Automatic Verification of Security Protocols: Formal Model and Computational Model

Bruno Blanchet

EPI Cascade

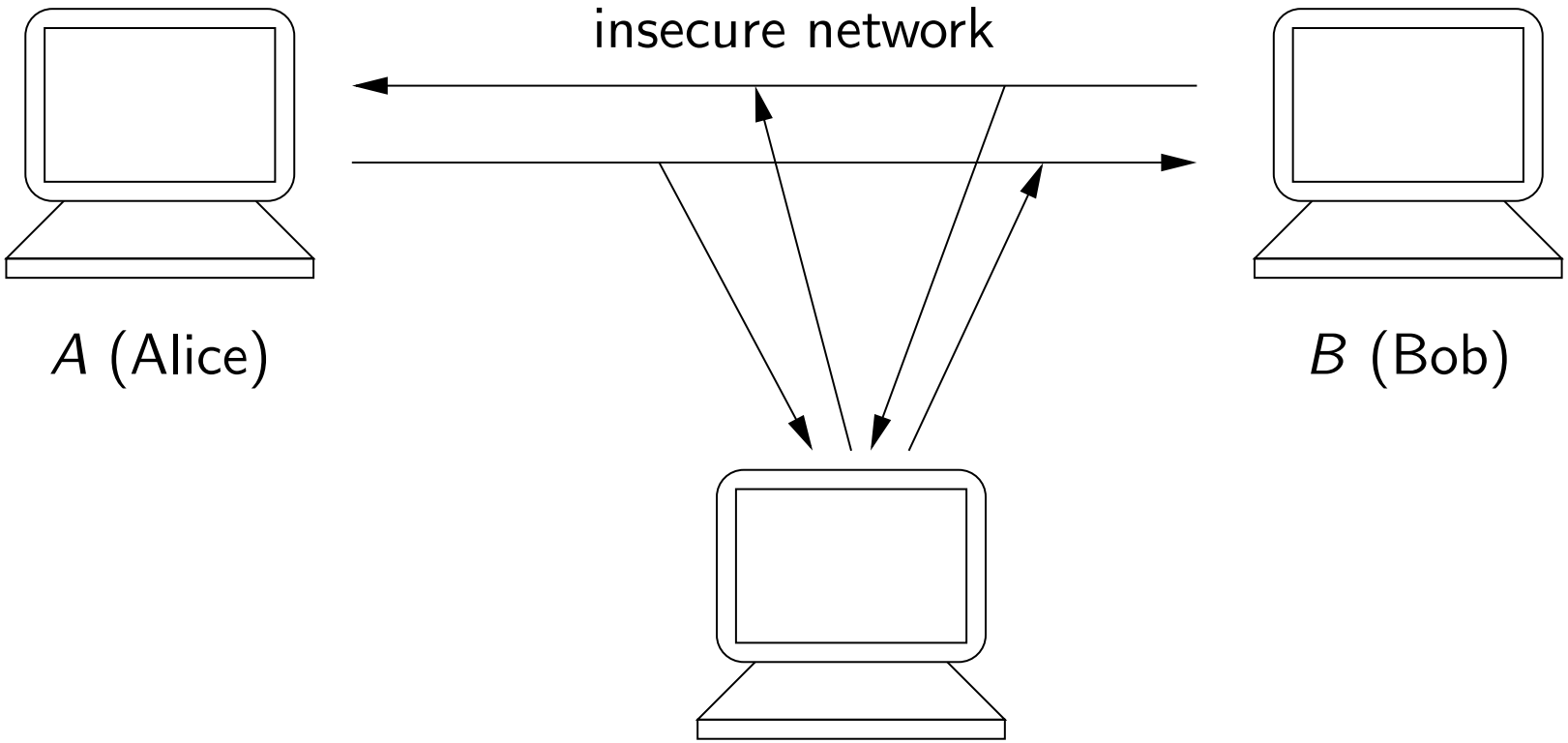
INRIA, École Normale Supérieure, CNRS
Bruno.Blanchet@ens.fr

December 2, 2010

Outline

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

Communications over an insecure network



A talks to B on an insecure network

$\Rightarrow$ need for cryptography in order to make communications secure
for instance, encrypt messages to preserve secrets.

