Automatic Verification of Security Protocols: ProVerif and CryptoVerif

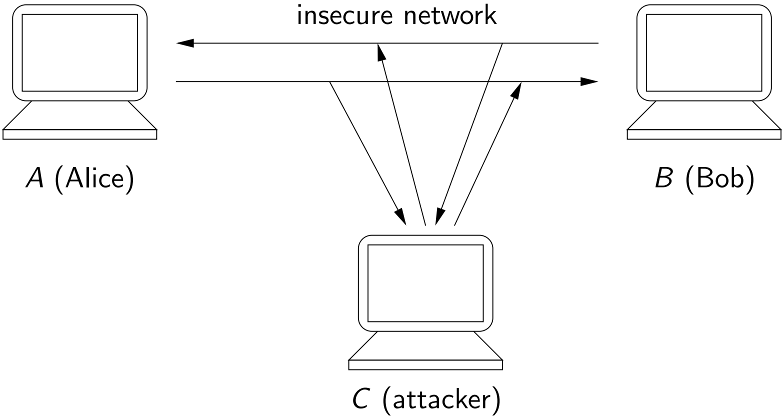
Bruno Blanchet

Google

On leave from Inria, Paris
bruno.blanchet@inria.fr

May 2016

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.

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 ?

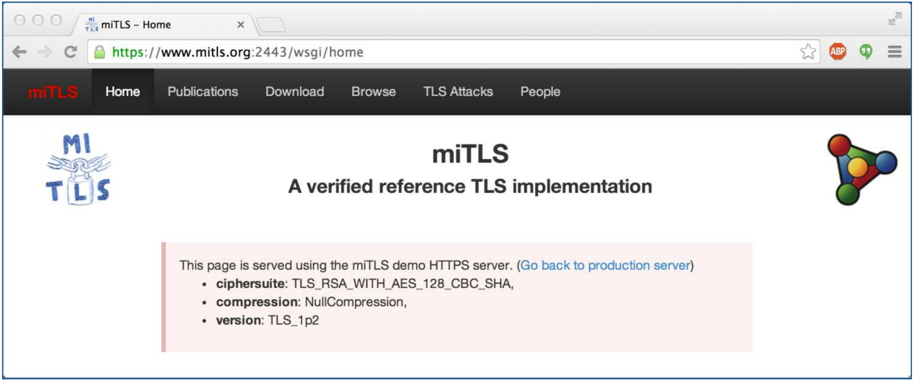
The verification of security protocols has been and is still a very active research area.

- 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.

Attacks against TLS

miTLS, `http://www.mitls.org/`

- Formally verified reference implementation of TLS 1.2 in F7/F* (working towards TLS 1.3)
- Written from scratch focusing on verification



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 symbolic model

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

- Cryptographic primitives are blackboxes. sencrypt
- Messages are terms on these primitives. $\mathrm{sencrypt}(Hello, k)$
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption
  - So the definitions of primitives specify what the attacker can do.
    One can add equations between primitives.
    Hypothesis: 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.

- Messages are bitstrings. 01100100
- Cryptographic primitives are functions on bitstrings.

  $\mathrm{sencrypt}(011, 100100) = 111$
- The attacker is any probabilistic polynomial-time Turing machine.
  - The security assumptions on primitives specify what the attacker cannot do.

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

Models of protocols: side channels

The computational model is still just a model, which does not exactly match reality.

In particular, it ignores side channels:

- timing
- power consumption
- noise
- physical attacks against smart cards

which can give additional information.

Verifying protocols in the symbolic model

Main idea (for most verifiers):

- Compute the knowledge of the attacker.

Difficulty: security protocols are infinite state.

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

Verifying protocols in the symbolic model

Solutions:

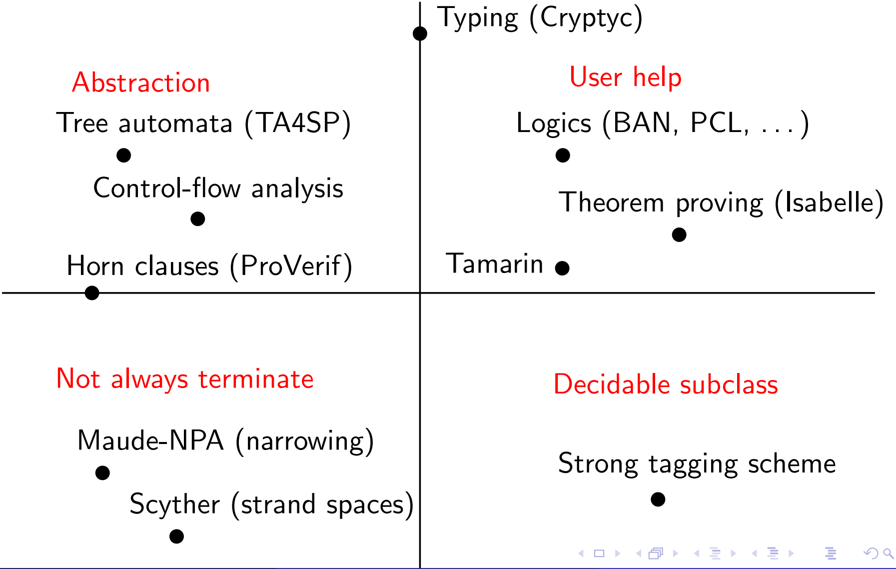
- Bound the state space arbitrarily:
  exhaustive exploration (model-checking: FDR, SATMC, ...);
  find attacks but not prove security.
- Bound the number of sessions: insecurity is NP-complete (with reasonable assumptions).
  OFMC, Cl-AtSe
- Unbounded case:
  the problem is undecidable.

