The automatic security protocol verifier ProVerif

Bruno Blanchet

CNRS, École Normale Supérieure, INRIA, Paris

June 2010

Introduction

- Many techniques exist for the verification of cryptographic protocols (theorem proving, model checking, typing, ...)
- We present a technique based on an abstract representation of the protocol by a set of Horn clauses.
- This technique yields fully automatic proofs of protocols for an unbounded number of runs (many clients connecting to a server, for instance) and an unbounded message space.

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).

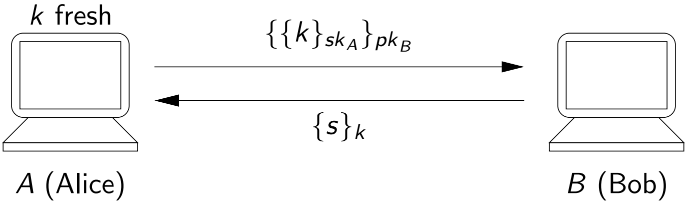
- The cryptographic primitives are blackboxes.
- The messages are terms on these primitives.
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption

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.

Example

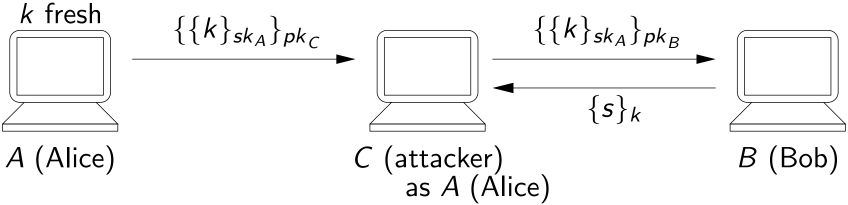
Denning-Sacco key distribution protocol [Denning, Sacco, Comm. ACM, 1981] (simplified)



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.

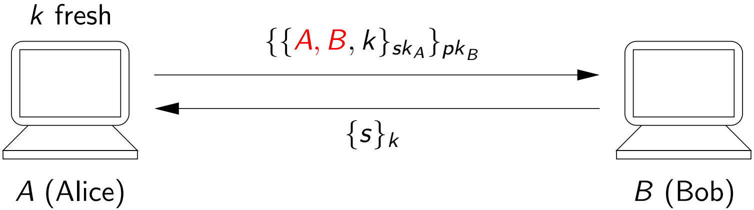
The attack

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.

Practical 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.
- Errors are not detected by testing:
  they appear only in the presence of an adversary.
- Errors can have serious consequences.

Security goals

- Secrecy: The attacker cannot have a message s.
- Authentication: If B thinks he talks to A then A thinks she talks to B.
- Key exchange: establish a shared secret key between A and B.
- Fair contract signing.
- ...

Difficulties of protocol verification

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.

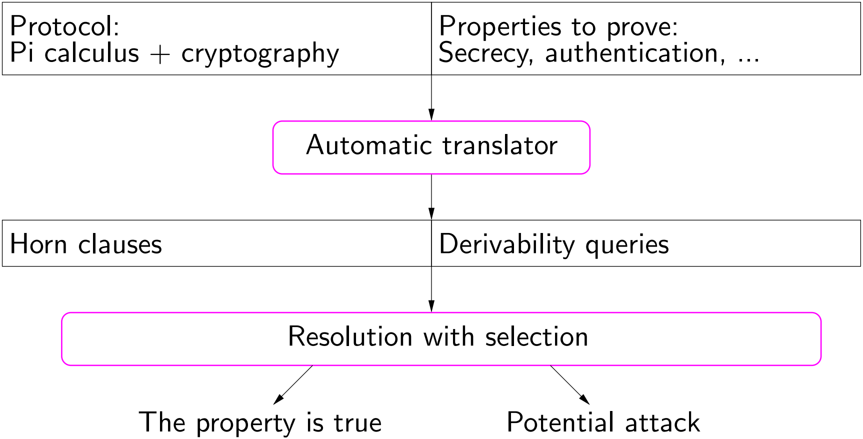
Solutions to undecidability

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.
- Wide variety of cryptographic primitives and security properties.

