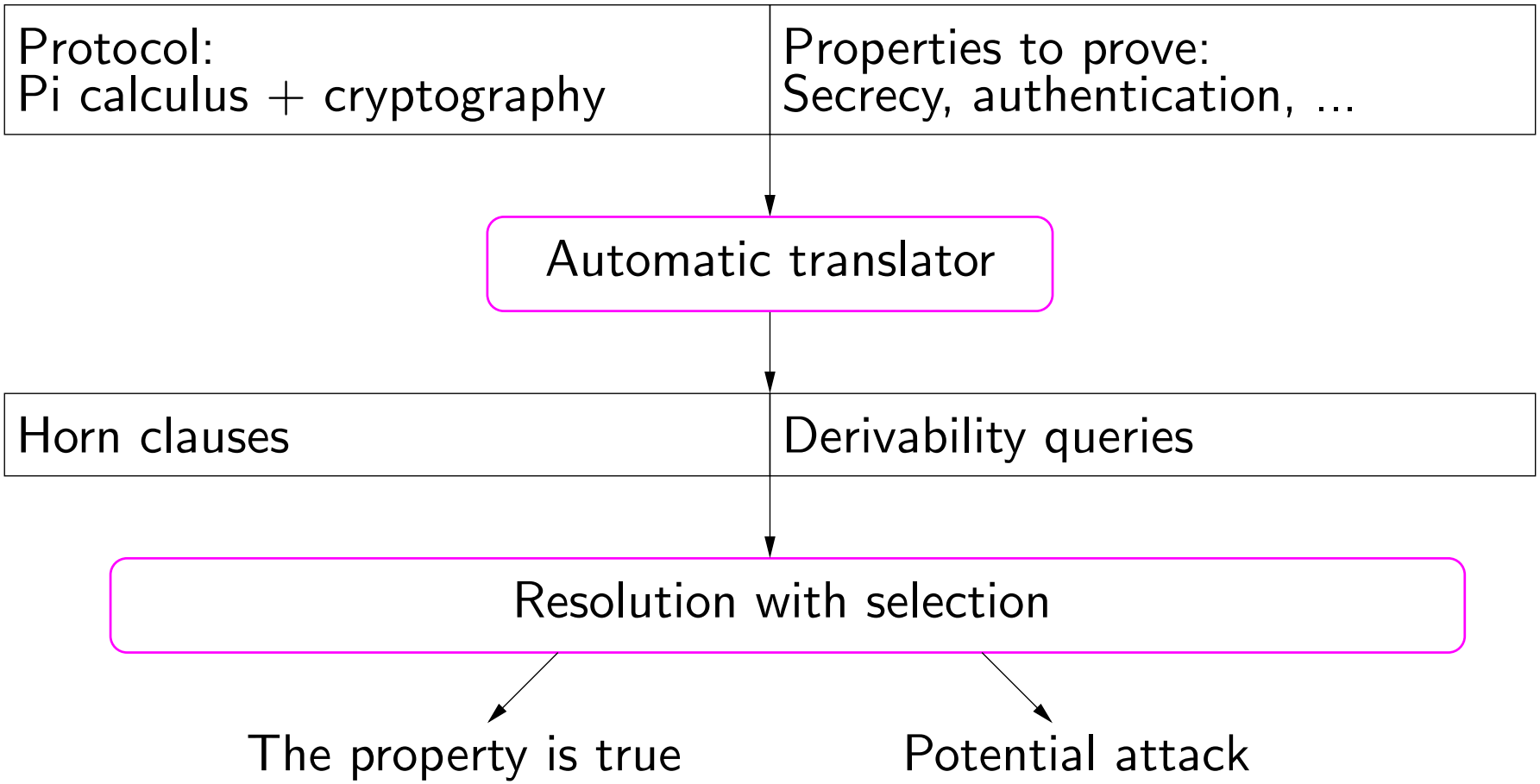
Solutions to undecidability

To solve an undecidable problem, we can

- Use approximations, abstraction.
- Terminate on a restricted class.
- Rely on user interaction or annotations

We do the first two, using a very precise abstraction.

ProVerif


Protocol:
Pi calculus + cryptography
Properties to prove:
Secrecy, authentication, ...
Automatic translator
Horn clauses
Derivability queries
Resolution with selection
The property is true
Potential attack

Features of ProVerif

- Fully automatic.
- Efficient: small examples verified in less than 0.1 s; complex ones in a few minutes.
- Very precise: no false attack in our tests for secrecy and authentication.
- Unbounded number of sessions and message space.
- Handles a wide variety of cryptographic primitives, defined by rewrite rules or equations.
- Handles various security properties: secrecy, authentication, some equivalences.

Definition of cryptographic primitives

Two kinds of operations:

- Constructors f are used to build terms $f(M_1, \ldots, M_n)$

  fun $f(T_1, \ldots, T_n) : T$.

Example: shared-key encryption

fun encrypt(bitstring, key) : bitstring.

