Automatic Proof of Strong Secrecy for Security Protocols

Bruno Blanchet
CNRS, Département d'Informatique,
École Normale Supérieure
and Max-Planck-Institut für Informatik

May 2004

Introduction

Verification of security protocols by formal methods.
"Dolev-Yao" model:
– perfect cryptography (one can decrypt only if one has the key)
– the adversary can receive, compute, and send messages.

Two notions of secrecy:

- Standard secrecy: an adversary cannot obtain exactly the value of the secret.
- Strong secrecy: when the value of the secret changes, the adversary sees no difference.

  This is similar to the notion of non-interference.

Why strong secrecy?

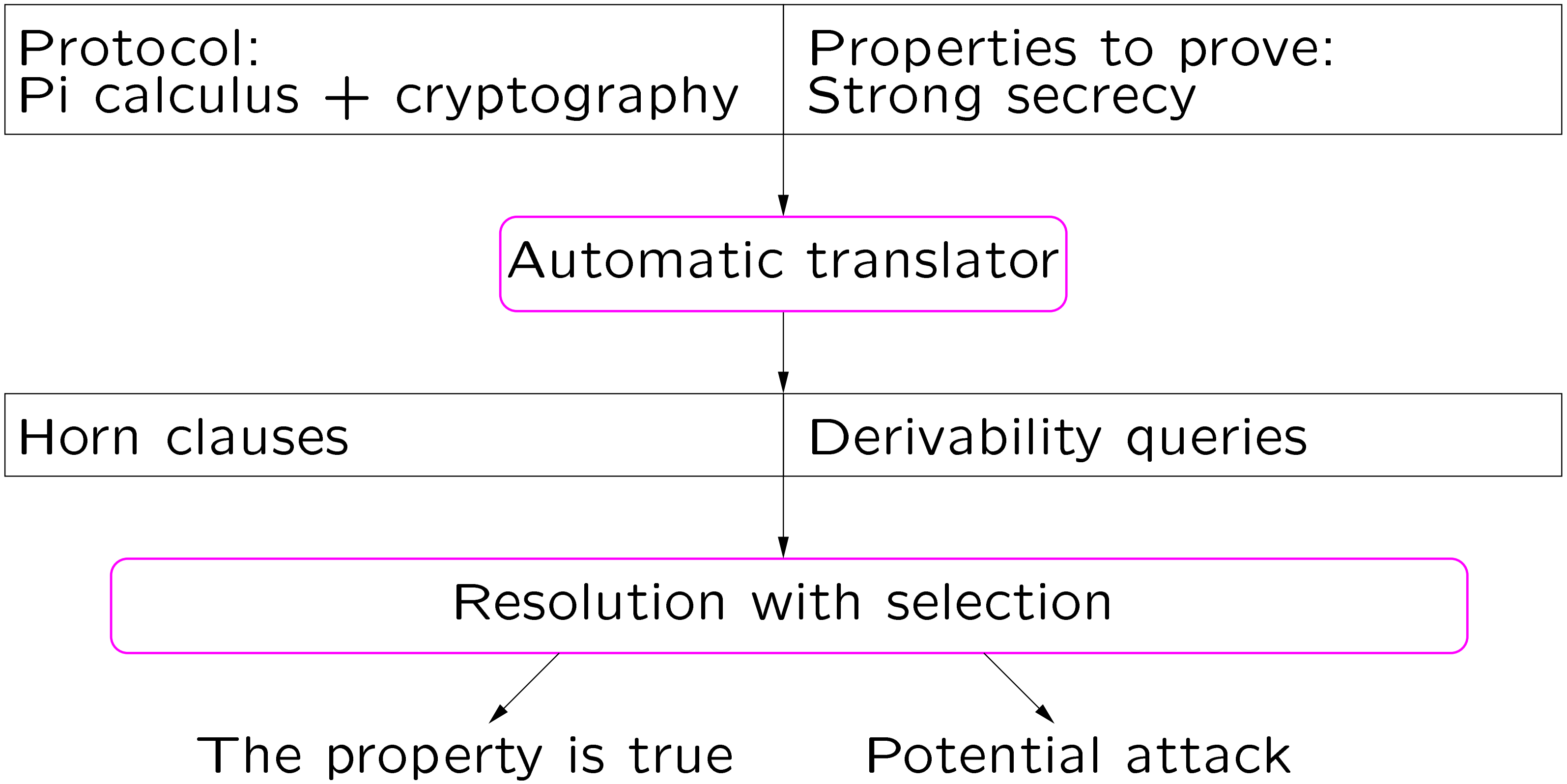
- It detects implicit flows.
  Example: Depending on the value of the secrets, a test in the protocol yields a different result, so different branches of code are executed and the adversary sees the difference.