Definition of cryptographic primitives

Two kinds of operations:

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

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

- Destructors g manipulate terms

  Destructors are defined by rewrite rules $g(M_1, \ldots, M_n) \rightarrow M$.

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

Examples of constructors and destructors

Shared-key encryption:

- Constructor: Shared-key encryption

  fun encrypt(bitstring, key) : bitstring.

- Destructor: Decryption

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

Examples of constructors and destructors (continued)

Probabilistic shared-key encryption:

- Constructor: Shared-key encryption

fun encrypt(bitstring, key, coins) : bitstring.

- Destructor: Decryption

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

To encrypt m under k, one generates fresh random coins r and computes encrypt(m, k, r).

Examples of constructors and destructors (continued)

Signature:

- Constructors:
  - Public key generation:

    fun spk(sskey) : spkey.

  - Signature:

    fun sign(bitstring, sskey) : bitstring.

- Destructors:
  - Signature verification:

    reduc forall m : bitstring, k : sskey; checksign(sign(m, k), spk(k)) $= m$.

  - Message extraction:

    reduc forall m : bitstring, k : sskey; getmess(sign(m, k)) $= m$.

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

Equations

An alternative way of modeling primitives is using equations:

- More powerful but harder to handle.
- The model used in the applied pi calculus [Abadi, Fournet, POPL'01].
- ProVerif handles equations by translating them into rewrite rules.
  - Advantage: can still use fast, syntactic unification.
  - Limitations: e.g., cannot handle associativity.

Equations: shared key-encryption

Symmetric encryption in which one cannot detect whether decryption succeeds.

- Encryption and decryption functions:

  fun encrypt(bitstring, key) : bitstring.
  fun decrypt(bitstring, key) : bitstring.

- Equations:

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

  - The first equation is standard.
  - The second equation is more surprising.
    Without it, we can test whether decrypt(x, y) succeeds by testing

    $$\mathrm{encrypt}(\mathrm{decrypt}(x, y), y) = x$$

  - encrypt and decrypt are two inverse bijections (cf. block ciphers).

Equations: Diffie-Hellman

Modular exponentiation on a group G with generator g.

const g : G.
fun exp(G, Z) : G.

$$\begin{array}{llll} \text{Message 1.} & A \rightarrow B: & g^a & a \text{ fresh} \\ \text{Message 2.} & B \rightarrow A: & g^b & b \text{ fresh} \end{array}$$

A computes $k = (g^b)^a$, B computes $k = (g^a)^b$.
The exponentiation is such that these quantities are equal.

equation forall y : Z, z : Z; exp(exp(g, y), z) = exp(exp(g, z), y).

For a richer model of Diffie-Hellman, see Ralf Küsters's talk.

Syntax of the process calculus

Pi calculus + cryptographic primitives

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

Example: The Denning-Sacco protocol

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

$$\begin{array}{lll}
 & & \textbf{new } sk_A : \mathrm{sskey}; \textbf{let } pk_A = \mathrm{spk}(sk_A) \textbf{ in} \\
 & & \textbf{new } sk_B : \mathrm{skey}; \textbf{let } pk_B = \mathrm{pk}(sk_B) \textbf{ in} \\
 & & \textbf{out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & & \quad !\, \textbf{in}(c, x_pk_B : \mathrm{pkey}); \\
 & & \quad\ \ \textbf{new } k : \mathrm{key}; \textbf{out}(c, \mathrm{pencrypt}(\mathrm{sign}(\mathrm{k2b}(k), sk_A), x_pk_B)); \\
 & & \quad\ \ \textbf{in}(c, x : \mathrm{bitstring}); \textbf{let } s = \mathrm{decrypt}(x, k) \textbf{ in } 0 \\
(B) & | & \quad !\, \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let } y' = \mathrm{pdecrypt}(y, sk_B) \textbf{ in} \\
 & & \quad\ \ \textbf{let } \mathrm{k2b}(k) = \mathrm{checksign}(y', pk_A) \textbf{ in } \textbf{out}(c, \mathrm{encrypt}(\mathrm{s}, k))
\end{array}$$