- Destructors g manipulate terms
  Destructors are defined by rewrite rules $g(M_1, \ldots, M_n) \longrightarrow M$.

  reduc forall $x_1 : T_1, \ldots, x_k : T_k; g(M_1, \ldots, M_n) = M$.

Example: shared-key decryption decrypt(encrypt$(m, k), k) \longrightarrow m$

reduc forall x : bitstring, y : key; decrypt(encrypt$(x, y), y) = x$.

Syntax of the process calculus

Pi calculus + cryptographic primitives

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\mathbf{out}(M, N); P$	output
$\mathbf{in}(M, x : T); P$	input
$\mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{in}\ P\ \mathbf{else}\ Q$	destructor application
$\mathbf{if}\ M = N\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mathbf{new}\ a : T; P$	restriction
$0 \quad P \mid Q \quad !P$	

Example: The Denning-Sacco protocol

Message 1. $A \rightarrow B : \quad \{\{k\}_{sk_A}\}_{pk_B} \quad k$ fresh
Message 2. $B \rightarrow A : \quad \{s\}_k$

new sk_A : sskey; **let** $pk_A = \mathrm{spk}(sk_A)$ **in**
new sk_B : skey; **let** $pk_B = \mathrm{pk}(sk_B)$ **in**
out(c, pk_A); **out**(c, pk_B);

(A) ! **in**$(c, x\_pk_B$: pkey);
new k : key; **out**$(c, \mathrm{pencrypt}(\mathrm{sign}(\mathrm{k2b}(k), sk_A), x\_pk_B))$;
in$(c, x$: bitstring); **let** $s = \mathrm{decrypt}(x, k)$ **in** 0

(B) | ! **in**$(c, y$: bitstring); **let** $y' = \mathrm{pdecrypt}(y, sk_B)$ **in**
let $\mathrm{k2b}(k) = \mathrm{checksign}(y', pk_A)$ **in** **out**$(c, \mathrm{encrypt}(\mathrm{s}, k))$

Security properties

- Secrecy: The attacker cannot obtain the secret s

  query attacker(s).

- Correspondence assertions: (authentication)
  If an event has been executed, then some other events must have been executed.
- Process equivalences:
  - Strong secrecy
  - Equivalences between processes that differ only by terms they contain (joint work with Martín Abadi and Cédric Fournet)

    In particular, proof of protocols relying on weak secrets.

The Horn clause representation

- The main predicate used by the Horn clause representation of protocols is attacker:

  attacker(M) *means* *"the attacker may have M".*

- We can model actions of the adversary and of the protocol participants thanks to this predicate.
- Processes are automatically translated into Horn clauses (joint work with Martín Abadi).

Coding of primitives

- Constructors $f(M_1, \ldots, M_n)$
  $\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \rightarrow \text{attacker}(f(x_1, \ldots, x_n))$

Example: Shared-key encryption $\text{encrypt}(m, k)$

$\text{attacker}(m) \wedge \text{attacker}(k) \rightarrow \text{attacker}(\text{encrypt}(m, k))$

- Destructors $g(M_1, \ldots, M_n) \rightarrow M$
  $\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \rightarrow \text{attacker}(M)$

Example: Shared-key decryption $\text{decrypt}(\text{encrypt}(m, k), k) \rightarrow m$

$\text{attacker}(\text{encrypt}(m, k)) \wedge \text{attacker}(k) \rightarrow \text{attacker}(m)$

General coding of a protocol

If a principal A has received the messages $M_1, \ldots, M_n$ and sends the message M,

$$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \rightarrow \text{attacker}(M).$$

Example

Upon receipt of a message of the form $\text{pencrypt}(\text{sign}(y, sk_A), pk_B)$, B replies with $\text{encrypt}(\text{s}, y)$:

$$\text{attacker}(\text{pencrypt}(\text{sign}(y, sk_A), pk_B)) \rightarrow \text{attacker}(\text{encrypt}(\text{s}, y))$$

The attacker sends $\text{pencrypt}(\text{sign}(y, sk_A), pk_B)$ to B, and intercepts his reply $\text{encrypt}(\text{s}, y)$.

Proof of secrecy

Theorem (Secrecy)

If attacker(M) *cannot be derived from the clauses, then* M *is secret.*

The term M cannot be built by an attacker.

The resolution algorithm will determine whether a given fact can be derived from the clauses.

Remark

Soundness and completeness are swapped.
The resolution prover is complete
(If attacker(M) is derivable, it finds a derivation.)
$\Rightarrow$ The protocol verifier is sound
(If it proves secrecy, then secrecy is true.)

Resolution with free selection

$$\frac{R = H \to F \qquad R' = F'_1 \wedge H' \to F'}{\sigma H \wedge \sigma H' \to \sigma F'}$$

where σ is the most general unifier of F and F'_1,
F and F'_1 are selected.

