

Automatic Verification of Security Protocols: Formal Model and Computational Model

Bruno Blanchet

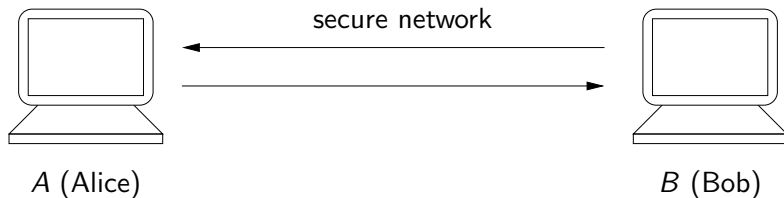
CNRS, École Normale Supérieure, INRIA
Bruno.Blanchet@ens.fr

November 26, 2008

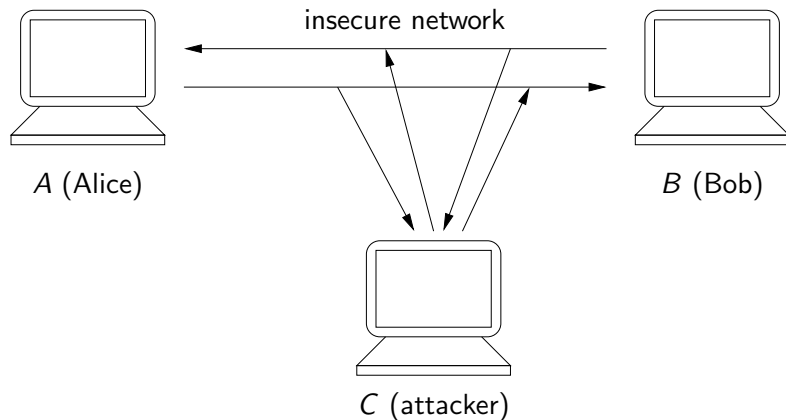
Outline

- 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 a secure network



Communications over an **insecure** network



A talks to B on an insecure network

⇒ 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 $\{m\}_k$.

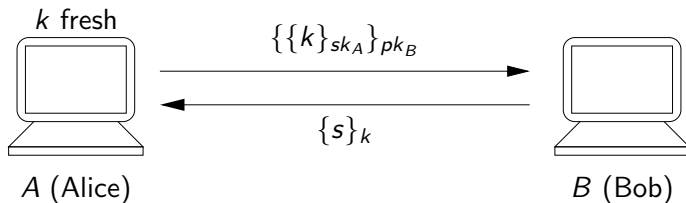
Shared-key encryption: the decryption key is equal to the encryption key.

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

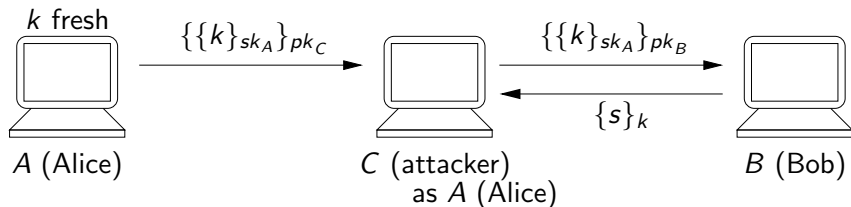
Denning-Sacco key distribution protocol [Denning, Sacco, 1981]
(simplified)



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.

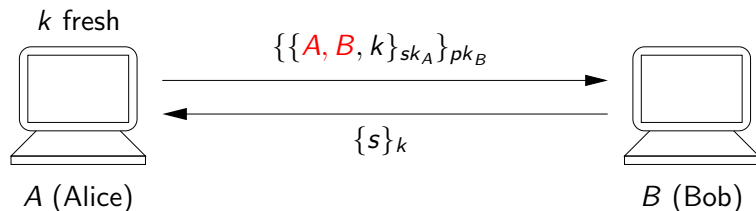
The attack

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
- e-voting
- contract signing
- certified email
- wifi (WEP/WPA/WPA2)
- banking
- mobile phones
- ...

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.
⇒ **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 developed 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 properties: trace and equivalence properties

- Trace properties: properties that can be defined **on a trace**.
In the **formal model**, they hold when they are true **for all traces**.
In the **computational model**, they hold when they are true **except for a set of traces of negligible probability**.
- **Equivalence** (or **indistinguishability**) properties: the attacker cannot distinguish two protocols (**with overwhelming probability**)
Give **compositional** proofs.
Hard to prove in the formal model.