Security properties: secrecy

Secrecy:

- The attacker cannot obtain the secret s

```
query attacker(s).
```

Security properties: correspondences

Correspondence assertions:

- ProVerif can also prove properties of the form:
  - If an event e has happened, then some other events $e_1, \ldots, e_n$ must have happened.
  - For each execution of event e, *distinct* events $e_1, \ldots, e_n$ have been executed.
- These properties formalize in particular authentication.

Security properties: equivalences

Process equivalences:

- $P \approx Q$ when the adversary cannot distinguish P from Q.
- Process equivalences can formalize various security properties.
  Example: P implements an ideal specification Q.
- ProVerif can check:
  - Strong secrecy: $P(x) \approx P(y)$ for all x, y.
  - Equivalences between processes that differ only by terms they contain
    (joint work with Martín Abadi and Cédric Fournet)
    Includes resistance to guessing attacks

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 $M_1, \dots, M_n$ and sends the message M,

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

Example

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

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

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

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.)

Determining derivability

The natural tool for determining derivability of a fact from clauses is resolution.
Basic SLD-resolution (as in Prolog) would never terminate:

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

yields a loop.

Resolution with free selection

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

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.

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:

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

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

Other possible variants

- Ordered resolution with selection [Weidenbach, 1999].
- Ordered resolution with factorization and splitting [Comon, Cortier, 2003]
  Terminates on clauses with at most one variable.
  Protocols which blindly copy at most one term.

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 are other approximations: e.g., in $\mathbf{out}(M, N).P$, the Horn clause model considers that P can always be executed.

These approximations help for efficient verification

- with infinite state spaces,
- with any number of protocol runs.

They can cause (rare) false attacks.

Reconstruction of attacks (joint work with Xavier Allamigeon)

- In general, a derivation of attacker(M) manifests an attack against the secrecy of M.
- We can reconstruct such an attack from the derivation: we explore the traces that correspond to the derivation.
- The attack reconstruction fails when the derivation does not correspond to an attack (false attack).

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)
  - Certified mailing lists (Khurama and Hahm)
  - Routing for ad-hoc networks (Godseken)
  - Bluetooth device pairing (Chang and Shmatikov)
  - 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).
  Used for verifying a certified email web service (Lux et al.)
- 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.

Computational soundness

Provide a proof in the computational model via

- a proof in the formal model (ProVerif or others) and
- a computational soundness theorem.

Examples of this approach:

- Using ProVerif,
  - Canetti and Herzog (for some public-key protocols),
  - Canetti and Gajek (Diffie-Hellman),
  - Abadi, B., and Comon (equivalence for Woo-Lam shared-key).
- AVISPA includes a module for computational soundness.

ProVerif has strong properties for this application:

- one of the rare tools that prove process equivalences,
- unbounded number of sessions.

but does not verify assumptions of computational soundness theorems.

Conclusion

Strong points of ProVerif:

- Automatic.
- Proves protocols for an unbounded number of sessions and unbounded message space.
- Handles a wide variety of primitives defined rewrite rules or equations.
- Proves a wide variety of security properties (secrecy, correspondences, equivalences).
- Precise and efficient in practice.

Limitations:

- Risk of non-termination.
- Risk of false attacks.
- Still limited on the equations it supports and the equivalences it can prove.

State of the art

- Automatic verification in the formal model: already advanced.
  Still some challenges:
  - primitives with complex equations
  - more general equivalences
  - group protocols
  - timestamps
  - state changes
  - ...
- Grand challenges:
  - proof of implementations
  - proofs in the computational model

  Some good work done, but these problems are far from being solved.