- It detects flows of partial information.

- It is closer to the computational notion of secrecy (e.g. semantic security of a key).

- It is a process equivalence $\Rightarrow$ nice compositionality properties.
  It can be used for combining manual and automatic proofs.

Overview of the verification method



Main features of the method

- Fully automatic.
- Efficient: small examples verified in less than 0.1 s; complex ones in a few minutes.
- For unbounded number of sessions and message space.
- For a wide range of cryptographic primitives.

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
$\quad \overline{M}\langle N \rangle.P$	output
$\quad M(x).P$	input
$\quad \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$	destructor application
$\quad \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional
$\quad 0 \quad P \mid Q \quad !P \quad (\nu a)P$	

Constructors and destructors

Two kinds of operations:

- Constructors f are used to build terms
  $f(M_1, \ldots, M_n)$
- Destructors g manipulate terms
  $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$
  Destructors are defined by reductions $g(M_1, \ldots, M_n) \rightarrow M$.

Example of constructors and destructors

Probabilistic public-key encryption:

- Constructors: Encryption $pencrypt(M, N, R)$.
  Public key generation $pk(N)$.

  Intended use: $(\nu r)\overline{c}\langle pencrypt(M, pk_A, r)\rangle$.

- Destructors: Decryption $pdecrypt(M', N')$

$$pdecrypt(pencrypt(M, pk(N), R), N) \rightarrow M.$$

We represent in the same way shared-key encryption (probabilistic or deterministic), signatures, hash functions, macs, ...

Example of protocol: simplified corrected Denning-Sacco

Message 1.	$A \rightarrow B:$	$\{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B}$	k fresh
Message 2.	$B \rightarrow A:$	$\{\mathsf{x}\}_k$	

$$(\nu sk_A)(\nu sk_B) let\ \ pk_A = pk(sk_A)\ \ in\ \ let\ \ pk_B = pk(sk_B)\ \ in$$

$$\overline{c}\langle pk_A\rangle \overline{c}\langle pk_B\rangle.$$

$$(A) \qquad !\ c(x\_pk_B).(\nu k)(\nu r)\overline{c}\langle pencrypt(sign((pk_A, x\_pk_B, k), sk_A), x\_pk_B, r)\rangle.$$

$$c(x).let\ \ s = sdecrypt(x, k)\ \ in\ \ 0$$

$$(B) \qquad |\quad !\ c(y).let\ \ y' = pdecrypt(y, sk_B)\ \ in$$

$$let\ \ (= pk_A, = pk_B, k) = checksign(y', pk_A)\ \ in\ \ \overline{c}\langle sencrypt(\mathsf{x}, k)\rangle$$

Formal definition of strong secrecy

Observational equivalence $P \approx Q$ intuitively means that the adversary cannot distinguish P from Q.

The process P_0 preserves the strong secrecy of its free variables if and only if for all closed substitutions σ and σ' of domain $fv(P_0)$,

$$\sigma P_0 \approx \sigma' P_0$$

Key of the proof of strong secrecy

We require a stronger criterion in order to simplify the proof of strong secrecy:

> Each reduction step of the process is independent of the values of the secrets.

This is a sound approximation.

Condition 1: The success or failure of communications is independent of the values of the secrets.

Condition 2: The success or failure of destructor applications is independent of the values of the secrets.

Key of the proof of strong secrecy: formalization

Let P_0' be obtained from P_0 by substituting distinct free names of a set $Secr$ for free variables of P_0.

Let Q be any adversary, i.e. any process such that $fn(Q) \cap Secr = \emptyset$.

Condition 1: $P_0' \mid Q$ does not communicate on a channel in $Secr$.

Otherwise, the adversary would see a communication succeed when the secret is some M it has and it tries to communicate on M.

Condition 2: if $P_0' \mid Q$ executes a destructor application $let\ y = g(M_1, \ldots, M_n)\ in\ Q'\ else\ R'$ and the destructor succeeds for some values of the secrets, then it succeeds for all values of the secrets.

Otherwise, the adversary might see the difference between the execution of Q' for some values of the secrets and of R' for others.

The Horn clause representation

The Horn clauses use the following predicates:

attacker(p)	the attacker may have p
mess(p, p')	the message p' may be sent on channel p
com(p)	there may be a communication on channel p
test$_g(p_1, \ldots, p_n)$	destructor application $g(p_1, \ldots, p_n)$ may be executed
testunif(p, p')	unification test (explained next)
bad	when it is derivable, strong secrecy may be wrong

Processes are automatically translated into Horn clauses.

This translation extends Abadi, Blanchet POPL'02.