Security properties: secrecy

The attacker cannot obtain certain information on the secrets.

- **Formal model:**
 - (syntactic) secrecy: the attacker cannot obtain the secret (trace property)
 - strong secrecy: the attacker cannot distinguish when the value of the secrecy changes (equivalence property)
- **Computational model:** the attacker can distinguish the secret from a random number only with negligible probability (equivalence property)

Security properties: authentication

If A thinks she talks to B , then B thinks he talks to A , with the same protocol parameters.

- **Formal model:** formalized using **correspondence assertions** of the form “if some event has been executed, then some other events have been executed” (trace property).
- **Computational model:** matching conversations or session identifiers, which essentially require that **the messages exchanged by A and B are the same** up to negligible probability (trace property).

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**.

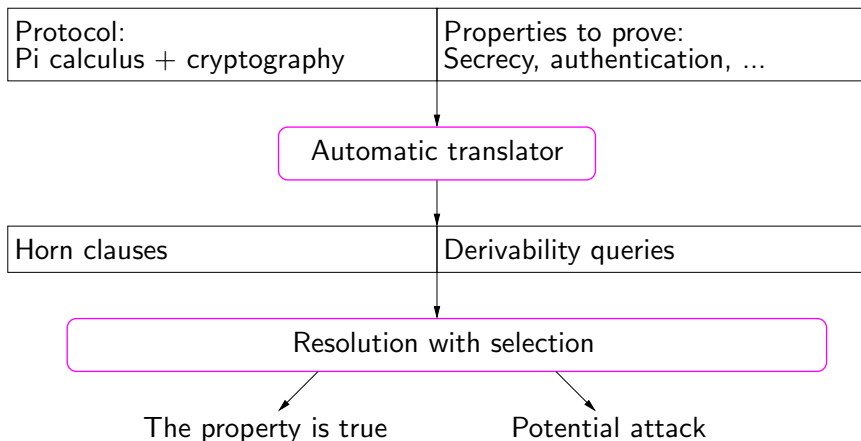
Solutions to undecidability

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



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.
- 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, authentication, some equivalences.

Syntax of the process calculus

Pi calculus + cryptographic primitives

$M, N ::=$

x, y, z, \dots

a, b, c, s, \dots

$f(M_1, \dots, M_n)$

terms

variable

name

constructor application

$P, Q ::=$

$\overline{M}\langle N \rangle.P$

$M(x).P$

let $x = g(M_1, \dots, M_n)$ **in** P **else** Q

if $M = N$ **then** P **else** Q

$0 \quad P \mid Q \quad !P \quad (\nu a)P$

processes

output

input

destructor application

conditional

Constructors and destructors

Two kinds of operations:

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

Example

Shared-key encryption $\text{sencrypt}(M, N)$.

- **Destructors** g manipulate terms
let $x = g(M_1, \dots, M_n)$ **in** P **else** Q
Destructors are defined by rewrite rules $g(M_1, \dots, M_n) \rightarrow M$.

Example

Decryption $\text{sdecrypt}(M', N)$: $\text{sdecrypt}(\text{sencrypt}(m, k), k) \rightarrow m$.

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

Example: The Denning-Sacco protocol (simplified)

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

Message 2. $B \rightarrow A : \{s\}_k$

$(\nu sk_A)(\nu sk_B) \mathbf{let} \ pk_A = \mathbf{pk}(sk_A) \ \mathbf{in} \ \mathbf{let} \ pk_B = \mathbf{pk}(sk_B) \ \mathbf{in}$
 $\bar{c}\langle pk_A \rangle . \bar{c}\langle pk_B \rangle .$

(A) $\quad ! c(x_{pk_B}) . (\nu k) \bar{c}\langle \mathbf{pencrypt}(\mathbf{sign}(k, sk_A), x_{pk_B}) \rangle .$
 $\quad \quad c(x) . \mathbf{let} \ s = \mathbf{sdecrypt}(x, k) \ \mathbf{in} \ 0$

(B) $\quad | \quad ! c(y) . \mathbf{let} \ y' = \mathbf{pdecrypt}(y, sk_B) \ \mathbf{in}$
 $\quad \quad \mathbf{let} \ k = \mathbf{checksign}(y', pk_A) \ \mathbf{in} \ \bar{c}\langle \mathbf{sencrypt}(s, k) \rangle$

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.