Some cryptographic primitives

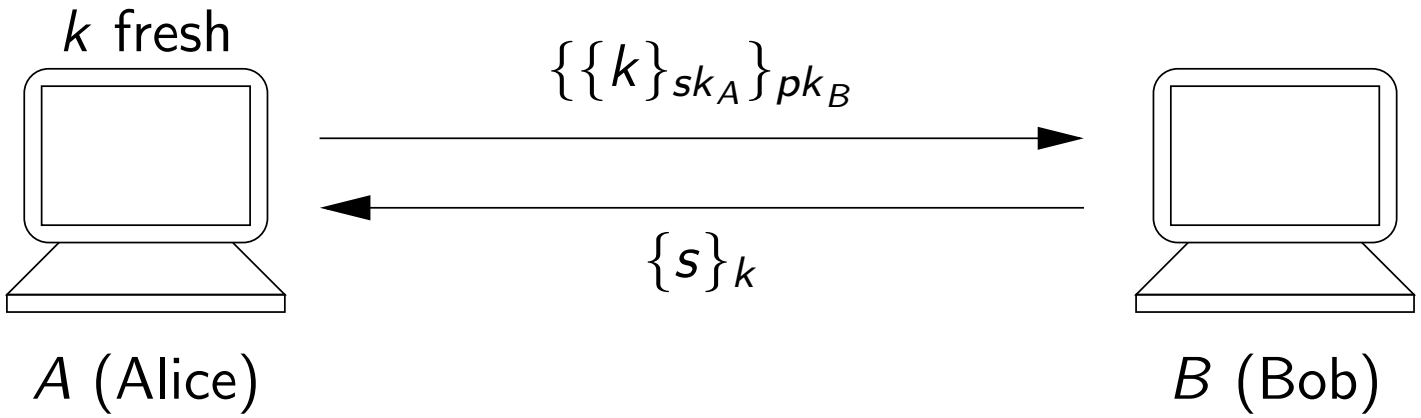
- Encryption: $\{m\}_k$ is the encryption of message m under key k. When you have the decryption key, you can get m from $\{m\}_k$.

  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

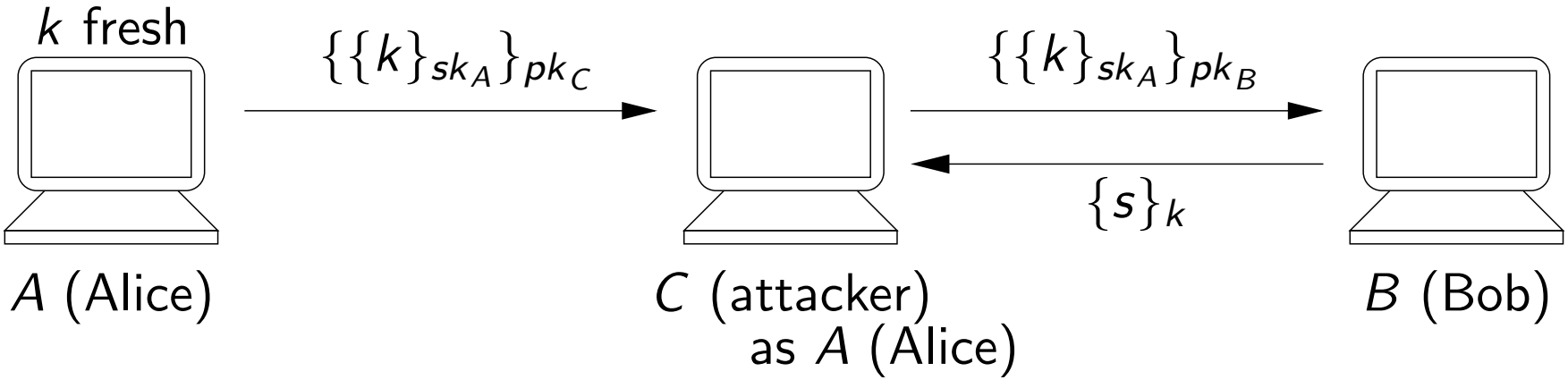
Denning-Sacco key distribution protocol [Denning, Sacco, 1981] (simplified)



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.