The Horn clause representation: testunif

The predicate testunif is used to detect cases when a destructor application succeeds for some values of the secrets and not for others.

p and p' may contain special constants in a set $EVar$.

testunif(p, p') is true when

- p and p' can be made equal by substituting terms for elements of $Secr$ and $EVar$, and
- p and p' cannot be made equal by substituting terms for elements of $EVar$.

The Horn clause representation: testunif (2)

Simple examples:

- $\mathsf{testunif}(p, p)$ is false
  (unification always succeeds)

- $\mathsf{testunif}(\mathsf{x}, p)$ is true (when $\mathsf{x} \in \mathit{Secr}$ and $p \neq \mathsf{x}$)
  (unification succeeds only when x is p)

- $\mathsf{testunif}(f(\ldots), g(\ldots))$ is false
  (unification always fails)

The Horn clause representation: testunif (3)

$\mathsf{testunif}((x,y),(pencrypt(\mathsf{e},pk(\mathsf{e}'),\mathsf{e}''),\mathsf{e}'))$ is true when (x,y) is of the form $(pencrypt(p,pk(p'),p''),p')$ for some values of the secrets, but not for all values of the secrets.

- $\mathsf{testunif}((\mathsf{x},c_0),(pencrypt(\mathsf{e},pk(\mathsf{e}'),\mathsf{e}''),\mathsf{e}'))$ is true.
- $\mathsf{testunif}((pencrypt(c_1,pk(\mathsf{x}),c_2),c_0),(pencrypt(\mathsf{e},pk(\mathsf{e}'),\mathsf{e}''),\mathsf{e}'))$ is true (unification succeeds only when $\mathsf{x}=c_0$).
- $\mathsf{testunif}((sencrypt(\ldots),\mathsf{x}),(pencrypt(\mathsf{e},pk(\mathsf{e}'),\mathsf{e}''),\mathsf{e}'))$ is false (unification fails for all values of the secret).

The Horn clause representation: testunif (4)

testunif$((x, y), (pencrypt(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}'))$ is true when the destructor application $pdecrypt(x, y)$ succeeds for some values of the secrets, but not for all values of the secrets.

testunif$((x, y), (pencrypt(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}'))$ is true when the success or failure of the destructor application $pdecrypt(x, y)$ gives some information on the values of the secrets.

$\Rightarrow$ the predicate testunif can be used to check condition 2.

testunif is not defined by clauses but by specific simplification steps in the solving algorithm.

Clauses to detect attacks

We are going to prove that bad is not derivable from the clauses.

Condition 1: For each x $\in Secr$,

$$\mathsf{com(x)} \Rightarrow \mathsf{bad}$$

If bad is not derivable, then there is no communication on a channel in $Secr$.

Condition 2: For each reduction $g(M_1, \dots, M_n) \to M$,

$$\mathsf{test}_g(x_1, \dots, x_n) \wedge \mathsf{testunif}((x_1, \dots, x_n), (N_1, \dots, N_n)) \Rightarrow \mathsf{bad}$$

where $N_1, \dots, N_n$ are obtained from $M_1, \dots, M_n$ by substituting variables with distinct constants in the set $EVar$.

If bad is not derivable, the success/failure of the destructor application $g(x_1, \dots, x_n)$ is independent from the values of the secrets.

Clauses for the attacker

For each $a \in fn(P_0)$, $\mathsf{attacker}(a[])$

For each constructor f of arity n,

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n))$$

For each reduction $g(M_1, \ldots, M_n) \to M$,

$$\mathsf{attacker}(M_1) \wedge \ldots \wedge \mathsf{attacker}(M_n) \Rightarrow \mathsf{attacker}(M)$$

For each destructor g of arity n,

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{test}_g(x_1, \ldots, x_n)$$

$\mathsf{mess}(x, y) \wedge \mathsf{attacker}(x) \Rightarrow \mathsf{attacker}(y)$

$\mathsf{attacker}(x) \wedge \mathsf{attacker}(y) \Rightarrow \mathsf{mess}(x, y)$

$\mathsf{attacker}(x') \Rightarrow \mathsf{com}(x')$

Example of clauses for cryptographic primitives

Public key probablistic encryption:

Encryption:

$\text{attacker}(x) \wedge \text{attacker}(y) \wedge \text{attacker}(z) \Rightarrow \text{attacker}(pencrypt(x, y, z))$

Public key generation:

$\text{attacker}(x) \Rightarrow \text{attacker}(pk(x))$

Decryption:

$\text{attacker}(pencrypt(x, pk(y), z)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x)$

$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{test}_{pdecrypt}(x, y)$

$\text{test}_{pdecrypt}(x, y) \wedge \text{testunif}((x, y), (pencrypt(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}')) \Rightarrow \text{bad}$

Clauses for the protocol

If a participant A receives $M_1, \ldots, M_n$ on channels $C_1, \ldots, C_n$ respectively, then