Processes are **automatically translated** into Horn clauses (joint work with Martín Abadi).

Coding of primitives

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

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

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

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

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

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

Coding of a protocol

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

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

Example

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

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

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

Proof of secrecy

Theorem (Secrecy)

If $\text{attacker}(M)$ *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 $\text{attacker}(M)$ is derivable, it finds a derivation.)

⇒ The protocol verifier is **sound**

(If it proves secrecy, then secrecy is true.)

Resolution with free selection

$$\frac{R = H \rightarrow F \quad R' = F'_1 \wedge H' \rightarrow F'}{\sigma H \wedge \sigma H' \rightarrow \sigma F'}$$

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.*

Other security properties

- **Correspondence assertions:**

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.

Termination (joint work with Andreas Podelski)

The technique does not always terminate,
but **terminates on tagged protocols**.

Each cryptographic primitive application is identified by a constant **tag**:

$$\text{sencrypt}((c_0, M), K)$$

- Simple **syntactic** annotation.
- Retains the **intended behavior** of the protocol.
- Prevents **type flaw attacks**.
- Good design practice. Used in real protocols (SSH).

Sound approximations

- Main approximation = repetitions of actions are ignored: the clauses can be applied any number of times.

This is the only approximation with respect to a linear logic (or multiset rewriting) model.
- With respect to the process calculus, there is another approximation: in $\overline{M}\langle N \rangle.P$, the Horn clause model considers that P can always be executed.

These approximations can cause (rare) false attacks.

We have built an algorithm that reconstructs attacks from derivations from Horn clauses, when the derivation corresponds to an attack (with Xavier Allamigeon).

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:
 - Web service verifier TulaFale (Microsoft Research)
 - Verification of F# implementations (Microsoft Research)
 - TLS (Microsoft Research and MSR-INRIA)
 - Certified email web service (Lux et al.)
 - E-voting protocols (Kremer and Ryan, Backes et al)
 - Certified mailing lists (Khurama and Hahm)
 - Routing for ad-hoc networks (Godseken)
 - Zero-knowledge protocols, DAA (Backes et al)
 - Extension to XOR (Küsters and Truderung)
 - Dolev-Yao proof implying computational security (Canetti and Herzog)

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), 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.)

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

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

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

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

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

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' .

$$Q_A = !^{i \leq n} c_A(); \mathbf{new} \ x'_k[i] : \mathit{key}; \mathbf{new} \ x''_r[i] : \mathit{coins};$$

$$\mathbf{let} \ x_m[i] : \mathit{bitstring} = \mathit{enc}(k2b(x'_k[i]), x_k, x''_r[i]) \mathbf{in}$$

$$\overline{c}_A \langle x_m[i], \mathit{mac}(x_m[i], x_{mk}) \rangle$$

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** $G_0 \approx G_1 \approx \dots \approx G_m$, 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
- so when checking a MAC with $check(m, k, t)$ 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 check with an array lookup:
if the call to mac is $mac(x, k)$, we replace $check(m, k, t)$ with

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

MACs: formal implementation

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

$$!^{N''} \mathbf{new} \ r : \text{mkeyseed}; ($$

$$\quad !^N (x : \text{bitstring}) \rightarrow \text{mac}(x, \text{mkgen}(r)),$$

$$\quad !^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow \text{check}(m, \text{mkgen}(r), t))$$

$$\approx !^{N''} \mathbf{new} \ r : \text{mkeyseed}; ($$

$$\quad !^N (x : \text{bitstring}) \rightarrow \text{mac}'(x, \text{mkgen}'(r)),$$

$$\quad !^{N'} (m : \text{bitsting}, t : \text{macstring}) \rightarrow$$

$$\quad \mathbf{find} \ j \leq N \ \mathbf{suchthat} \ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge$$

$$\quad \quad \text{check}'(m, \text{mkgen}'(r), t) \ \mathbf{then} \ \mathbf{true} \ \mathbf{else} \ \mathbf{false})$$

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).

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; improve its documentation and interface (contract with CELAR).
- 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 (Diffie-Hellman).
 - 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).