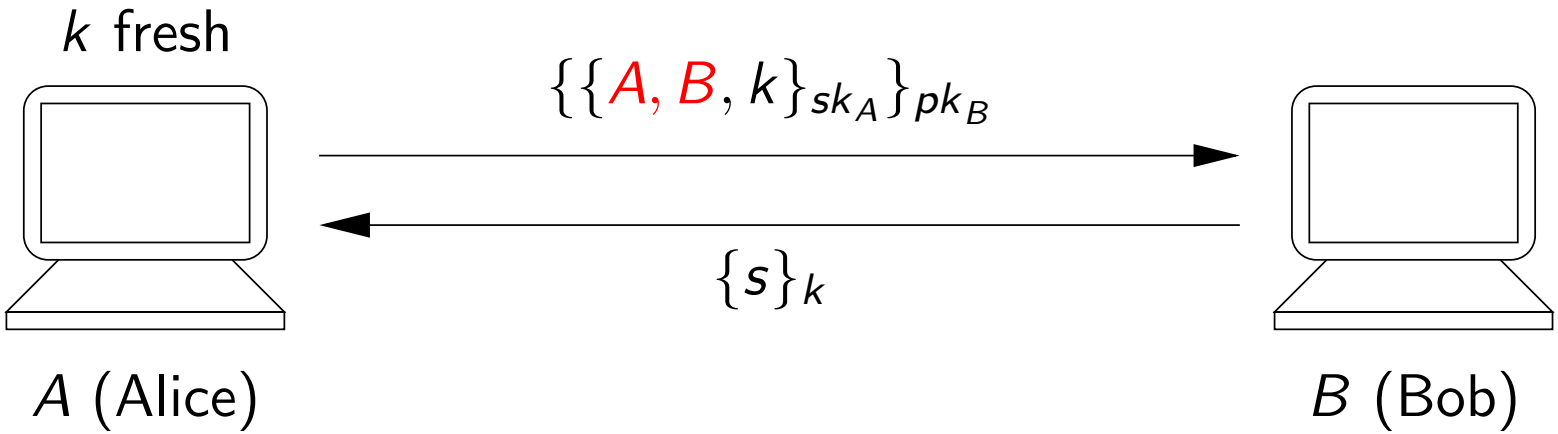
The attack

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.

Examples

Many protocols exist, for various goals:

- secure channels: SSH (Secure SHell);
  SSL (Secure Socket Layer), renamed TLS (Transport Layer Security);
  IPsec
- wifi (WEP/WPA/WPA2)
- banking
- mobile phones
- e-voting
- contract signing
- certified email
- ...

Why verify security protocols ?

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

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

Models 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

Models 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.

This model makes automatic proofs relatively easy.

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.

Protocol verification in the formal model

Security 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.