The selection function selects:

- a hypothesis not of the form attacker(x) if possible,
- the conclusion otherwise.

Key idea: avoid resolving on facts attacker(x).

Resolve until a fixpoint is reached.
Keep clauses whose conclusion is selected.

Theorem

The obtained clauses derive the same facts as the initial clauses.

Demo

Demo

Denning-Sacco example

Applications

- Tested on many protocols of the literature.
- More ambitious case studies:
  - Certified email (with Martín Abadi)
  - JFK (with Martín Abadi and Cédric Fournet)
  - Plutus (with Avik Chaudhuri)
- Case studies by others:
  - E-voting protocols (Delaune, Kremer, and Ryan; Backes et al)
  - Zero-knowledge protocols, DAA (Backes et al)
  - Shared authorisation data in TCG TPM (Chen and Ryan)
  - Electronic cash (Luo et al)
  - ...
- Extensions and tools:
  - Extension to XOR and Diffie-Hellman (Küsters and Truderung)
  - Web service verifier TulaFale (Microsoft Research).
  - Translation from HLPSL, input language of AVISPA (Gotsman, Massacci, Pistore)

Verification of implementations

- By translation from implementations to a ProVerif model:
  F# implementations, TLS (Microsoft Research and MSR-INRIA)
- By direct translation from implementations to Horn clauses:
  C implementations (Goubault-Larrecq and Parennes).
  Resolution performed via the $\mathcal{H}_1$ prover rather than ProVerif.
- By translation from specifications to implementations:
  Java implementations, Spi2Java tool (Pozza, Pironti, Sisto).
  Also offers the possibility of verifying the specifications via translation to ProVerif.

Protocol verification in the computational model

Two approaches for the automatic proof of security 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.

  Results by Abadi&Rogaway (2000), Cortier&Warinschi (2005), Comon&Cortier (2008), and many others.

- Direct approach:

  Design automatic tools for proving protocols in a computational model.

  Approach pioneered by Laud (2004).

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

The automatic prover CryptoVerif:

- 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).
- 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. 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 usually 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.

Example

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

$$Q_0 = \mathbf{in}(start, ()); \mathbf{new}\ r : keyseed; \mathbf{let}\ k = kgen(r)\ \mathbf{in}$$
$$\mathbf{new}\ r' : mkeyseed; \mathbf{let}\ mk = mkgen(r')\ \mathbf{in}\ \mathbf{out}(c, ()); (Q_A \mid Q_B)$$

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

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

Arrays

The variables defined in repeated processes (under a replication) are arrays, with one cell for each execution, to remember the values used in each execution.
These arrays are indexed with the execution number i, i'.

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

Arrays replace lists generally used by cryptographers.

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

Indistinguishability as observational equivalence

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

$$Q_1 \approx Q_2$$

The adversary is represented by an acceptable evaluation context C (essentially, a process put in parallel with the considered games).

- 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 properties of 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):

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

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 $k \stackrel{R}{\leftarrow} mkgen$ is used only for generating and checking MACs, 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 verification with an array lookup: if the call to *mac* is $mac(x, k)$, we replace $verify(m, k, t)$ with

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

MACs: formal implementation

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

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

$$\approx\ !^{N''}\,\textbf{new}\ r : mkeyseed; (!^{N}\,Omac(x : bitstring) := mac'(x, mkgen'(r)),\ !^{N'}\,Overify(m : bitsting, t : macstring) :=$$

$$\textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j]) \wedge (m = x[j]) \wedge verify'(m, mkgen'(r), t)\ \textbf{then true else false})$$

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

Proof strategy: advice

- CryptoVerif tries to apply all equivalences given as axioms, which represent security assumptions.

  It transforms the left-hand side into the right-hand side of the equivalence.
- If such a transformation succeeds, the obtained game is then simplified, using in particular equations given as axioms.
- When these transformations fail, they may return syntactic transformations to apply in order to make them succeed, called advised transformations.

  CryptoVerif then applies the advised transformations, and retries the initial transformation.

Results

Tested on:

- 16 "Dolev-Yao style" protocols that we study in the computational model. CryptoVerif proves all correct properties except in 3 cases.
- FDH (Full Domain Hash) signature scheme and encryption schemes of Bellare, Rogaway, 1993 (with David Pointcheval).
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- OEKE (One-Encryption Key Exchange), variant of EKE.

Starts being used by others:

- Verification of F# implementations (Microsoft Research and MSR-INRIA).
- TLS (Microsoft Research and MSR-INRIA).

Conclusion and future work

- The automatic prover ProVerif works in the formal model.
  It is essentially mature.
- The automatic prover CryptoVerif works in the computational model.
  Much work still to do on this topic:
  - Improvements to the proof strategy.
  - Handle more cryptographic primitives (stateful encryption, ...)
  - Handle more equations (associativity, ...).
  - Handle more security properties (forward secrecy, ...).
  - Make more case studies.
- An important topic for future work is the verification of implementations of protocols.