Solutions to undecidability

To solve an undecidable problem, we can

- Use approximations, abstraction.
- Not always terminate.
- Rely on user interaction or annotations.
- Consider a decidable subclass.

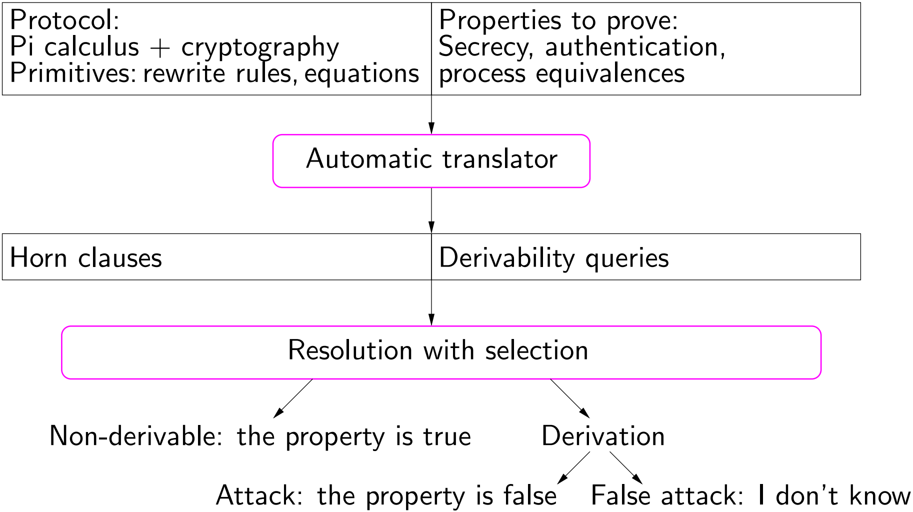
Solutions to undecidability



ProVerif, http://proverif.inria.fr

- Symbolic security protocol verifier.
- Fully automatic.
- 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.
- Does not always terminate and is not complete. In practice:
  - Efficient: small examples verified in less than 0.1 s; complex ones from a few minutes to hours.
  - Very precise: no false attack in 19 protocols of the literature tested for secrecy and authentication.

ProVerif


Protocol:
Pi calculus + cryptography
Primitives: rewrite rules, equations
Properties to prove:
Secrecy, authentication,
process equivalences
Automatic translator
Horn clauses
Derivability queries
Resolution with selection
Non-derivable: the property is true
Derivation
Attack: the property is false
False attack: I don't know

Syntax of the process calculus

Pi calculus + cryptographic primitives

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

Constructors and destructors

Two kinds of operations:

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

Example

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

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

Example

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

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

Example: The Denning-Sacco protocol (simplified)

$$\begin{array}{llll} \text{Message 1.} & A \rightarrow B: & \{\{k\}_{sk_A}\}_{pk_B} & k \text{ fresh} \\ \text{Message 2.} & B \rightarrow A: & \{s\}_k & \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\textbf{new } sk_A; \textbf{new } sk_B; \textbf{let } pk_A = \mathrm{pk}(sk_A) \textbf{ in let } pk_B = \mathrm{pk}(sk_B) \textbf{ in}} \\ & \multicolumn{2}{l}{\textbf{out}(c, pk_A); \textbf{out}(c, pk_B);} \\ (A) & & !\,\textbf{in}(c, x\_pk_B); \textbf{new } k; \textbf{out}(c, \mathrm{pencrypt}(\mathrm{sign}(k, sk_A), x\_pk_B)). \\ & & \quad \textbf{in}(c, x); \textbf{let } s = \mathrm{sdecrypt}(x, k) \textbf{ in } 0 \\ (B) & \mid & !\,\textbf{in}(c, y); \textbf{let } y' = \mathrm{pdecrypt}(y, sk_B) \textbf{ in} \\ & & \quad \textbf{let } k = \mathrm{checksign}(y', pk_A) \textbf{ in out}(c, \mathrm{sencrypt}(\mathrm{s}, k)) \end{array}$$

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:

$\mathsf{attacker}(M)$ *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 $f(M_1, \dots, M_n)$
  attacker$(x_1) \wedge \dots \wedge$ attacker$(x_n) \rightarrow$ attacker$(f(x_1, \dots, x_n))$

Example: Shared-key encryption sencrypt(m, k)

attacker$(m) \wedge$ attacker$(k) \rightarrow$ attacker(sencrypt(m, k))

- Destructors $g(M_1, \dots, M_n) \rightarrow M$
  attacker$(M_1) \wedge \dots \wedge$ attacker$(M_n) \rightarrow$ attacker(M)

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

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

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

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.