– outputs M on channel C:

$$\mathsf{mess}(C_1, M_1) \wedge \ldots \wedge \mathsf{mess}(C_n, M_n) \Rightarrow \mathsf{mess}(C, M)$$

– executes an input or an output on channel C:

$$\mathsf{mess}(C_1, M_1) \wedge \ldots \wedge \mathsf{mess}(C_n, M_n) \Rightarrow \mathsf{com}(C)$$

– executes $let\ x = g(N_1, \ldots, N_n)\ in$:

$$\mathsf{mess}(C_1, M_1) \wedge \ldots \wedge \mathsf{mess}(C_n, M_n) \Rightarrow \mathsf{test}_g(N_1, \ldots, N_n)$$

The formal definition is done by induction on the process.

Example of clauses for the protocol

B tests if the incoming message is $pencrypt(sign((pk_A, pk_B, y), sk_A), pk_B, r)$:

$$\mathsf{mess}(c, x) \Rightarrow \mathsf{test}_{pdecrypt}(x, sk_B)$$
$$\mathsf{mess}(c, pencrypt(x, pk_B, r)) \Rightarrow \mathsf{test}_{checksign}(x, pk_A)$$
$$\ldots$$

or the optimized clause:

$$\mathsf{mess}(c, x) \wedge \mathsf{testunif}(x, pencrypt(sign((pk_A, pk_B, \mathsf{e}), sk_A), pk_B, \mathsf{e}')) \Rightarrow \mathsf{bad}$$

If yes, B replies with $sencrypt(\mathsf{x}, y)$:

$$\mathsf{mess}(c, pencrypt(sign((pk_A, pk_B, y), sk_A), pk_B, r)) \Rightarrow \mathsf{mess}(c, sencrypt(\mathsf{x}, y))$$

Solving algorithm

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) or testunif(p, p') if possible,
– the conclusion otherwise.

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

Solving algorithm: simplification steps for testunif

The simplification of testunif facts has several goals:

- reduce the size of testunif facts, so that they do not grow indefinitely;

- make progress towards knowing the truth value of these facts, by instantiating the variables;

- simplify them out when their truth value is known.

The complete list of simplification steps is in the paper.
An example follows.

Solving algorithm: simplification steps for testunif **(2)**

Instantiation: replace R of the form

$\mathsf{attacker}(x) \wedge \mathsf{testunif}((\ldots, x, \ldots), (\ldots, f(p_1, \ldots, p_n), \ldots)) \wedge H \Rightarrow \mathsf{bad}$

with $R\{f(x_1, \ldots, x_n)/x\}$.

When R derives bad,
either $x \in \mathit{Secr}$: bad is derived from attacker(x) by other clauses;
or $x \notin \mathit{Secr}$: x is an instance of $f(x_1, \ldots, x_n)$ so bad is derived by $R\{f(x_1, \ldots, x_n)/x\}$.

Correctness

Theorem: bad is derivable from the clauses if and only if the solving algorithm generates the clause bad.

If the solving algorithm does not generate the clause bad, we have proved strong secrecy.

Experimental results

The corrected Denning-Sacco protocol preserves the strong secrecy of x.

The protocol

Message 1.	$A \to B:$	$\{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B}$	k fresh
Message 2.	$B \to A:$	$\{\mathsf{x}\}_k$	
Message 2'.	$A \to B:$	$\{\mathsf{x}'\}_k$	

preserves the strong secrecy of x and x′ when shared-key encryption is probabilistic, but not when it is deterministic (the attacker knows whether $\mathsf{x} = \mathsf{x}'$ or not).

It also preserves the strong secrecy of x and x′ if we add tags c_0, c_0' to messages 2 and 2'.

Similar results on other protocols of the literature (Otway-Rees, Yahalom, . . .).

JFK, joint work with Martín Abadi and Cédric Fournet

JFK is a complex protocol, candidate replacement for IPsec.
We have shown identity protection properties of JFK.

The variant JFKi protects the identity of the initiator I.

We consider a configuration in which two initiators I_1 and I_2 can have their secret keys among sk_1 and sk_2 and show

$$(\nu sk_1)(\nu sk_2)\sigma P \approx (\nu sk_1)(\nu sk_2)\sigma' P$$

where P models the protocol, σ and σ' map the variables sk_{I_1} and sk_{I_2} (secret keys of I_1 and I_2 respectively) to elements of $\{sk_1, sk_2\}$.

(This proof uses some extensions described in the paper.)

Conclusion

- A flexible, fully automatic, efficient system for proving strong secrecy.

- Unbounded number of sessions and message space.

- Terminates on a large class of protocols (tagged protocols).

- Future work: extend to more general process equivalences.