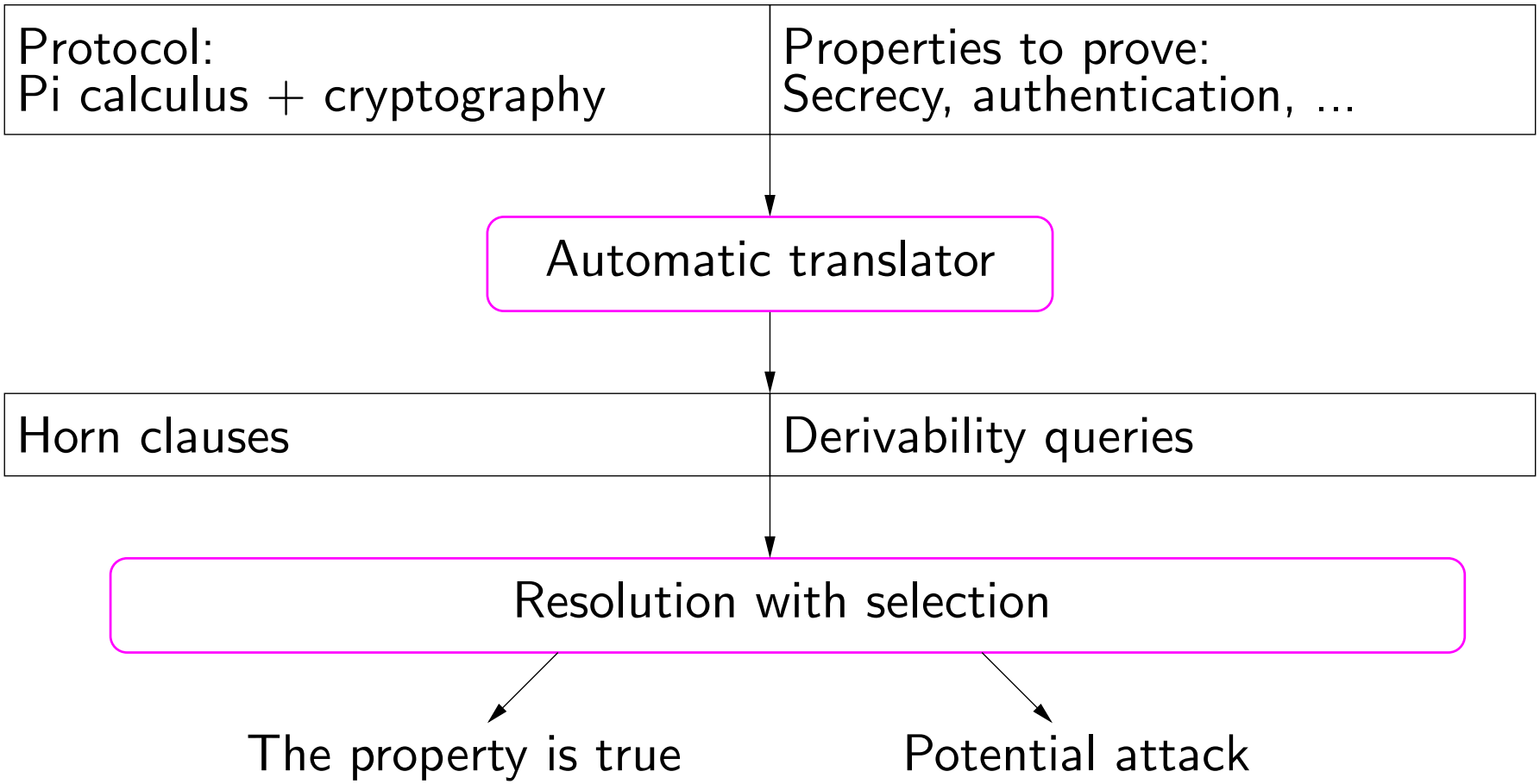
Solutions to undecidability

To solve an undecidable problem, we can

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

In ProVerif, we do the first two, using a very precise abstraction by Horn clauses.

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.
- Very precise: false attacks extremely rare.
  Main approximation: number of repetitions of actions ignored.
- Terminates in most cases; always terminates on a large class, tagged protocols.
- Works for unbounded number of sessions and message space.
- Handles a wide range of cryptographic primitives, defined by rewrite rules or equations.
- Handles various security properties: secrecy, correspondences (including authentication), some equivalences (that the adversary cannot distinguish between two processes).

The Horn clause representation

The first encoding of protocols in Horn clauses was given by Weidenbach (1999).

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.

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 sencrypt(m, k)

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

- 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 sdecrypt(sencrypt$(m, k), k) \rightarrow m$

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

We represent in the same way public-key encryption, signatures, hash functions, ...

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 $\text{sencrypt}(\text{s}, y)$:

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

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

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.

We determine whether a given fact can be derived from the clauses using resolution with free selection.

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.)

Results

- 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)
- Used by others in at least 27 papers, including:
  - Tools that use ProVerif:
    - Web service verifier TulaFale (Microsoft Research)
    - Verification of F# implementations, including TLS (Microsoft Research and MSR-INRIA)
    - Spi2Java includes verification by ProVerif via Spi2ProVerif (Pozza et al)
  - E-voting protocols (Kremer and Ryan, Backes et al)
  - Zero-knowledge protocols, DAA (Backes et al)
  - Dolev-Yao proof implying computational security (Canetti and Herzog)
- Interest from DGA-MI.

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, Diffie-Hellman, ...
- 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. 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.)

Games are formalized in a process calculus.

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.
- Checking function $check(m, k, t)$ 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):

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

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

MACs: intuitive implementation

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- if $k \xleftarrow{R} mkgen$ is used only for generating and checking MACs, the check $check(m, k, t)$ can succeed only if the *mac* of m has been computed by the protocol.
- so we can replace a check with a lookup: if the call to *mac* is $mac(x, k)$, we replace $check(m, k, t)$ with

  if m is equal to some $x \wedge check(m, k, t)$ **then true else false**

MACs: intuitive implementation (2)

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

$$!^{i \leq n} \ldots \textbf{let } x[i] : \mathit{bitstring} = \ldots \textbf{ in } \ldots \mathit{mac}(x[i], k) \ldots$$

- We can find a value of x by an array lookup:
  if the call to mac is $\mathit{mac}(x, k)$, we replace $\mathit{check}(m, k, t)$ with

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

MACs: formal implementation

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

$$\begin{array}{l}
!^{N''}\,\textbf{new}\ r : mkeyseed; (\\
\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),\\
\quad !^{N'} Ocheck(m : bitstring, t : macstring) := check(m, mkgen(r), t))\\
\approx !^{N''}\,\textbf{new}\ r : mkeyseed; (\\
\quad !^{N} Omac(x : bitstring) := mac'(x, mkgen'(r)),\\
\quad !^{N'} Ocheck(m : bitsting, t : macstring) :=\\
\qquad \textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j]) \wedge (m = x[j]) \wedge\\
\qquad\quad check'(m, mkgen'(r), t)\ \textbf{then true else false})
\end{array}$$

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

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, *Encrypted Key Exchange*), with David Pointcheval.

Starts being used by others:

- Verification of F# implementations, including TLS (Microsoft Research and MSR-INRIA).
- Verification of SSH Transport Layer (Pironti and Sisto)

Conclusion and future work

- The automatic prover ProVerif works in the formal model.
  It is essentially mature; facilitate a wider adoption.
  Extentions to group protocols (Miriam Paiola)
- The automatic prover CryptoVerif works in the computational model.
  Much work still to do on this topic:
  - Improvements to the proof strategy.
  - Handle more equations (associativity, . . .).
  - Make more case studies.
- An important topic for future work is the verification of implementations of protocols.
  Already considered for C (Goubault-Larrecq and Parennes) and F# (Microsoft Research and MSR-INRIA, using ProVerif, CryptoVerif, or a type system as back-end).
  CryptoVerif specifications $\longrightarrow$ implementations (David Cadé)