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

Other security properties

- Correspondence assertions: (authentication)
  If an event has been executed, then some other events must have been executed.
- Process equivalences: the adversary cannot distinguish between two processes.
  - Strong secrecy: the adversary cannot see when the value of the secret changes.
  - 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.

Demo

Demo

Denning-Sacco example

Applications

1. Case studies:
    - 19 protocols of the literature
    - Certified email (with Martín Abadi)
    - JFK (with Martín Abadi and Cédric Fournet)
    - Plutus (with Avik Chaudhuri)
    - Avionic protocols (ARINC 823)

   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)
    - ...
2. Extensions
3. ProVerif as back-end

Applications

1. Case studies
2. Extensions:
   - Extensions to XOR and Diffie-Hellman (Küsters and Truderung), to bilinear pairings (Pankova and Laud)
   - StatVerif: extension to mutable state (Arapinis et al)
   - Set-Pi: extension to sets with revocation (Bruni et al)
3. ProVerif as back-end

Applications

1. Case studies
2. Extensions
3. ProVerif as back-end:
    - TulaFale: Web service verifier (Bhargavan et al)
    - FS2PV: F# to ProVerif, applied to TLS and TPM (Bhargavan et al)
    - JavaSpi: Java to ProVerif (Avalle et al)
    - Web-spi: web security mechanisms (Bansal et al)

Linking the symbolic and the computational models

- Computational soundness theorems:

  Secure in the symbolic model $\Rightarrow$ secure in the computational model

  modulo additional assumptions.

  Approach pioneered by Abadi & Rogaway [2000]; many papers since then.

Linking the symbolic and the computational models: application

- Indirect approach to automating computational proofs:

$$\begin{array}{ccc}
\text{1. Automatic symbolic protocol verifier} & & \\
\downarrow & \text{2. Computational soundness} & \\
\text{proof in the symbolic model} & \longrightarrow & \text{proof in the computational model}
\end{array}$$

Advantages and limitations

- + symbolic proofs easier to automate
- + reuse of existing symbolic verifiers
- – additional hypotheses:
  - – strong cryptographic primitives
  - – length-hiding encryption or modify the symbolic model
  - – honest keys [but see Comon-Lundh et al, POST 2012]
  - – no key cycles
- Going through the symbolic model is a detour

- An attempt to solve these problems:
  symbolic model in which we specify what the attacker cannot do
  [Bana & Comon-Lundh, POST 2012]

Direct computational proofs

Following Shoup and Bellare&Rogaway, the proof is typically 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.)

Game 0	$\longleftrightarrow$	Game 1	$\longleftrightarrow$		$\longleftrightarrow$	Game n
Protocol to prove	p_1 negligible		p_2 negligible	$\cdots$	p_n negligible	Property obvious

Mechanizing proofs by sequences of games

CertiCrypt, `http://certicrypt.gforge.inria.fr/`

- Machine-checked cryptographic proofs in Coq
- Interesting case studies, *e.g.* OAEP
- Good for proving primitives: can prove complex mathematical theorems
- Requires much human effort

EasyCrypt, `https://www.easycrypt.info/trac/`:

- Successor of CertiCrypt
- Less human effort: give games and hints on how to prove indistinguishability
- Relies on SMT solvers

Idea also followed by Nowak et al.

CryptoVerif, `http://cryptoverif.inria.fr`

- Computational security protocol verifier.
- 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).
- Has automatic and manually guided modes.
- Can generate OCaml implementations (joint work with David Cadé).

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

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

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

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

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

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'.

$$\begin{aligned} Q_A = {} & !^{i \leq n} \mathbf{in}(c_A, ()); \mathbf{new}\ x'_k[i] : key; \mathbf{new}\ x''_r[i] : coins; \\ & \mathbf{let}\ x_m[i] : bitstring = enc(k2b(x'_k[i]), x_k, x''_r[i])\ \mathbf{in} \\ & \mathbf{out}(c_A, x_m[i], mac(x_m[i], x_{mk})) \end{aligned}$$

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.
- 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):

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

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 $k \xleftarrow{R} mkgen$ 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

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

MACs: formal implementation

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

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

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.

Applications

- 16 "Dolev-Yao style" protocols that we study in the computational model. CryptoVerif proves all correct properties except in one case.
- Full domain hash signature (with David Pointcheval)
  Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay)
- OEKE (variant of Encrypted Key Exchange)
- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA)
- SSH Transport Layer Protocol (with David Cadé)
- Avionic protocols (ARINC 823, ICAO9880 3rd edition)

Conclusion and future work

- The automatic prover ProVerif works in the symbolic model.
  It is essentially mature; improve its documentation and interface.
- The automatic prover CryptoVerif works in the computational model.
  Much work still to do:
  - Improvements to the game transformations and the proof strategy.
  - Handle more cryptographic primitives (stateful encryption, . . .)
  - Extend the input language (loops, mutable variables, . . .)
  - Make more case studies.